3-23-2018

Pattern-of-Life Modeling using Data Leakage in Smart Homes

Steven M. Beyer

PATTERN-OF-LIFE MODELING USING DATA LEAKAGE IN SMART HOMES

THESIS

Steven M. Beyer, Capt, USAF

AFIT-ENG-MS-18-M-009

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-18-M-009

PATTERN-OF-LIFE MODELING USING DATA LEAKAGE IN SMART HOMES

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 Computer Engineering

Steven M. Beyer, B.S.E.E.

Capt, USAF

March 2018

AFIT-ENG-MS-18-M-009

PATTERN-OF-LIFE MODELING USING DATA LEAKAGE IN SMART HOMES

Steven M. Beyer, B.S.E.E.
Capt, USAF

Committee Membership:

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

Scott R. Graham, Ph.D.
Member

Maj Jason M. Bindewald, Ph.D.
Member

AFIT-ENG-MS-18-M-009

Abstract

In recent years, smart home devices have become one of the most popular categories in the Internet of things (IoT). Smart devices are relatively inexpensive, readily available, and easily integrated into homes and offices. As smart technologies become more prevalent, consumers must make informed purchasing decisions as retailers provide IoT devices from manufacturers with little scrutiny in regards to device security or known vulnerabilities. In response to the growth and commonplace of IoT, the United States Government Accountability Office released a report in July 2017 to congressional committees stressing that further assessments and guidance are needed to address security risks of IoT in the Department of Defense (DoD) [1]. Specifically, there must be investigation into the security ramifications IoT devices have on operations security, intelligence collection, and leadership safety.

Wi-Fi and Bluetooth Low Energy (BLE) are two protocols increasingly used in a range of IoT devices such as security cameras, locks, and motion sensors. There are two characteristics that cause these devices to inadvertently leak privacy information: (i) they continuously broadcast unencrypted information, such as Wi-Fi Media Access Control (MAC) addresses or BLE device names, which anyone with a properly-tuned receiver can observe, and (ii) IoT devices send unique and predictable wireless traffic in the clear during communications. This research shows how data leakage from these protocols, combined with device vulnerabilities, enable an eavesdropper to collect wireless traffic from IoT devices in a smart home or office to identify devices, track user movements, identify events, and ultimately gain physical access to the home.

To demonstrate these capabilities, a Smart Home Automation Architecture (SHAA) was designed by integrating a variety of commercial off-the-shelf (COTS) Wi-Fi and

BLE devices with Apple's home automation application, HomeKit. SHAA provides real smart home traffic that is used to investigate IoT data leakage in the wild. Furthermore, a device classifier and pattern-of-life analysis tool, CITIoT (Classify, Identify, and Track Internet of Things), was developed to exhibit how an eavesdropper can utilize data leakage to classify devices, identify events, and track users. CITIoT operated against SHAA during a five day trial in which a user activated devices within the smart home. During this experiment, CITIoT was able to capture traffic from the smart home network and classify 17 of 18 devices, identify 95% of the 343 events that occurred, and track when users were home or away with near 100% accuracy. CITIoT only identified an average of 3 false positives per day.

Additionally, a data leakage mitigation technique was created that introduces spoofed wireless traffic sent on behalf of IoT devices to inhibit CITIoT's ability to classify devices, identify events, and track users. The MIoTL (Mitigation of IoT Leakage) tool was tested during an additional five day experiment; during these trials, CITIoT was unable to identify motion sensor and camera devices and was inundated with an average of 221 false positives per day that made CITIoT ineffective at identifying real events. Also, CITIoT was only able to recognize 8 minutes of 24 hours that the user was away from the smart home. MIoTL made CITIoT ineffective at classifying devices, identifying events, and tracking when the user was away from the home.

This research closes by stressing how data leakage, combined with device vulnerabilities, can be used to recognize if a user is away and crack a Bluetooth lock to gain access to the home or office. The security implications of IoT devices on military operations are discussed and the need for ongoing evaluation of IoT in the DoD is emphasized. Lastly, operational security recommendations are provided to defend against presented vulnerabilities and create a safer smart home and office.

Acknowledgements

Faithless is he that says farewell when the road darkens.

-J.R.R. Tolkein, The Fellowship of the Ring

I am grateful to AFIT and the USAF for letting me participate in this unique and rare opportunity.

I have learned much in this program thanks to the dedication and commitment demonstrated by AFIT faculty members like Dr. Barry Mullins, Dr. Scott Graham, and Maj Jason Bindewald. This would have gone poorly without their support, encouragement, and guidance. I am also indebted to a number of colleagues and students whose friendship and professional collaboration were invaluable throughout this process.

Lastly, I am grateful for my beloved wife. The support you provide day in and day out is unparalleled and beyond what I could ask for. Without you, none of this would be possible.

Steven M. Beyer

Table of Contents

Page

Page

Page

List of Figures

Figure Page

List of Tables

List of Acronyms

Abbreviation Page

Abbreviation Page

PATTERN-OF-LIFE MODELING USING
DATA LEAKAGE IN SMART HOMES

I. Introduction

1.1 Background

In recent years, smart home devices have become one of the most popular categories in the Internet of things (IoT), accounting for $4.5 billion of a $351 billion industry; over 40.8 million smart home devices are expected to ship in 2018, a 41 percent increase over 2017 [2]. Informed purchasing is of primary concern as retailers provide smart home devices from manufacturers with little scrutiny in regards to device security or known vulnerabilities. These devices are relatively inexpensive and can be purchased, shipped, and integrated into a smart home in days; there is a low threshold of entry. As smart home technologies become more popular and easier to obtain, the increased prevalence of IoT devices in the home necessitates the need for investigation into what kind of privacy information these devices inadvertently broadcast, what vulnerabilities exist, and how privacy leakage can be used against consumers. More importantly, with the rise of vulnerable smart home devices available to consumers, defenses need to be researched and implemented.

1.2 Problem Statement

Wi-Fi and Bluetooth Low Energy (BLE) are two protocols increasingly used in a range of IoT devices such as security cameras, locks, medical devices, sensors, and a myriad of other devices. These protocols broadcast some information in the clear

that anyone with a properly-tuned receiver can observe. Smart home devices using these technologies are at risk of leaking privacy data that an outside observer may use to infer facts about the smart home such as what devices are in the home and when is the user away. Foremost, smart home owners must be informed of potential physical security implications IoT devices can introduce to their home. For example, installing a vulnerable BLE lock can allow attackers unfettered access to the home. Consumers must also have a way to defend against privacy leakage in their homes and mitigation methods need to be developed. The problem statements this thesis answers are what kind of privacy data do smart home devices leak, how can an attacker exploit this leakage to gain physical access to a home, and are there ways to defend against these vulnerabilities?

1.3 Research Goals

This work attempts to investigate the problem of data leakage in smart home devices and to what extent privacy information is sent in the clear. It observes what reconnaissance methods an eavesdropper can use to collect wireless traffic from devices without being connected to the smart home environment and what kind of knowledge can be inferred about the smart home and its users. After data analysis, this research attempts to discover what physical security ramifications this leakage introduces. It also attempts to defend against data leakage by mitigating the weaknesses in vulnerable devices.

1.4 Hypothesis

This research hypothesizes that IoT device leakage and vulnerabilities can be used to create a tool that classifies smart home devices, tracks a user's presence within the home, identifies events, and enables an eavesdropper to gain physical access to the

home. It also theorizes that the capabilities of this tool can be mitigated by deploying existing methods and techniques in a novel way to conceal devices and events within the smart home and make it appear that users are always home.

1.5 Approach

A smart home environment is created to provide realistic wireless traffic for investigation—a voice activated digital assistant and IoT architecture is developed by integrating a variety of commercial off-the-shelf (COTS) Wi-Fi and BLE devices with Apple's home automation application, HomeKit. Furthermore, a device classifier and pattern-of-life analysis tool is created to analyze data leakage within the smart home architecture attempting to classify devices, identify events, and track whether a user is in the home or away. A defense tool is designed to mitigate vulnerabilities by minimizing and shielding data leakage. Lastly, findings are synthesized to observe physical security implications.

1.6 Assumptions/Limitations

The following assumptions/limitations are understood when designing and executing the device classifier and pattern-of-life analysis tool:

- The devices selected and smart home architecture are representative of a realistic environment.
- Wi-Fi device categories are limited to the following: outlet, sensor, and camera.
- All Wi-Fi devices must be compatible with the Homebridge server.

1.7 Contributions

This thesis contributes to the field of IoT security, specifically privacy within a smart home through five principal contributions:

1. **Smart home architecture.** To analyze IoT data leakage in the wild, a realistic Smart Home Automation Architecture (SHAA) is provided that integrates Wi-Fi and BLE COTS devices with Apple's home automation application, HomeKit.

2. **Vulnerability Analysis.** This work explains how an eavesdropper can use device vulnerabilities, characteristic data exchanges, and packet sizes to create a classifier able to identify components and events within the smart home environment.

3. **Classification, Identification, and Tracking of Internet of things (CITIoT).** It presents a tool that demonstrates four capabilities enabled by data leakage: network mapping, device classification, event identification, and user tracking.

4. **Mitigation of IoT Leakage (MIoTL)**. It provides a defensive tool that uses existing techniques in a unique way to mitigate the smart home device leakage capabilities developed in CITIoT.

5. **Synthesis.** It stresses the importance of smart home and office operational security by demonstrating how CITIoT can be used to gain physical access to a home or office when a user is away. It also emphasizes the security implications these vulnerabilities present to Department of Defense (DoD) operations.

1.8 Thesis Overview

This thesis document is arranged in six chapters. Chapter 2 provides a brief summary of relevant wireless protocols, an outline of open-source security analysis tools used, and other relevant research. Chapter 3 presents the system design details, smart home architecture, and mitigation techniques developed to analyze, exploit,and prevent smart home privacy leakage. The experiment methodology and the analysis of results are presented in Chapters 4 and 5 respectively, while Chapter 6 summarizes the research and discusses opportunities for future work in this domain.

II. Background and Related Research

2.1 Overview

This chapter provides a technical summary of the Wi-Fi and BLE protocols to exhibit characteristics of these technologies that enable data leakage and vulnerabilities in smart home devices. It also provides a brief overview of other comparable wireless protocols. It follows with an outline of the current state of IoT and smart home security, a survey of open-source tools used in this work, and discussion of related research.

2.2 Wireless Protocols

2.2.1 Wi-Fi.

The 802.11 wireless specification defines the physical and link layers for communication in the 2.4 GHz radio band [3]. The 802.11 architecture contains four major physical components: (i) access points (APs), (ii) wireless medium, (iii) stations (devices), and (iv) distribution systems (i.e., router) [4]. In a typical home Wi-Fi network, the AP and router are combined into one unit which is used to connect the network to the Internet. During setup of a secure network, such as the one analyzed in this work, the AP is assigned a service set identifier (SSID), channel number, and password. Each wireless station must prove knowledge of the password to associate with the AP and communicate within the network. Association to a secure network results in the encryption of wireless traffic sent within the network if encryption is enabled. Figure 1 depicts the frame format for the MAC Protocol Data Unit (MPDU), the unit of data exchanged between entities using the physical layer (wireless medium). It also shows which portions of the packet are encrypted when

connected to a secure wireless AP. Figure 2 provides the fields within the Media Access Control (MAC) Header, which include addressing information for Wi-Fi packets at the link layer. Only the first three addresses pertain to this work and indicate the destination address (DA) (Address 1), source address (SA) (Address 2), and basic service set identifier (BSSID) (Address 3) MAC addresses [3]. The MAC Header is not encrypted and is key to enabling the data leakage capabilities presented in this work.

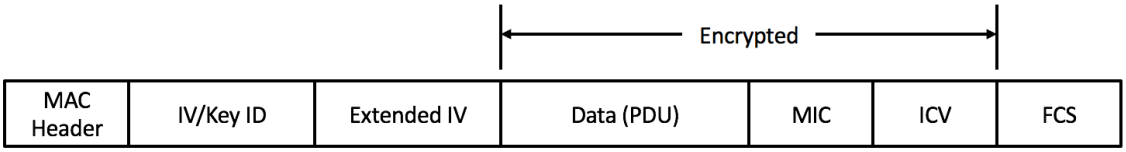


Figure 1. MPDU format when using WPA2 [3]

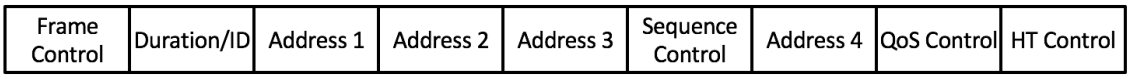


Figure 2. MAC header frame format [3]

A client station uses a Wireless Network Interface Controller (WNIC) to collect wireless traffic. In normal operation, a WNIC only passes traffic destined to the client station, dropping all other packets [4]. There are two other modes in which particular WNICs can operate: promiscuous mode and monitor mode. Promiscuous mode sets the WNIC to pass traffic with a BSSID of the associated AP to the client station. Monitor mode, used throughout this work, sets the WNIC to pass any traffic to the client station regardless of the DA or BSSID [5].

Two other wireless characteristics used by the data leakage tool presented in this work include the time of packets and the frame size (FSize), or packet size. The packet timestamp is calculated by the host kernel, while the receiving application determines the packet size [6].

2.2.2 BLE.

The Bluetooth Special Interest Group (SIG) introduced BLE (Bluetooth Smart) in Bluetooth Core Specification v4.0 to complement Bluetooth basic rate/enhanced data rate (BR/EDR) (Bluetooth Classic) [7]. Although these two implementations share some key attributes (e.g., both operate in the 2.4 GHz band and use adaptive frequency hopping), they are different protocols with unique design goals [8]. While Bluetooth Classic is used in high-bandwidth applications, such as transferring files or streaming audio, BLE is designed to minimize power, cost, and data rate. These goals are accomplished by limiting overhead at every level of the architecture and using simple communication protocols. To this same end, BLE devices predominantly transmit state data in short, infrequent bursts. These characteristics make BLE ideal for IoT applications where battery life is a top priority.

Bluetooth Core Specification v5.0 was adopted in December 2016 [9], however, during initial device investigation, it was observed that the majority of commercially-available devices still used v4.2 at the time of this research (the Apple iPhone 8 was the first iPhone to implement v5.0 and was not released until September 2017 [10]); the devices investigated throughout this thesis use v4.2 or older. The rest of this section discusses elements of the BLE architecture (shown in Figure 3) and protocol which are relevant to the data leakage investigation of this research.

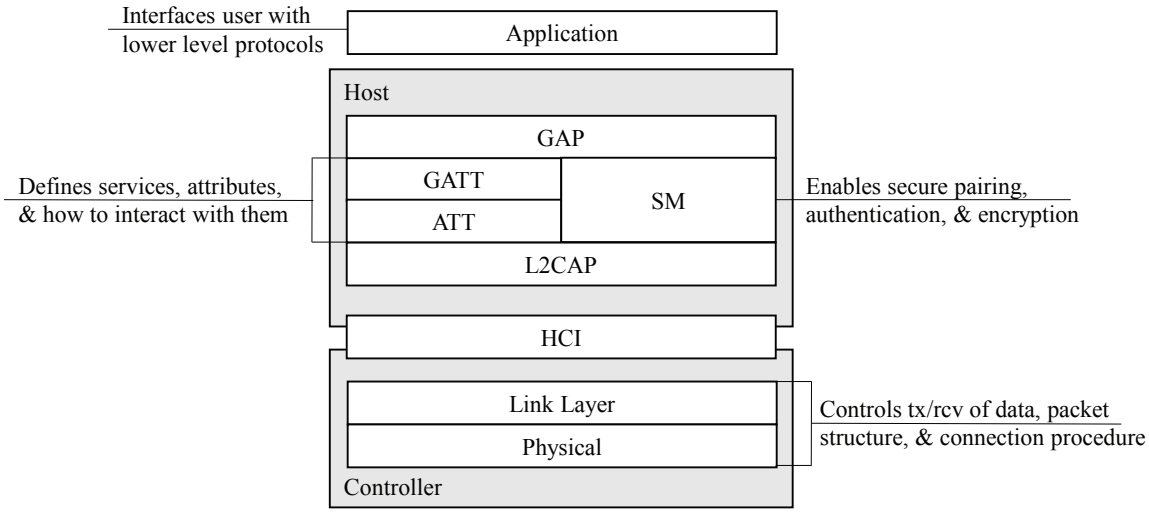


Figure 3. The BLE architecture

2.2.2.1 Attribute Protocol.

Data is communicated between BLE devices in the form of "attributes" using a client-server architecture ruled by the Attribute Protocol (ATT). Each attribute contains state information addressed by a unique handle and type. These attributes are then grouped into characteristics based on discovery method and accessibility. The master device (e.g., smartphone or computer) typically acts as the client, periodically reading/writing information in the form of attributes from/to the server (e.g., locks, sensors) as required by a user. For example, a user (client) may request the status of a door lock using an ATT Read Request with the type "Lock Status" and handle 0x0019; the device (server) then responds with an attribute containing the value "Unlocked." When the user decides to lock the door, an ATT Write Request is sent with the value set as the user's password; if a valid value is provided, the lock changes state to "Locked." The attack presented in Section 6.3 uses the ATT commands and Generic Attribute Profile (GATT) characteristics BLE devices use to communicate. Commands are passed from the application to the physical layer using the Logical Link

Control and Adaptation Protocol (L2CAP) and The Host/Controller Interface (HCI).

2.2.2.2 Security Manager.

The Security Manager (SM) defines a process called pairing and bonding to secure BLE connections [8]. When a device wants to create a new connection in which security parameters have not been previously exchanged or have been forgotten, the devices must first establish a trust relationship through the pairing process. While there are application-specific ways to implement the SM, generally, pairing is accomplished by the devices exchanging pairing information, authenticating each other, encrypting the link, and then sharing keys. After pairing is accomplished, bonding is simply saving the keys for faster connection establishment in the future. If either device loses the encryption keys, the entire pairing and bonding sequence must be re-accomplished. The security of a connection is dependent on how the SM is implemented. Many of the devices examined in this work are vulnerable due to poor SM implementations that did not enforce encryption or authentication.

2.2.2.3 Physical and Link Layers.

The physical and link layers control device discovery, establishing connections, packet structure, and transmitting/receiving data. For a connection to occur, one device advertises its presence while another scans. When the scanning device sees the correct advertising device, a connection is created. These advertisements and the overall connection process are key to observing pattern-of-life information via BLE sniffers. Connections occur between one master and one slave and are broken up into a series of connection events (CEs) with the master transmitting packets during a CE and each CE occurring on a different channel. The connection parameters are set by the master in the connection request packet and include the frequencies to be used

and CE interval. Each step of this process is shown in Figure 4 and described.

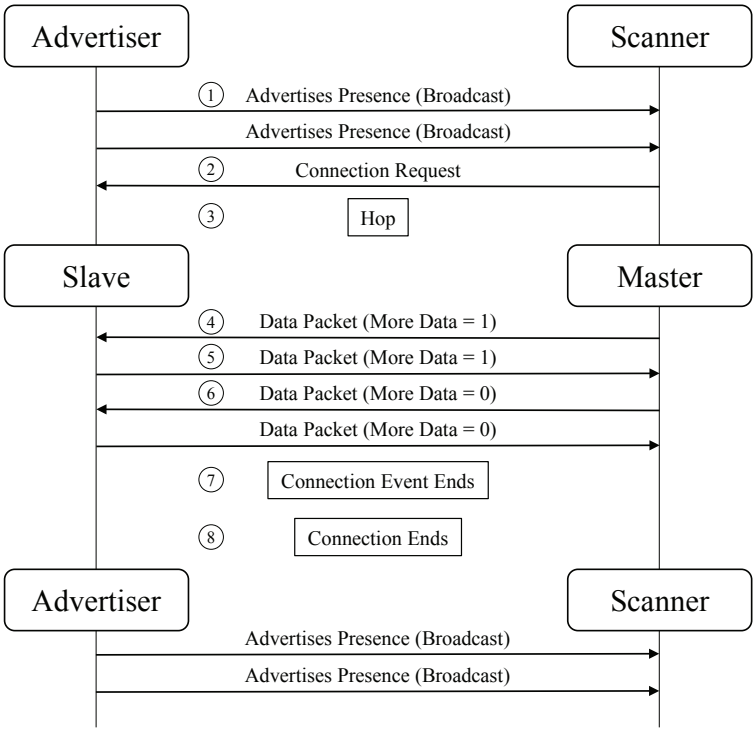


Figure 4. The BLE connection process

1- Device Advertises Presence. A connection begins with a slave announcing its presence by broadcasting Advertising Indication (`ADV_IND`) packets on three advertisement channels (see Figure 6). Each `ADV_IND` packet includes device information such as connectability, scannability, services provided, and the name of the device. `ADV_IND` packets also include a "TxAdd" bit that indicates if the advertiser is using a public or random address. A master actively or passively scans the advertisement channels detecting connectible devices. Depicted in Figure 5, active scanning is a key concept for the data leakage discussion in this thesis and occurs prior to the connection request shown in Figure 4. When actively scanning, a master observes an advertising packet and, if the device is scannable, sends a Scan Request (`SCAN_REQ`) packet to the device. The advertiser sends a Scan Response (`SCAN_RESP`) packet back with more information, typically expanding on the device name and possibly includ-

ing broadcast data such as battery level. A master can only connect to a device that advertises its presence and is connectable.

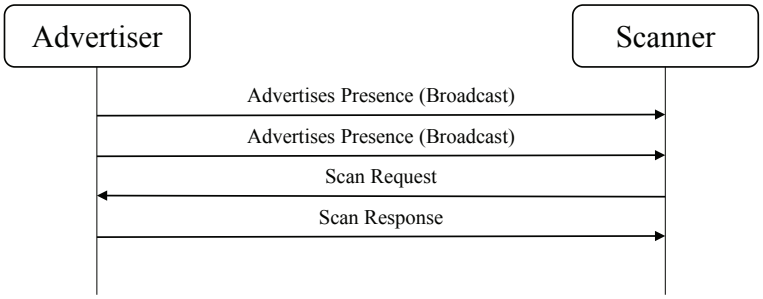


Figure 5. Active scanning process

2- Initiator Sends Connection Request. Once a scanner observes a connectible device, a Connect Request (`CONNECT_REQ`) packet is sent to the advertiser. This packet establishes all of the necessary parameters to start the connection to include the access address, connection interval, transmit window size and offset, hop interval, channel map, and sleep clock accuracy (SCA). The access address is a random value used to identify packets that are part of the connection. The master may update a subset of these parameters at any time in a connection parameter update message. In this message, the master provides a future time at which the new parameters will take effect. To follow a connection, the sniffer used in this work must observe and implement all of the connection parameters and potential changes throughout the connection.

3- Hop. As shown in Figure 6, the BLE frequency band is divided into forty channels separated by 2 MHz. These frequencies are distributed into three advertising channels and thirty-seven data channels. When in a connection, a master and slave communicate on one channel per CE. After each CE, both the master and slave hop to a new frequency per the Channel Map, Hop Increment, and hopping algorithm; these parameters are established by the master at the beginning of a connection or in a parameter update and are non-negotiable.

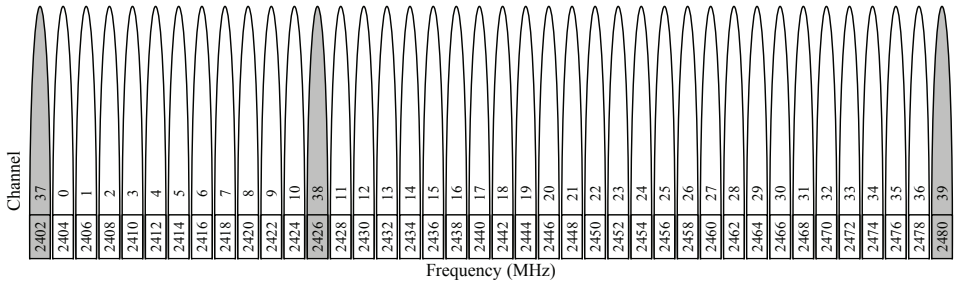


Figure 6. BLE channel mapping; darker channels represent advertisement channels

4- Master Sends Data Packet. The first data packet sent in a CE is called an anchor packet which establishes the timing for all future CEs.

5- Slave Responds. The slave must always respond to a received data packet from the master unless two consecutive packets are received with an invalid cyclic redundancy check (CRC). However, to conserve energy, the slave does not have to listen to a predetermined number of CEs.

6/7- Packets are Sent until Connection Event Ends. The length of a CE is, at most, the predetermined connection interval, but may be shorter depending on how much data needs to be transmitted. If the slave is listening to the CE and responds to the anchor packet, the master and slave exchange packets until a CE end condition is met. A master can send empty packets to maintain the connection. There are four ways to close a CE—if neither device has more data to send (indicated by the more data bit in the packet); if the more data bit is set and either slave or master do not receive a subsequent packet within 150 μs; if two consecutive packets are received with an invalid CRC; or if the connection interval is reached. CEs can be likened to bursts of data and each CE ends at the end of the burst while the overall connection is still maintained.

8- Connection Ends. A connection continues until either device sends a terminate indication packet, no packets are received within the supervision timeout, or the message integrity check fails. After a connection ends, the slave resumes advertising

its presence.

2.2.3 Other Wireless Protocols.

Other protocols used in IoT applications include those based on IEEE 802.15.4 (e.g., ZigBee) and on the ITU-T G.9959 recommendation (e.g., Z-Wave) [11]. These wireless protocols have many of the same privacy leakage issues found in BLE and Wi-Fi. Similar to BLE, these protocols have proper encryption, but do not encrypt the physical layer [11]. This creates unique security challenges for wireless broadcast networks in which anyone with a properly-tuned receiver can see these data packets. Also, the physical and link layers for each of these protocols inherently advertise legitimate information before and after a connection is established that can be used by an attacker.

Like BLE, Zigbee advertises MAC addresses, which has been used to infer whether a person is in a room or not [12]. Z-Wave provides source, destination, and home identification information in the clear that can be used in reconnaissance and device tracking [13].

Efforts are being made to provide techniques to limit the amount of data leakage by these protocols. Some examples include periodically changing MAC addresses, encrypting lower-layer data packets, and not setting devices in active service discovery mode [14]. This work seeks to observe and prevent privacy leakage in BLE and Wi-Fi through the understanding of IoT leakage and design of mitigation tools.

2.3 Smart Home Technologies

A list of smart home technologies relevant to the data leakage and mitigation work is provided:

- **Devices**: BLE or Wi-Fi devices such as switches, smart outlets, cameras, and

sensors. Devices can be connected to and controlled by controllers.

- **Controllers**: A master device such as an iPhone or Android phone that connects to a device within the smart home to get status updates or change states.
- **Hub**: A system that sits on the home network, connects to different devices via the manufacturer application programming interface (API), and exposes control of the devices via a centralized application on the controller. Hubs often provide access to the devices while a user is away from the smart home. Examples of hubs include Apple's HomeKit and the open-source server Homebridge.
  - **Apple's HomeKit**: A hub that provides a controller with voice control and automation capabilities for devices.
  - **Homebridge**: An open-source server that provides integration of some IoT devices with HomeKit. Added as a hub in the HomeKit, it allows a user to use Siri to control devices that are not typically supported within HomeKit.
- **Applications**: Many smart home devices require proprietary applications to interact with the device's full range of capabilities. A controller must use these applications to control the device.

2.4 Tools

Table 1 provides a list of open-source tools used to analyze device data leakage and vulnerabilities.

Table 1. Wi-Fi and BLE tools used in this research

Tool Name	Version	Description
Ubertooth One	Firmware: 2017-03R2	Bluetooth sniffer with open-source firmware and hardware [15]
BlueZ	5.43	Linux Bluetooth stack with utilities to scan for BLE devices and transmit packets [16]
Plugable USB Bluetooth Adapter	2.0	Commercial Broadcom BCM20702-based Bluetooth adapter to communicate with Bluetooth devices with 33 ft range [17]
Alfa Card	AWUS036ACH	802.11ac Wireless Adapter
Airodump-ng	Aircrack-ng 1.2	Wi-Fi network security tool to capture raw 802.11 frames
Python	2.7.10	Programming language used in scripting
Pyshark	0.3.7.8	Python wrapper allowing Python packet parsing with Wireshark dissectors [18]
Scapy	2.3.3	Interactive packet manipulation tool used to send or receive 802.11 packets [19]

2.5 Related Research

Although Wi-Fi and BLE smart home devices are becoming commonplace, the privacy leakage and security vulnerabilities of these devices is largely unexplored. In 2016, Ed Skoudis presented a voice-controlled and automated IoT smart office architecture, J.A.R.V.I.S. [20]. J.A.R.V.I.S. represents a way forward for smart homes by integrating Wi-Fi devices, Apple's Homekit, and automation, but Skoudis admits that security was an afterthought in developing the architecture. At the end of his presentation, Skoudis challenged developers to explore the security implications of the

growing IoT field. The SHAA developed in this work is influenced by Skoudis' work, but extends on it by expanding on the number of devices, including BLE devices, and integrating a privacy leakage mitigation method. This work also explores the privacy consequences of a smart home architecture such as J.A.R.V.I.S. by analyzing privacy leakage in BLE and Wi-Fi devices.

While the BLE specification defines security procedures to encrypt the payload, generate private addresses, and provide authentication [21], implementation of the SM is left up to the developer; each additional security measure contributes to increased energy consumption [8]. Limiting power, developing devices quickly, and other design constraints drive developers toward poor implementation of the SM, leaving devices with essentially no Link Layer authentication or encryption.

Recently, oversight in Link Layer security has enabled researchers to crack twelve BLE locks from up to a quarter mile away [22]. Two man-in-the-middle frameworks were developed due to the lack of Link Layer security that allow home automation denial of service, data manipulation, and command injection [23][24]. The lack of lower-layer security employment also creates vulnerabilities in firmware update procedures; a team of researchers were able to upload customized firmware onto a BLE industrial monitor that then provided false sensor readings or locked out legitimate users [25]. Similarly, a lack of encryption enables unintended privacy leakage. In a few recent studies focused on BLE wearable fitness trackers, one group of researchers observed device address and connection information sent in the clear that enabled them to identify users based off of activity level and gait [26], while another group used device addresses and received signal strength indicator (RSSI) information to track a user wearing a Fitbit Surge up to 1,000 meters away with greater than eighty percent accuracy [27]. Privacy data has also been used to create pattern mining models to track tourist attraction visits in Belgium to help determine the best locations

to put hotels [28].

Privacy leakage in Wi-Fi has likewise been exploited in recent research. Researchers have used Wi-Fi MAC addresses sent in the clear and RSSI values to create location tracking systems on campuses, crowd tracking tools at mass events, and in customer relationship management (CRM) allowing commercial businesses to track customer interactions and data [29][30][31]. A group from the United Kingdom was able to use raw Wi-Fi signals to create fingerprinting techniques able to identify applications used on a mobile phone [31]. One researcher was able to use raw Wi-Fi signals to activate alerts when a security camera observes motion [32]. This research, however, did not look at other types of smart home devices, observe how an attacker may use this traffic, or provide methods of mitigation.

In response to protocol vulnerabilities, a few different efforts have been made to increase Wi-Fi and BLE security and privacy. For Wi-Fi, this includes periodically changing MAC addresses, randomizing FSize, and encrypting lower-layer data packets. M. Gruteser and D. Grunwald provide a framework to change MAC addresses frequently while still maintaining wireless connectivity [33]. The technique of chaffing and winnowing, as introduced by R. Rivest, can be adapted in smart home technologies to send legitimate packets intertwined with fake packets of random size to make events impossible to identify [34]. In BLE, devices need to make their advertisements private. Fawaz et al. designed an authentication system, BLE-Guardian, that restricts who can discover, scan, and connect to BLE devices [35]. BLE-Guardian uses jamming techniques to hide advertisements from unauthorized users. It is limited, however, to protecting devices prior to a connection and does not hide packets that are transmitted after a connection is created. As privacy data is still leaked during a connection, much of the reconnaissance information mentioned above can still be collected by an attacker. With BLE-Multi, Gutierrez et al. developed an enhance-

ment to the Ubertooth One BLE scanner that enables sniffing of multiple connections simultaneously [36]. However, the scanner is limited to tracking three connections simultaneously and only saw an eighty-five percent probability of successful packet capture.

The Air Force and DoD are not immune to data leakage or device vulnerabilities and their impact on operational security is a topic of recent discussion. A group of researchers from the Naval Postgraduate School observed how the growth of mobile devices used in deployed operations has introduced several potential security threats such as rogue Wi-Fi APs and location tracking [37]. Additionally, in January 2018 it was revealed that the fitness-tracking application, Strava, provided heat maps of user activity around the world [38]. These maps, correlated with user selected route names and operating locations of military personnel in the Middle East, revealed sensitive information such as troop exercise paths, patrol routes, and forward operating base perimeters. In response to the growth of IoT and corresponding threats to operational security, the United States Government Accountability Office released a report in 2017 to congressional committees stressing that further assessments and guidance are needed to address security risks of IoT in the DoD [1]. They note that there are generally two types of risks for IoT devices: (i) risks with how the devices are used, such as operational risks like unauthorized communication of information to third parties, and (ii) risks with the devices themselves, such as lack of encryption. From these findings, the report identifies a research gap: the DoD has not accomplished assessments related to the impact of IoT devices on operations security, intelligence collection, and leadership safety. This thesis analyzes both types of risks and fills the research gap by demonstrating how IoT device leakage and vulnerabilities enable an eavesdropper to track users, crack a smart lock, map Wi-Fi networks, and identify motion within smart environments—ultimately, it stresses security implications of

IoT on military operations.

2.6 Background Summary

This chapter presents a brief technical summary of the Wi-Fi and BLE protocols and how their security features relate to those of other comparable wireless protocols. It provides background on key smart home technologies and open-source tools as they pertain to this work. It observes related research into the development of automated smart home architecture, how BLE and Wi-Fi properties leave them open to privacy leakage, and current efforts in securing IoT. While research has been done in the realm of Wi-Fi and BLE privacy leakage, little work has provided a broad review of privacy leakage from smart home devices in the wild or methods to secure smart homes from data leakage. This thesis contributes to the field of IoT security, specifically privacy within a smart home, by illustrating how devices leak data and demonstrating how users can prevent leakage through mitigation techniques.

III. SHAA, CITIoT, and MIoTL Design

3.1 Overview

This research introduces three novel contributions in analyzing and mitigating IoT data leakage in smart home environments: a Smart Home Automation Architecture (SHAA), the Classification, Identification, and Tracking of Internet of things (CITIoT) tool, and the Mitigation of IoT Leakage (MIoTL) tool. SHAA is a testbed that provides realistic smart home traffic by integrating Wi-Fi and BLE COTS devices with Apple's home automation application, HomeKit. To analyze privacy leakage within SHAA, the CITIoT tool is used to classify IoT devices, identify IoT events, and track smart home users. The MIoTL tool supplies mitigation techniques that neutralize aspects of CITIoT. This chapter provides a detailed description of SHAA, each component of the CITIoT and MIoTL tools, and their respective roles within the experiment.

3.2 System Summary

Figure 7 displays a simplified system diagram for all components and their interactions. The boundaries of the system are limited to these components.

3.3 Smart Home Automation Architecture (SHAA)

SHAA is developed to provide real IoT traffic to be analyzed by the system under test. As depicted in Figure 8, SHAA includes three controller components and various connected devices. The controller components include (i) a Raspberry Pi running the Homebridge server that emulates the iOS HomeKit API and exposes supported devices to Apple's HomeKit, (ii) an iPhone 6+ running Apple's HomeKit and device specific applications, and (iii) an Apple TV Generation 2 acting as a smart home

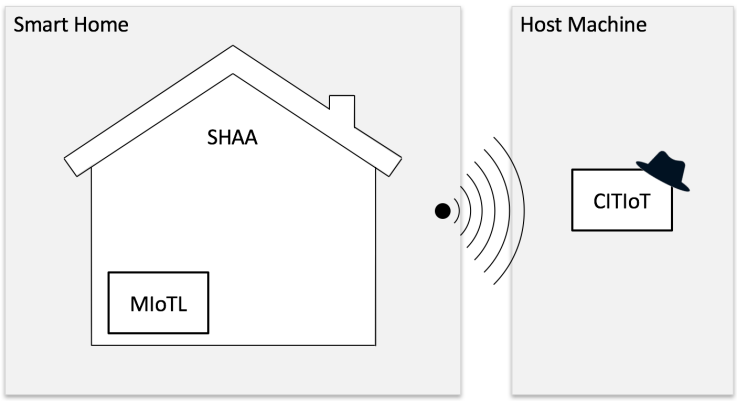


Figure 7. Overall system diagram

hub to allow access to HomeKit supported devices while the user is away from the smart home. The communication between controllers and devices can be observed in Figure 8 and is described in the rest of this section.

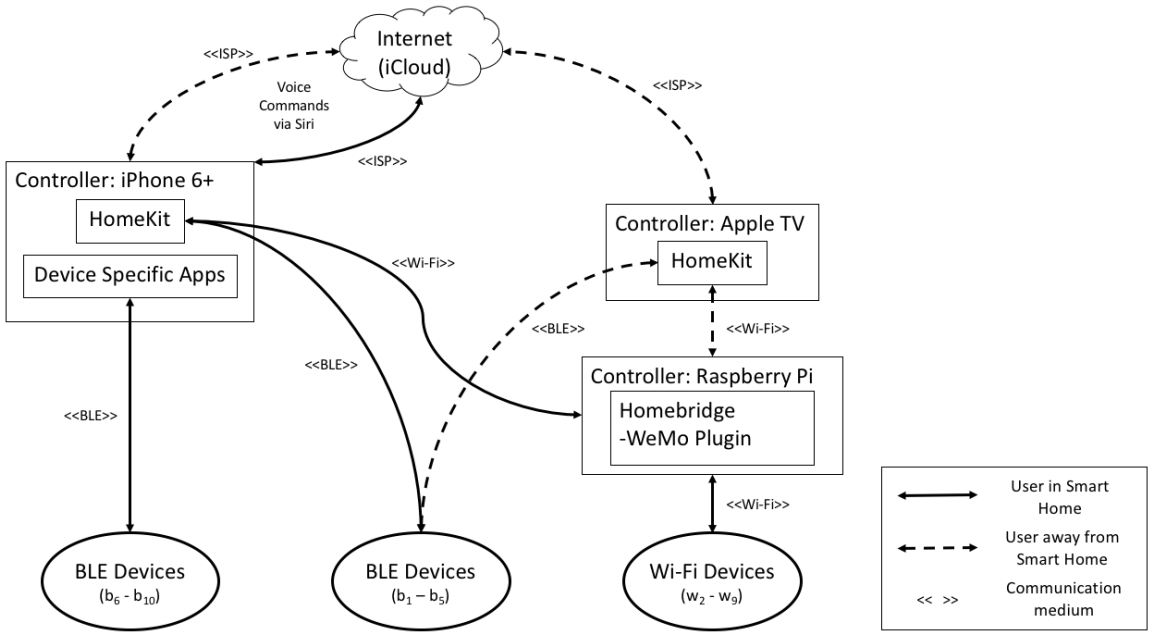


Figure 8. Diagram of SHAA components

3.3.1 Raspberry Pi.

The Raspberry Pi 3 Model B with Raspbian Jessie Lite version 4.9 operating system is connected to the smart home network via the on-board 802.11 b/g/n 2.4 GHz wireless chip [39]. The Raspberry Pi runs Homebridge version 0.4.14 as a systemd service and each interaction between a controller and device is logged in the systemd journal [40]. A Homebridge plug-in is utilized to enable communication between the Belkin devices in Table 2 and Apple's HomeKit on the iPhone 6+ and Apple TV [41].

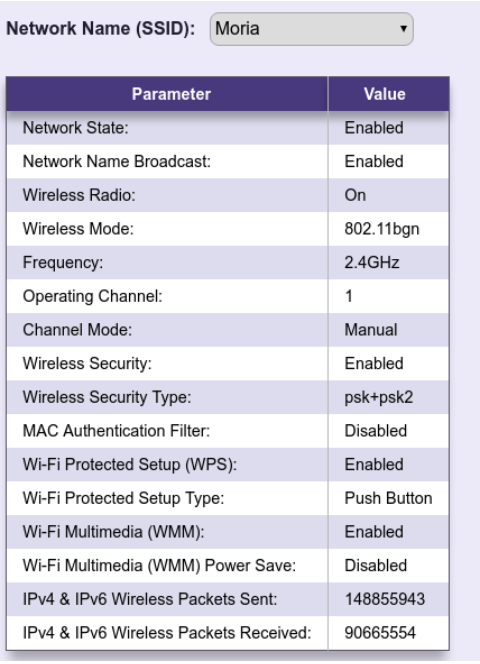
3.3.2 Apple Devices.

The iPhone 6+ and Apple TV act as controllers in the smart home architecture and connect to devices via Wi-Fi and BLE. When the user is home, the iPhone connects to Wi-Fi devices via the Homebridge server on the Raspberry Pi and connects directly to BLE devices. Some of the BLE devices are not supported by Apple's HomeKit and can only be accessed through the manufacturer provided iOS application on the iPhone. When the user is away from the smart home, the iPhone can communicate with HomeKit supported devices using an Internet service provider (ISP). For example, if the user is away from home and wants to access the temperature in a room, the iPhone interfaces with the Apple TV via the iCloud and the Apple TV will communicate with the device in the home via the Homebridge server for Wi-Fi or directly for BLE. This only works with HomeKit supported devices, therefore, BLE devices b_7-b_{12} (see Table 3) cannot be accessed while the user is away from the home.

3.3.3 Wi-Fi Devices.

To facilitate Wi-Fi communication in the smart home architecture, a 2.4 GHz Wi-Fi AP, with the SSID set as "Moria", was setup with Wi-Fi Protected Access 2 (WPA2) security on channel 1. Figure 9 provides a complete list of settings. De-

vices w_2-w_9 are connected to the AP and are listed in Table 2. The smart home devices include a camera, six outlets (four smart outlets, one mini outlet, and one energy outlet), and a motion sensor. These devices use the Homebridge server to communicate with Apple's HomeKit on the iPhone.



Parameter	Value
Network State:	Enabled
Network Name Broadcast:	Enabled
Wireless Radio:	On
Wireless Mode:	802.11bgn
Frequency:	2.4GHz
Operating Channel:	1
Channel Mode:	Manual
Wireless Security:	Enabled
Wireless Security Type:	psk+psk2
MAC Authentication Filter:	Disabled
Wi-Fi Protected Setup (WPS):	Enabled
Wi-Fi Protected Setup Type:	Push Button
Wi-Fi Multimedia (WMM):	Enabled
Wi-Fi Multimedia (WMM) Power Save:	Disabled
IPv4 & IPv6 Wireless Packets Sent:	148855943
IPv4 & IPv6 Wireless Packets Received:	90665554

Figure 9. Moria AP settings

Table 2. Wi-Fi devices

ID	Manuf	Device Type	Device Name	MAC	IP Address
w_1	Calix	Wireless Router	Moria	EC:4F:82:73:D1:1A	-
w_2	Belkin	Camera	NetCam	EC:1A:59:E4:FD:41	192.168.1.44
w_3	Belkin	Outlet	Switch1	B4:75:0E:0D:33:D5	192.168.1.40
w_4	Belkin	Outlet	Switch2	B4:75:0E:0D:94:65	192.168.1.41
w_5	Belkin	Outlet	Switch3	94:10:3E:2B:7A:55	192.168.1.42
w_6	Belkin	Outlet	Switch4	14:91:82:C8:6A:09	192.168.1.7
w_7	Belkin	Motion Sensor	Motion	EC:1A:59:F1:FB:21	192.168.1.43
w_8	Belkin	Energy Outlet	Insight	14:91:82:24:DD:35	192.168.1.47
w_9	Belkin	Mini Outlet	Mini	60:38:E0:EE:7C:E5	192.168.1.51
w_{10}	Raspberry Pi 3B	Computer	Pi	B8:27:EB:09:1A:81	192.168.1.50
w_{11}	Apple	iPhone 6+	iPhone	A0:18:28:33:34:F8	192.168.1.4
w_{12}	Apple	TV 2	Apple-TV	08:66:98:ED:1E:19	192.168.1.54

3.3.4 Bluetooth Low Energy Devices.

A Bluetooth master must be present for Bluetooth communication to occur in the smart home architecture. In SHAA, the iPhone and Apple TV act as masters while each of the BLE devices are slaves. A list of BLE devices can be found in Table 3. Devices b_1-b_5 are HomeKit supported and can be accessed with voice commands via the iPhone 6+ or Apple TV. Devices b_6-b_{10} are not HomeKit supported and can only be accessed through their manufacturer specific applications on the iPhone 6+.

Table 3. BLE devices

ID	Manuf	Device Type	Device Name
b_1	Elgato	Indoor Temperature	Eve Room
b_2	Elgato	Outdoor Temperature	Eve Weather
b_3	Elgato	Motion Sensor	Eve Motion
b_4	Elgato	Outlet	Eve Energy
b_5	Elgato	Door Sensor	Eve Door
b_6	Instant Pot	Smart Cooker	Instant Pot
b_7	MPow	Lightbulb	Playbulb
b_8	ZKTeco	Lock	BioLock
b_9	BitLock	Lock	Bike lock
b_{10}	SafeTech	Gunsafe	Gunsafe
b_{11}	Apple	iPhone 6+	iPhone
b_{12}	Apple	TV 2	Apple TV

3.4 Classification, Identification, and Tracking of Internet of things (CITIoT)

CITIoT contributes four capabilities enabled by data leakage from smart home Wi-Fi and BLE devices: device classification, event identification, user tracking, and network mapping. Figure 10 depicts the CITIoT system diagram which can be summarized by six components: (i) reconnaissance and scanning, (ii) passive sniffing, (iii) data preprocessing, (iv) tracking, (v) classification, and (vi) network mapping. Components i and ii require user interaction, while components iii-vi are executed via Python scripts. The next sections provide a description of all components and their interactions.

3.4.1 Hardware.

The Host Machine is used to operate the software and sniffers. The host is a Hewlett-Packard 8570w with a 64-bit Intel Core i7 3270QM processor running at 2.60 GHz, 16 GB DDR3 (4 x 4 GB) RAM, 120 GB SATA Hard Drive, and using Kali version 2017.1 as the operating system. These specifications are considered when observing processing time in Section 5.3.1. Three BLE sniffers (Ubertooth One with

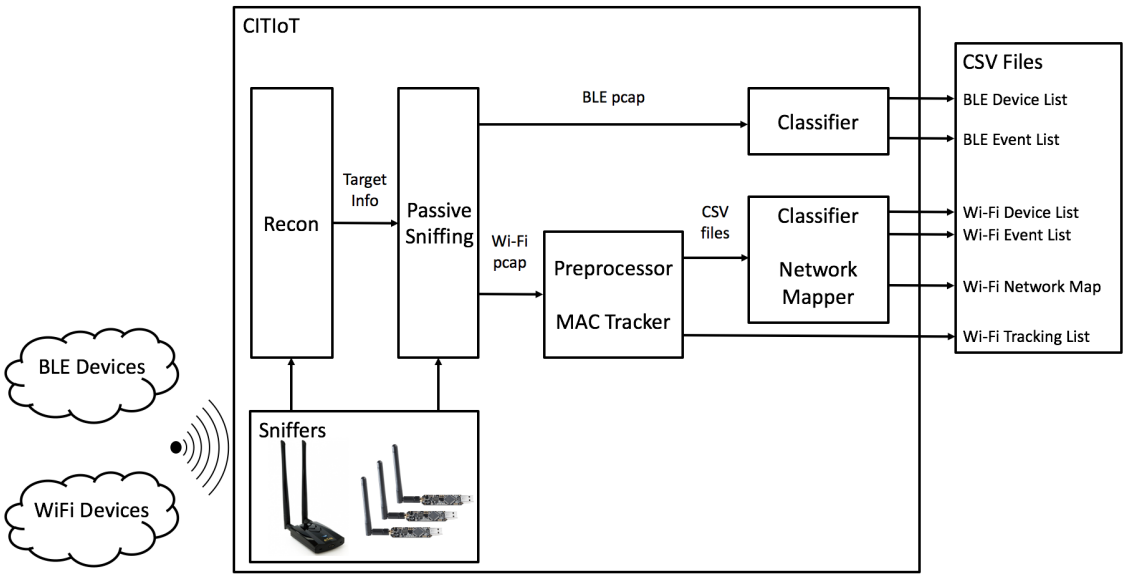


Figure 10. Diagram of CITIoT components and interactions

firmware 2017-03R2), the Plugable Bluetooth adapter, and a wide range 802.11ac dual band wireless adapter (Alfa Card AWUS036ACH) are connected to the host via Universal Serial Bus (USB). Each Ubertooth One device uses a 2.4 GHz 2.2 dBi antenna and connects to the host using USB 2.0, while the Alfa Card uses a 2.4 GHz and 5 GHz Dual-Band 5 dBi dipole antenna and connects to the host using USB 3.0.

3.4.2 Reconnaissance and Scanning.

Reconnaissance is necessary to ascertain five characteristics of the smart home network which CITIoT requires for operation: AP MAC address, AP channel, Wi-Fi device MAC addresses, BLE device names, and controller MAC addresses. Prior to beginning reconnaissance, the wireless interface that corresponds to the Alfa Card must be set to monitor mode to capture all Wi-Fi traffic regardless of the packet's BSSID or DA. Figure 11 shows the five commands used to set the interface to monitor mode: (i) kill any processes that may interfere with the Aircrack-ng tool, (ii) bring down the interface, (iii) set the interface to monitor mode, (iv) bring the interface

back up, and (iv) ensure the interface is in monitor mode.

```
root@gimli:gimli# airmon-ng check kill

Killing these processes:

  PID Name
  650 wpa_supplicant
  651 dhclient

root@gimli:gimli# ifconfig wlan1 down
root@gimli:gimli# iwconfig wlan1 mode monitor
root@gimli:gimli# ifconfig wlan1 up
root@gimli:gimli# iwconfig
lo        no wireless extensions.

wlan1     IEEE 802.11  Mode:Monitor  Frequency:2.412 GHz  Tx-Power=20 dBm
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Power Management:off

eth0      no wireless extensions.

wlan0     IEEE 802.11  ESSID:off/any
          Mode:Managed  Access Point: Not-Associated   Tx-Power=15 dBm
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Encryption key:off
          Power Management:off
```

Figure 11. Commands used to set Wi-Fi interface to monitor mode

Figure 12 shows the command used to operate the Alfa Card and the Airodump-ng tool to scan for Wi-Fi devices and APs. This scan identifies the target and smart home information needed for passive sniffing: (1) the target device's MAC address (e.g., iPhone 6+), (2) associated AP MAC address, (3) SSID of the smart home, and (4) AP channel.

Next, Figure 13 shows the command used to scan for devices connected to the smart home network using the Alfa Card and Airodump-ng tool while filtering on the target AP MAC address found in the previous step. The list of device MAC addresses associated with the target AP is collected and device manufacturers are discovered using an organizationally unique identifier (OUI) lookup tool [42]. Table 4 shows results from the OUI lookup using the Wi-Fi devices found during the scan.

```
root@gimli:gimli# airodump-ng wlan1

 BSSID              PWR  Beacons    #Data, #/s  CH  MB   ENC  CIPHER AUTH ESSID

 44:1C:A8:D5:E9:67  -63        2        0    0   6  54e  WPA2 CCMP   PSK  EIL
 EC:4F:82:73:15:AF  -46        2        0    0   1  54e  WPA2 CCMP   PSK  Buckeyes2.4
 EC:4F:82:73:D0:AE  -66        2        0    0   1  54e  WPA2 CCMP   PSK  EMF_411WS_106_2.4
 EC:4F:82:73:D1:1A  -35        2        0    0   1  54e  WPA2 CCMP   PSK  Moria
 EC:4F:82:73:15:B8  -65        2        0    0   1  54e  WPA2 CCMP   PSK  EMF_411WS_402_2.4
 B8:EE:0E:E9:82:7E  -60        3        0    0   1  54e. WPA2 CCMP   PSK  MySpectrumWiFi78-2G
 68:14:01:A8:E4:67  -49        3        0    0   1  54e  WPA2 CCMP   PSK  EWING-2.4
 B8:A1:75:23:60:43  -48        2        0    0   1  54e  WPA2 CCMP   PSK  <length: 22>
 EC:4F:82:73:17:0E  -67        2        0    0   1  54e  WPA2 CCMP   PSK  EMF_411WS_105_2.4
 EC:4F:82:73:D3:87  -52        3        0    0   1  54e  WPA2 CCMP   PSK  EMF_411WS_302_2.4

 BSSID              STATION            PWR   Rate    Lost    Frames  Probe

 EC:4F:82:73:D1:1A  A0:18:28:33:34:F8  -12    0 -24      3        5
```

Figure 12. Command and results to accomplish a scan of Wi-Fi devices and associated APs

This information is used to infer which devices are IoT devices (e.g., Belkin devices) and which are controllers (e.g., Raspberry Pi or Apple devices).

```
root@gimli:gimli# airodump-ng wlan1 --bssid ec4f8273d11a

 BSSID              PWR  Beacons    #Data, #/s  CH  MB   ENC  CIPHER AUTH ESSID

 EC:4F:82:73:D1:1A  -23      242       47    0   1  54e  WPA2 CCMP   PSK  Moria

 BSSID              STATION            PWR   Rate    Lost    Frames  Probe

 EC:4F:82:73:D1:1A  EC:1A:59:E4:FD:41  -23   54e-54e     0    64269
 EC:4F:82:73:D1:1A  EC:1A:59:F1:FB:21  -23   54e-46e     0    61572
 EC:4F:82:73:D1:1A  94:10:3E:2B:7A:55  -28   54e-54e     0    31353
 EC:4F:82:73:D1:1A  B4:75:0E:0D:33:D5  -36   46e-54e     0    32601
 EC:4F:82:73:D1:1A  60:38:E0:EE:7C:E5  -39   54e- 1e     0    48631
 EC:4F:82:73:D1:1A  B8:27:EB:09:1A:81  -38   54e-54e     0   301214
 EC:4F:82:73:D1:1A  14:91:82:C8:6A:09  -41   54e-46e     0    28617
 EC:4F:82:73:D1:1A  A0:18:28:33:34:F8  -45   54e-24    476   205282
 EC:4F:82:73:D1:1A  14:91:82:24:DD:35  -48   54e-54e     0    42845
 EC:4F:82:73:D1:1A  08:66:98:ED:1E:19  -49   54e-54e   238    32549
 EC:4F:82:73:D1:1A  B4:75:0E:0D:94:65  -55   36e- 1e     0    31496
```

Figure 13. Command and results to scan for devices connected to the target AP

Table 4. Wi-Fi MAC OUI search and results

OUI Search	**Results**
EC:1A:59:E4:FD:41	EC:1A:59 Belkin International Inc.
EC:1A:59:F1:FB:21	EC:1A:59 Belkin International Inc.
94:10:3E:2B:7A:55	94:10:3E Belkin International Inc.
B4:75:0E:0D:33:D5	B4:75:0E Belkin International Inc.
60:38:E0:EE:7C:E5	60:38:E0 Belkin International Inc.
B8:27:EB:09:1A:81	B8:27:EB Raspberry Pi Foundation
14:91:82:C8:6A:09	14:91:82 Belkin International Inc.
A0:18:28:33:34:F8	A0:18:28 Apple, Inc.
14:91:82:24:DD:35	14:91:82 Belkin International Inc.
08:66:98:ED:1E:19	08:66:98 Apple, Inc.
B4:75:0E:0D:94:65	B4:75:0E Belkin International Inc.

Similarly, Figure 14 shows the command used to operate the Bluetooth wireless adapter (Plugable USB 2.0) and sniffing tool (BlueZ) to scan for BLE devices. The Bluetooth service is started, the interface is activated, and scanning is initiated. The results show device names and MAC addresses found from `ADV_IND` and `SCAN_RESP` packets collected using the low range Plugable Bluetooth adapter from within the smart home.

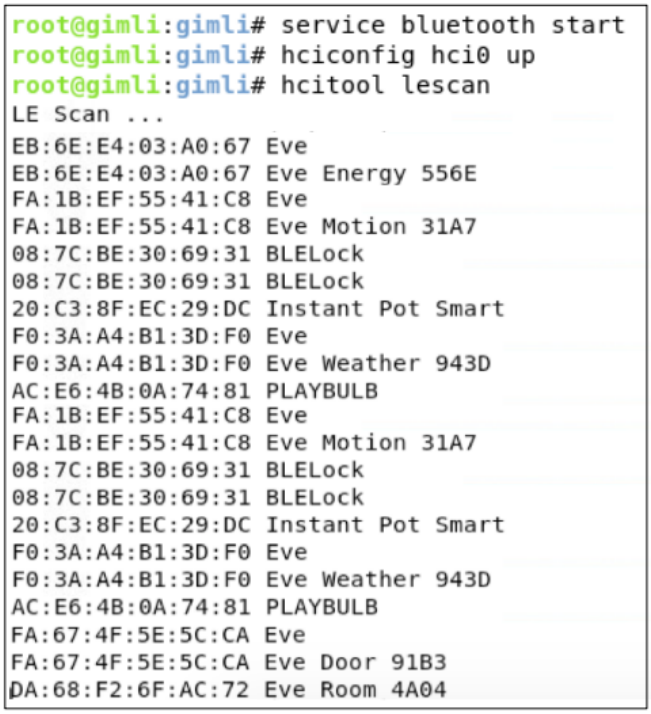


Figure 14. Command and results to scan for BLE devices within the smart home

3.4.3 Passive Sniffing.

Passive sniffing is used to capture Wi-Fi and BLE traffic from the smart home. Sniffing occurs simultaneously for Wi-Fi and BLE traffic using the Alfa Card and three Ubertooth One sniffers respectively. Prior to capturing Wi-Fi traffic, the Alfa Card's wireless interface must be in monitor mode to capture Wi-Fi packets destined to any device (see Figure 11). The Airodump-ng tool is then used to capture Wi-Fi traffic from the smart home. When operating the capture tool, the Alfa Card's interface ("wlan1"), capture output file format (".pcap"), target AP MAC address ("ec4f8273d11a"), and target AP Wi-Fi channel ("1") options are set. Sniffing is initiated using `# airodump-ng -c 1 wlan1 -o pcap -w wifi --bssid ec4f8273d11a`.

To capture BLE traffic, three Ubertooth One sniffers are set to operate with each device ("U0"-"U2") tuned to one of three advertisement channels ("37"-"39"), to fol-

low traffic ("f"), and output packets to a capture file (".pcap"). Appendix A provides a script which uses the Bash Unix shell command language to simplify activating all three of the Ubertooth One sniffers simultaneously. A single Ubertooth One sniffer can be operated using `# ubertooth-btle -f -U0 -A37 -qble.pcap`. When sniffing is complete, each tool is terminated and the resulting capture files are stored on the host machine.

3.4.3.1 Ubertooth One Issues.

Two issues with the Ubertooth One firmware were discovered during testing and experimentation. First, an investigation of a 3-hour capture identified that the Ubertooth's clock would drift over time making it impossible to identify the real time of a packet. This issue was resolved by updating the Ubertooth One firmware to utilize the clock of the host computer rather than the internal Ubertooth One clock for the packet timestamp. This fix was published to the Ubertooth firmware GitHub issues page as issue #251 [43]. Second, the Ubertooth One sometimes froze when a master sent a `CONNECT_REQ` packet. This second issue was discovered during the first day of trials and occurs because newer BLE devices, such as the iPhone 6+, can elect to use a subset of available BLE channels and the Ubertooth One did not support this feature at the time of experimentation. The Ubertooth One sniffers were observed during the rest of experimentation and if the sniffers froze, they were restarted. A bug report was submitted as issue #270 and has since been resolved by the Ubertooth One developers [44]. The second issue was the cause for the lack of BLE data during trial 1 of experimentation.

3.4.4 Preprocessor.

After passive sniffing is complete, the Wi-Fi packet captures are parsed and organized in preparation for the classifier. This is accomplished with a script written in Python using Pyshark, a Python wrapper for parsing packets with Wireshark dissectors (see Appendix B). The command used to operate the preprocessor is `$ python wifi.py -p <input file>` where the input file is a Wi-Fi capture file (.pcap). The time, size, source, and destination for each 802.11 data packet are extracted from the captures and the resulting 4-tuples are stored in comma-separated values (CSV) files. All other 802.11 packet types and packets with a source or destination not in the list of Wi-Fi devices found during reconnaissance do not include information used by the classifier and are not saved. The 4-tuples are stored in two files per device per day: one file in which each 4-tuple has a source address of the device and one file in which each 4-tuple has a destination address of the device. For example, the Belkin Motion Sensor (w_7) will have two files per day, one in which every 4-tuple represents a packet from the sensor to another device in the Wi-Fi device list and one file in which every 4-tuple represents a packet from a device in the list to the Belkin Motion Sensor. These CSV files are used by the Wi-Fi classifier and network mapper components of CITIoT.

3.4.5 MAC Tracker.

The MAC tracker unit tracks when devices are in the smart home based off Wi-Fi packets sent from that device. It is implemented within the preprocessing script (see Appendix B), operates at the same time as data preprocessing, and utilizes every 802.11 packet that has a source MAC address of a device in the Wi-Fi device list. As the packet capture is parsed, the tracker keeps a list of the first and last time a packet was sent from a device within the smart home. If the tracker observes no packets sent

from a device for five minutes it marks the device as no longer in the smart home and records the arrival and departure times in a CSV file. In testing it was observed that inactive Apple devices stop sending packets to save battery power, but still send at least one packet per five minutes even when idle. Therefore, a five minute interval was chosen as the appropriate amount of time to account for idle devices; this ensures that a lack of packets sent from a device occurs because the device is away from the home and not just idle. This time period is easily changed within the script and can be increased to provide more confidence that the device is away from the home or decreased to improve precision.

During testing it was observed that some packets caused the MAC tracker to report erroneous times. Two anomalies in packet times were found: (i) intermittent packets would have a negative epoch timestamp and (ii) periodically, sequentially numbered packets would have a timestamp that should appear much later in the capture. Figures 15 and 16 show example Wireshark traces depicting both of these bugs. The cause of these inconsistencies was not discovered.

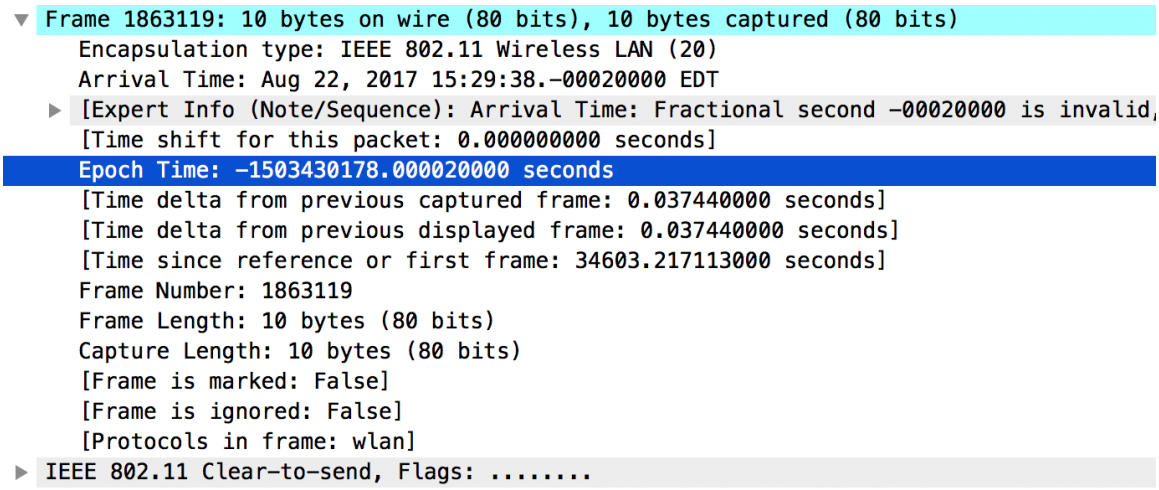
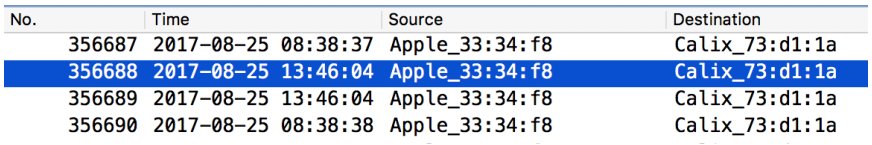


Figure 15. Encrypted packet used in MAC tracker showing corrupted timestamp



No.	Time	Source	Destination
356687	2017-08-25 08:38:37	Apple_33:34:f8	Calix_73:d1:1a
356688	2017-08-25 13:46:04	Apple_33:34:f8	Calix_73:d1:1a
356689	2017-08-25 13:46:04	Apple_33:34:f8	Calix_73:d1:1a
356690	2017-08-25 08:38:38	Apple_33:34:f8	Calix_73:d1:1a

Figure 16. Encrypted packets used in MAC tracker showing sequential frame numbers, but wrong times

These issues were overcome by checking for two conditions and, if either was met, ignoring the packet: first, if a packet's timestamp was less than zero, and second, if the difference between two consecutive packet timestamps was greater than five seconds. These fixes, however, presented a third issue in tracking devices: if there was a jump in time (i.e., from a paused capture), all new packets would meet the second condition and the packets would be ignored. This new issue was resolved by checking if more than five consecutive packets were received with a timestamp difference of greater than 5 seconds; if so, the baseline time would be shifted to the time of the new packets. For example, the tool processes a packet with a timestamp of 13:00:00. The next packet observed has a timestamp of 14:00:00 which is ignored because it meets the second condition: the difference between consecutive packets is greater than five seconds. The next four packets, though, each have a timestamp of 14:00:01. Since five subsequent packets have been received with a timestamp greater than five seconds from the first packet, the baseline time is reset from 13:00:00 to 14:00:01 and future packets will no longer be ignored. The tracker provides the user with the number and time of invalid packets to facilitate further investigation.

3.4.6 Classifier.

This section describes the training and operation of the component of CITIoT used to classify devices and identify events within the smart home. There are three parts to the classifier: (i) the Wi-Fi classifier trainer, (ii) the Wi-Fi classifier, and (iii) the BLE classifier. These components are implemented separately within two scripts

provided in Appendices B and C (one for each protocol). Appendix D includes helper functions used by both of the protocol scripts.

3.4.6.1 Wi-Fi Classifier Trainer.

The Wi-Fi classifier must be trained to the devices within SHAA to be able to classify devices and identify events. This was accomplished using traffic captured during a training trial in which each Wi-Fi and BLE device was activated according to the Classifier Training Event Log (see Appendix E). Events were chosen that represent a real user activating devices throughout a normal day in a smart home. The traffic from the smart home was captured and preprocessed as described in Section 3.4.4. For Wi-Fi traffic, the classifier is trained to classify devices into one of three types (outlet, sensor, or camera) and identify device events (outlet, motion, or camera event). The CSV files created during the preprocessing stage are used by the classifier training script (see Appendix B) to provide the user with two scatter plots per device (one depicting packets sent from a device and one depicting packets sent to a device). The command used to operate the classifier trainer is `$ python wifi.py -t` and it must be called after the preprocessor and from the same directory. Appendix F shows the plots for packets sent from the Raspberry Pi to each Wi-Fi device, while Appendix G provides the plots for packets sent from each device to the router. An example of each is provided in Figures 17 and 18, respectively. The x-axis of these plots represents time in hours, while the y-axis represents packet size in bytes. These plots are used as a graphical representation of the traffic within SHAA to help determine trends and patterns in packet sizes for devices and events.

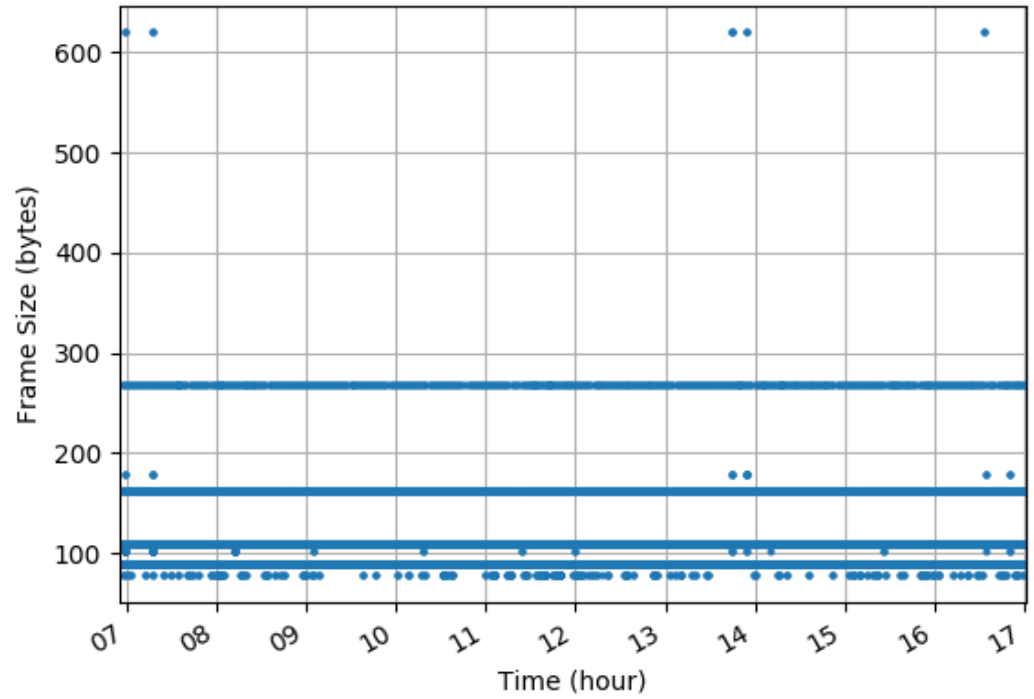


Figure 17. Example plot showing packets sent from Raspberry Pi to Switch1 used to train the classifier

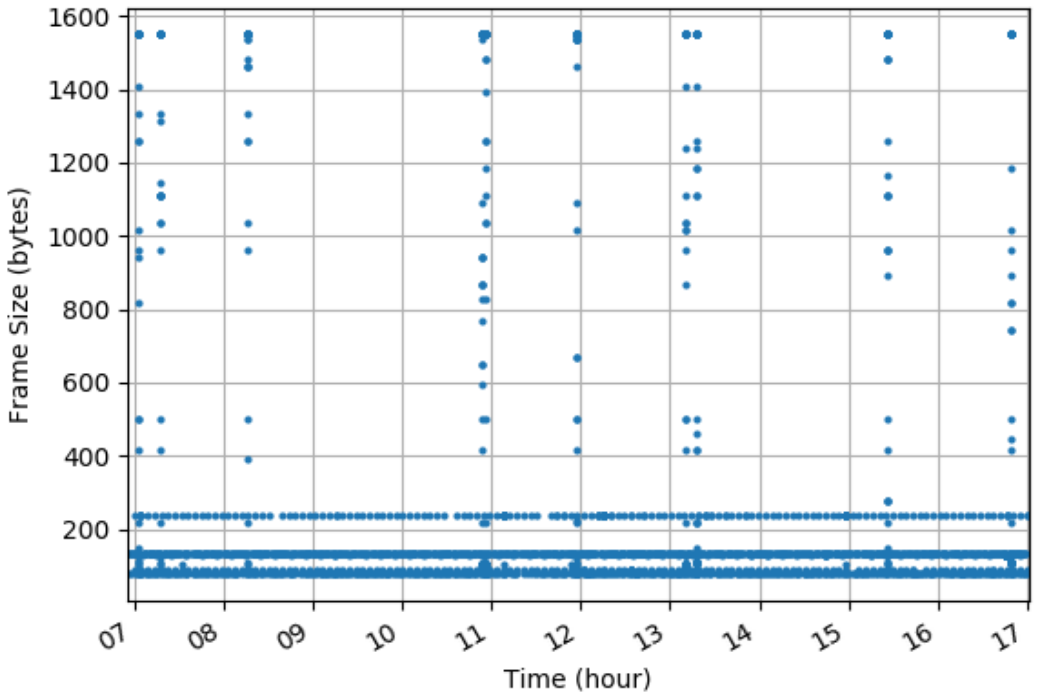


Figure 18. Example plot showing packets sent from Motion to Router used to train the classifier

After observing the plots created in training, it was discovered that the Raspberry Pi communicates with each type of device in a unique way and, therefore, traffic from the Raspberry Pi to the devices can be used to classify devices. Figures 19, 20, and 21 show three plots from Appendix F (packets from the Raspberry Pi to the NetCam, Motion, and Switch1) zoomed in on the unique packet traffic that helped generate the classification criteria. Figure 22 lists the criteria established to classify devices using the plots. This criteria is described as follows: if the device receives a packet from the Raspberry Pi between 619 and 632 bytes, then it is an outlet; if the device is not an outlet and receives 269-byte packet from the Raspberry Pi, then it is a sensor device; and if the device is not an outlet and receives 281-byte packet from the Pi, then it is a camera device.

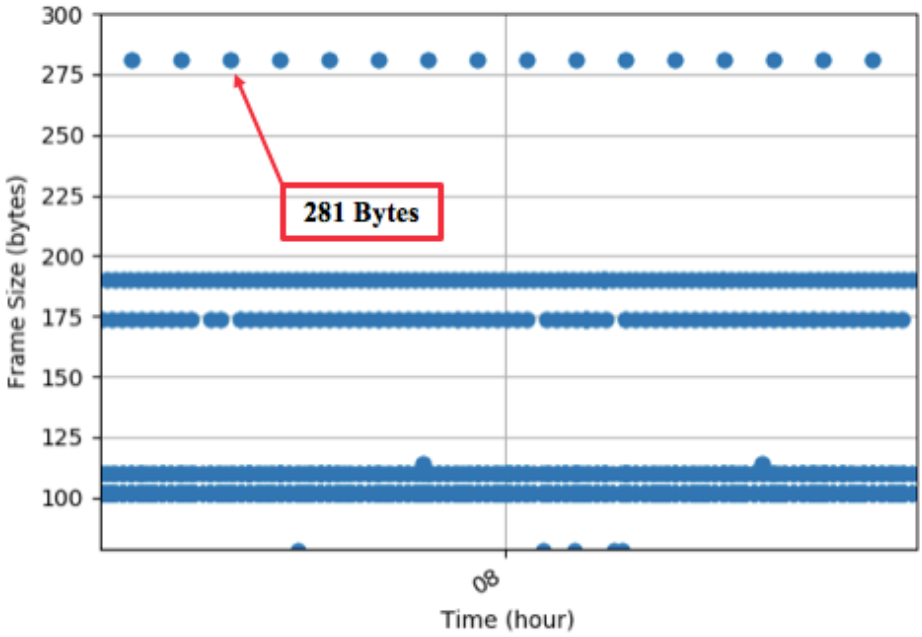


Figure 19. Packets sent from Pi to NetCam used to classify camera devices

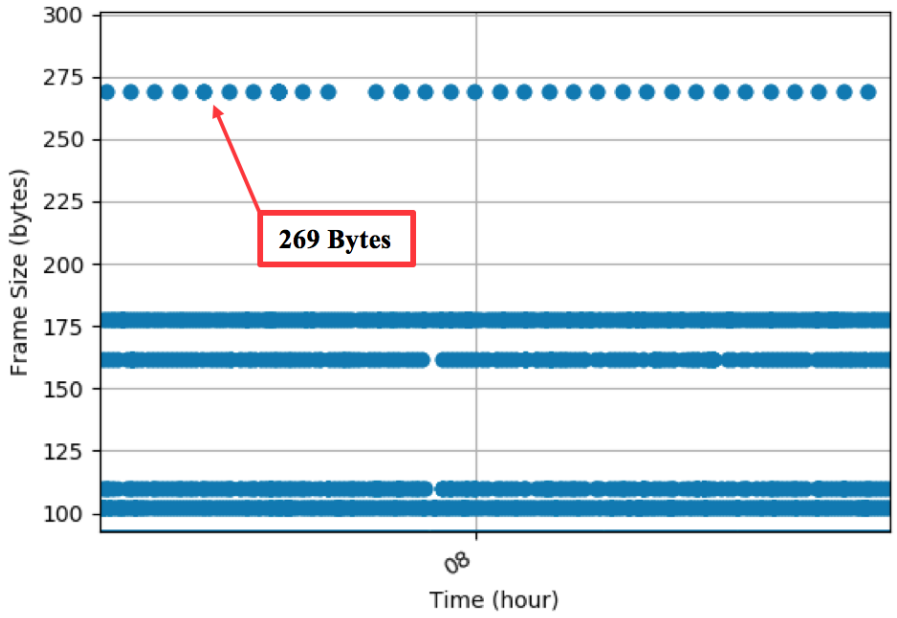


Figure 20. Packets sent from Pi to Motion used to classify motion devices

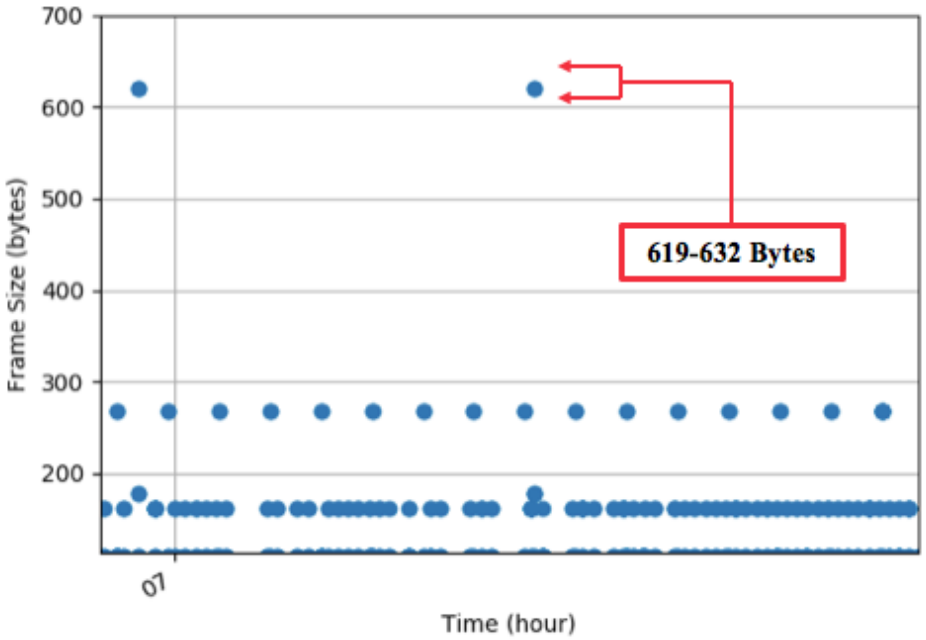


Figure 21. Packets sent from Pi to Switch1 used to classify outlet devices

Device Type	Device Classification Criteria
Outlet	$619\ bytes \leq FSize_{Incoming} \leq 632\ bytes$
Sensor	$if\ device\ != outlet \quad and \quad FSize_{Incoming} = 269\ bytes$
Camera	$if\ device\ != outlet \quad and \quad FSize_{Incoming} = 281\ bytes$

Figure 22. Criteria used to classify devices

Similarly, comparing event logs to the plots in Appendix F from training revealed that the Raspberry Pi sends a packet with a frame size of 619, 620, or 632 bytes to an outlet every time an event occurs. Figure 23 shows a plot depicting packets sent from the Raspberry Pi to the Mini outlet during training zoomed in on two events that helped generate the event identification criteria for outlets. At times, multiple packets meeting the event criteria are sent within a minute due to retransmission. Therefore, CITIoT only identifies one outlet event per device per minute. This event interval provides enough precision for the data leakage analysis. Additionally, by correlating event logs with the plots in Appendix G, it was observed that the sensors and cameras send a burst of packets to the router every time an event occurs. Figures 24 and 26 show two plots from training depicting packets sent from the NetCam and Motion to the router zoomed around the unique packet traffic that helped generate the event identification criteria for these devices. These bursts can be recognized among network traffic by adding the FSize of all packets sent by a device within a minute. During an event, cameras send a burst of packets that amount to a FSize greater than 100,000 bytes, while a sensor event burst is greater than 10,000 bytes. Figures 25 and 27 provide plots depicting the result after adding the FSize of all packets sent in a minute by the NetCam and Motion during training. The plots are zoomed in around events to show the unique traffic used to identify camera and sensor

events. The bursts sent by the sensor or camera may span a two-minute period to avoid CITIoT flagging two events (one per minute), the tool only identifies one sensor or camera event per device per two minutes. Figure 28 lists the criteria established to identify events. This criteria is described as follows: if the device is a camera and the total FSize of packets sent to the router in a minute is greater than 100,000 bytes, then a camera event occurred; if the device is a sensor and the total FSize of packets sent to the router in a minute is greater than 10,000 bytes, then a sensor event occurred; and if the FSize of a packet sent from the Raspberry Pi to an outlet device is between 619 and 632 bytes, then an outlet event occurred.

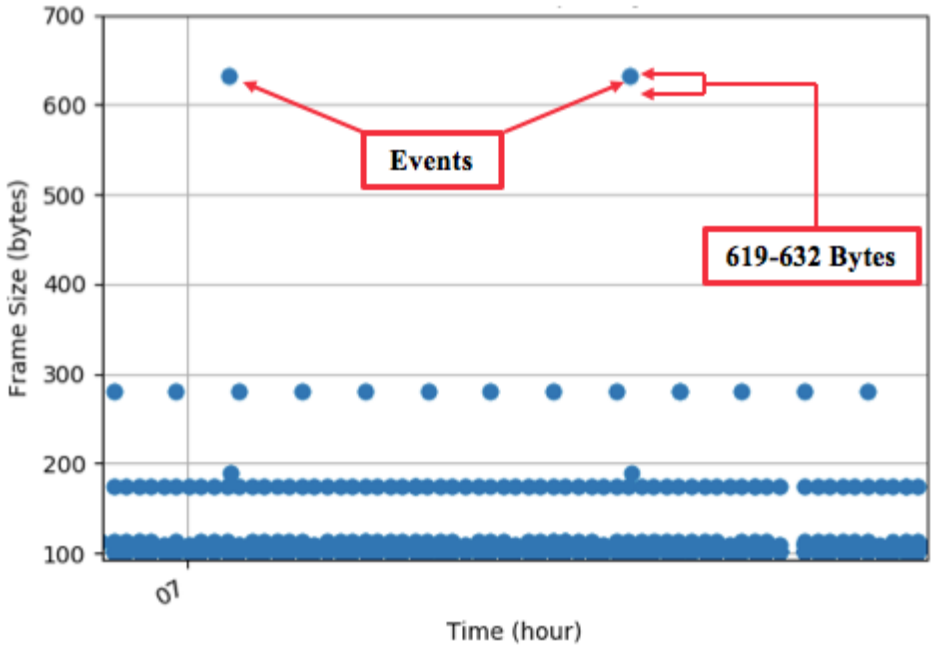


Figure 23. Packets sent from Pi to Mini during an outlet event

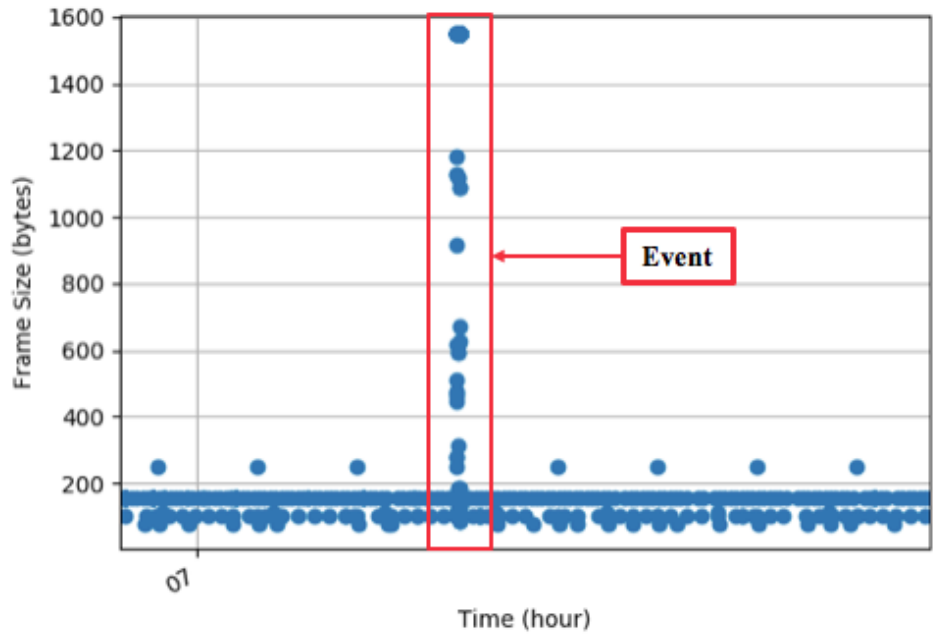


Figure 24. Packets sent from NetCam to router during a camera event

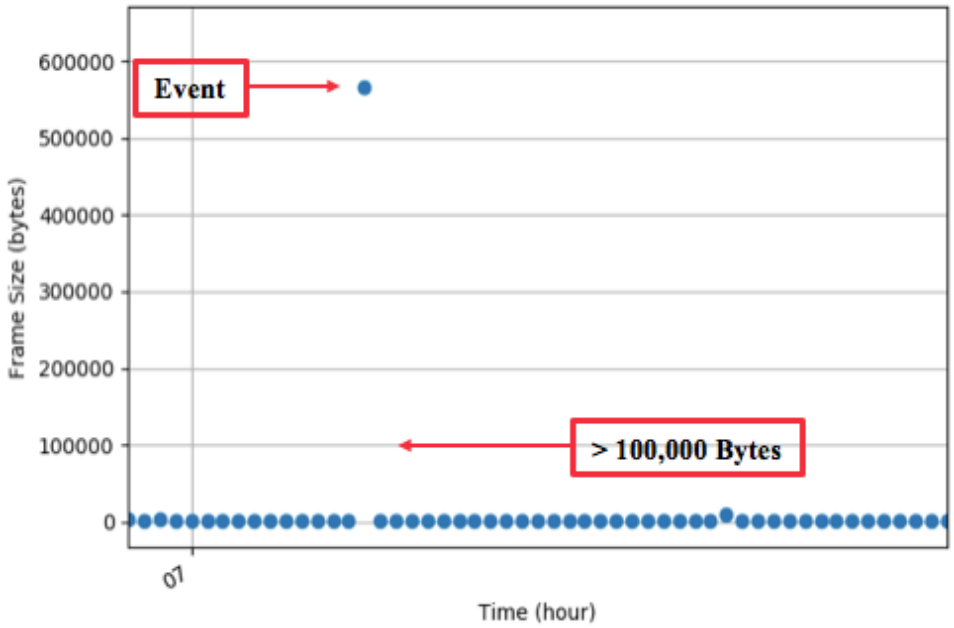


Figure 25. Packets sent from NetCam to router with one minute cumulative FSize during a camera event

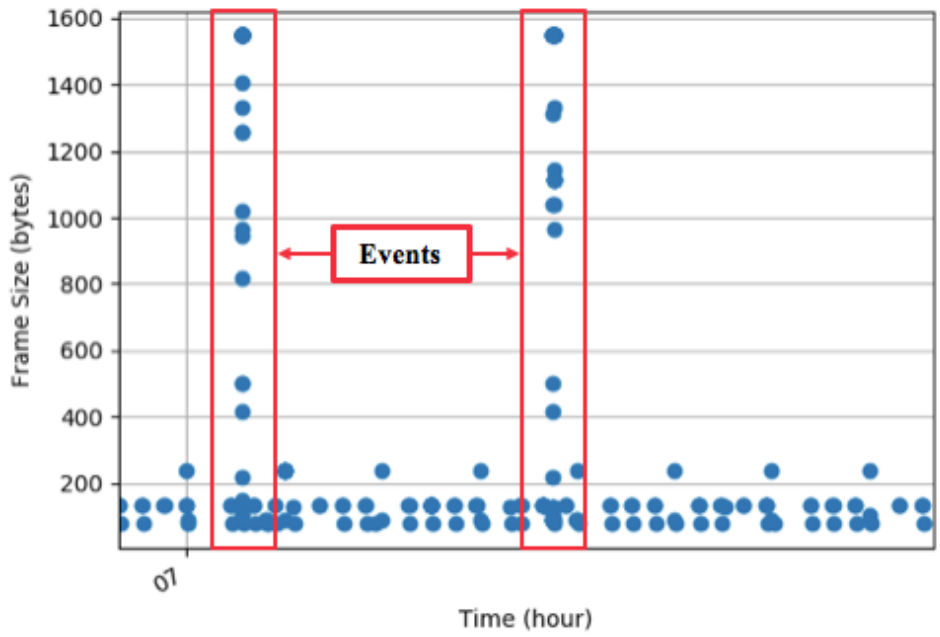


Figure 26. Packets sent from Motion to router during a motion event

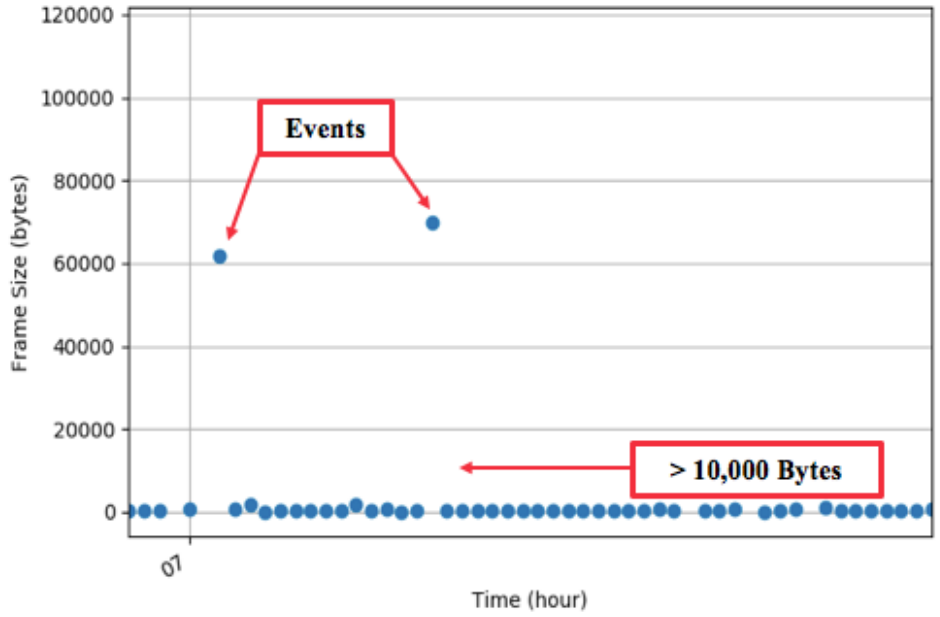


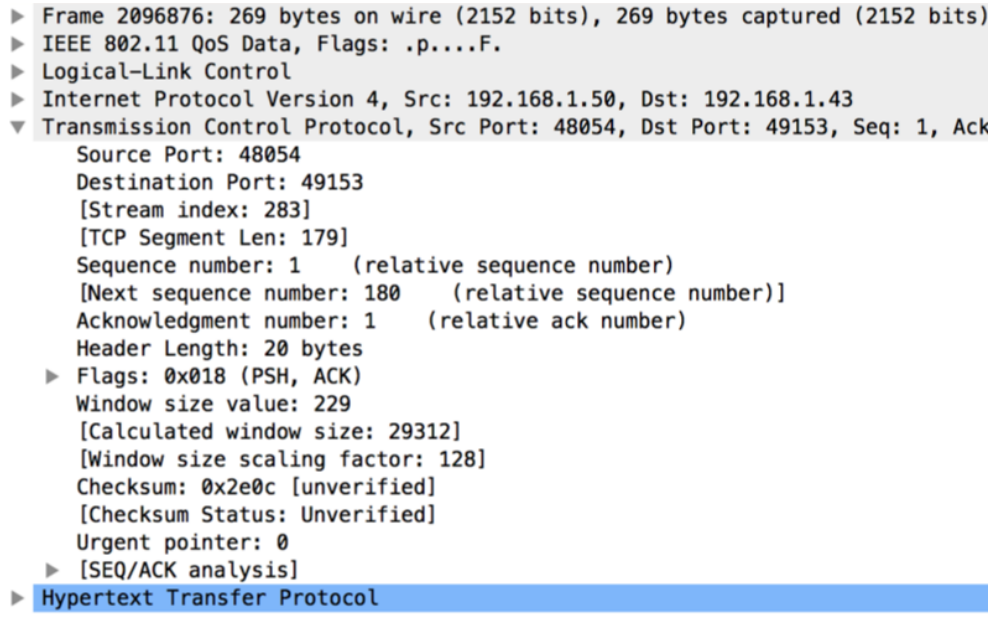
Figure 27. Packets sent from Motion to router with one minute cumulative FSize during a motion event

	Event Identification Criteria
Outlet Event	$if\ device = outlet \quad and \quad 619\ bytes \leq FSize_{Incoming} \leq 632\ bytes$
Sensor Event	$if\ device = sensor \quad and \quad \sum_{t}^{t+60s} FSize_{Outgoing} > 10{,}000\ bytes$
Camera Event	$if\ device = camera \quad and \quad \sum_{t}^{t+60s} FSize_{Outgoing} > 100{,}000\ bytes$

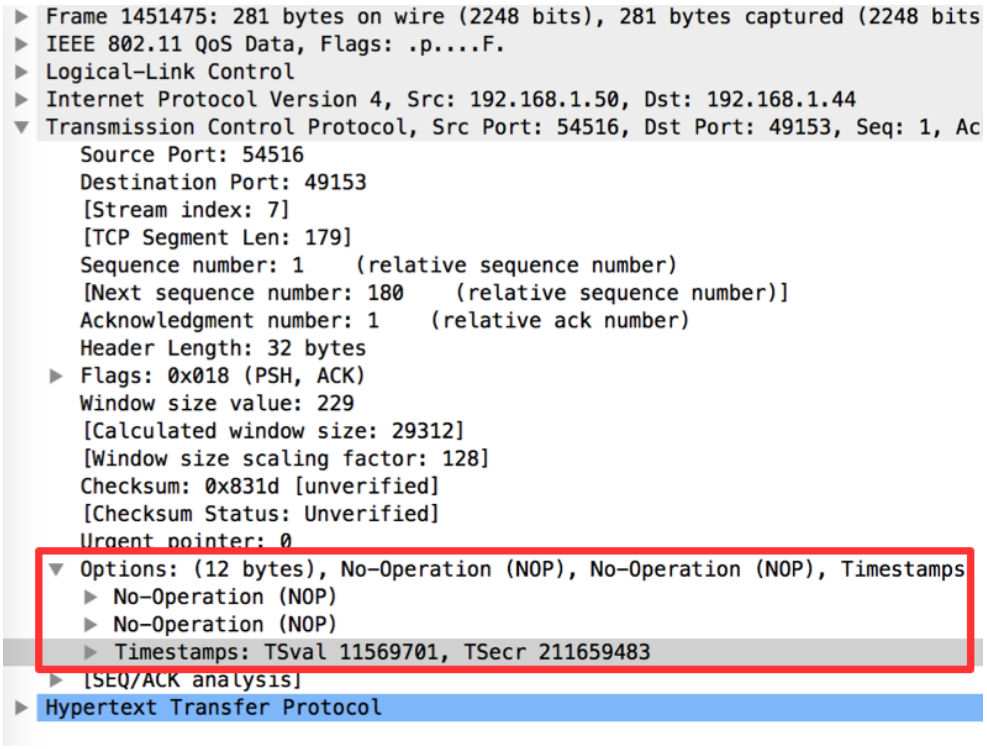
Figure 28. Criteria used to identify events

Further investigation of each characteristic Wi-Fi packet exchange was accomplished to determine why patterns exist in these packets. The Wi-Fi captures were decrypted by adding the AP password into Wireshark and selecting the decrypt 802.11 packets option. It was discovered that the Raspberry Pi sends a 269 or 281 byte Hypertext Transfer Protocol (HTTP) `SUBSCRIBE` packet to subscribe to events from a given device. The NetCam device is the only one to receive a 281 byte `SUBSCRIBE` packet, and Figure 29 shows that a timestamp Transmission Control Protocol (TCP) option accounts for the packet length difference. The Raspberry Pi sends a 619, 620, or 632 byte HTTP `POST` request to activate an outlet. Figure 30 shows an example packet for each of the different sized `POST` packets. These packets revealed that Switch4 received a 619 byte `POST` packet because the Internet Protocol (IP) address had one character less in the final octet resulting in one less byte in the HTTP host address. The `POST` packet sent to the Mini outlet was 632 bytes with the additional 12 bytes coming from TCP options that were not included in other devices. Every time the camera observes motion after one minute of idleness it sends the user a camera snapshot via email. Every time a sensor observes motion after one minute of idleness it sends a notification to the user. To do this, the device creates a secure connection to Belkin's cloud service hosted by Amazon EC2 Cloud and then sends the notification to the Belkin application on the iPhone 6+. Decrypting this traffic

is beyond the scope of this thesis.

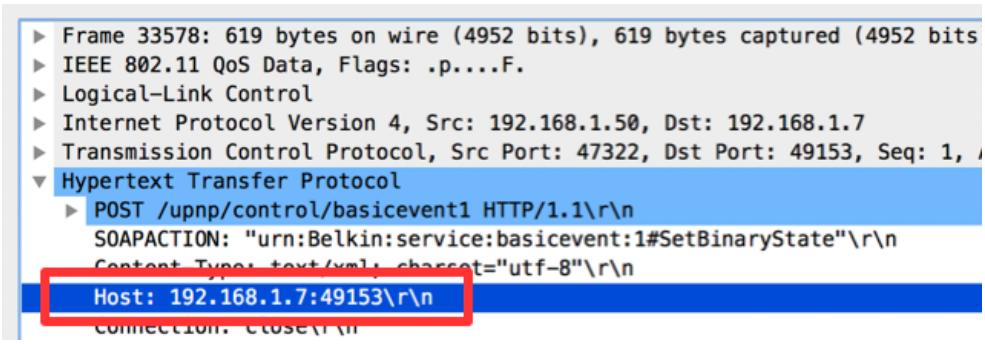


(a) Motion device

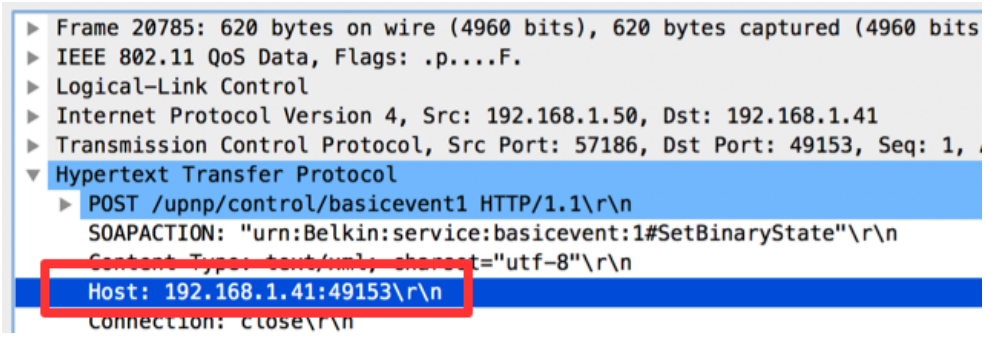


(b) NetCam device

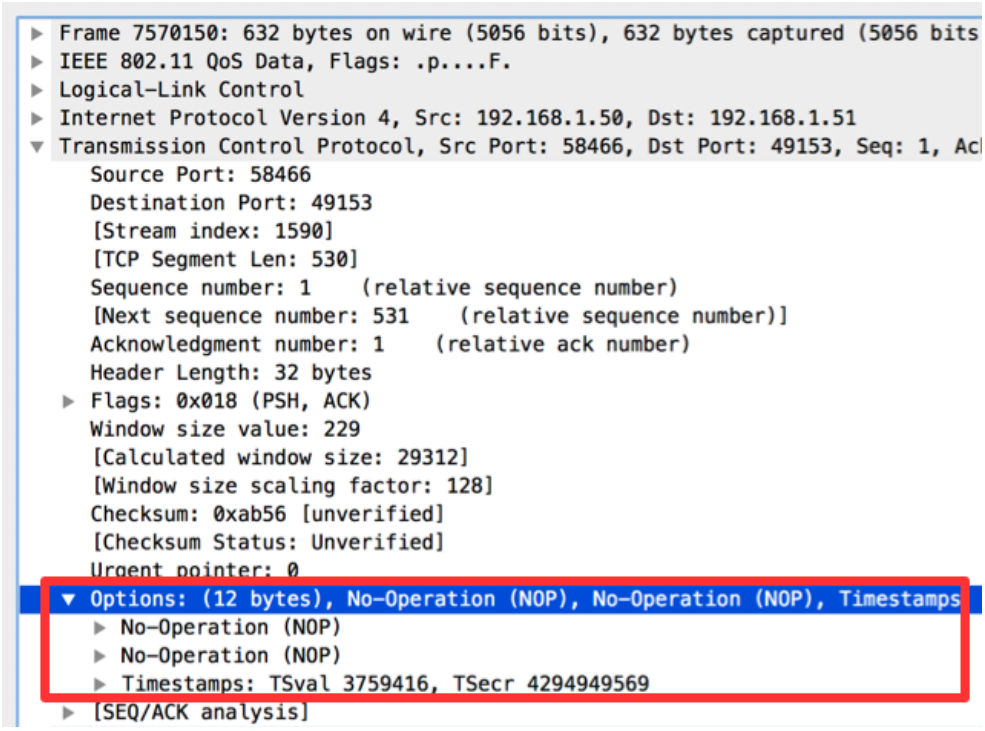
Figure 29. Decrypted `SUBSCRIBE` packets from Raspberry Pi to the Motion and NetCam devices depicting difference in FSize



(a) Switch4 device



(b) Switch2 device



(c) Mini device

Figure 30. Decrypted `POST` packets from Raspberry Pi to the Switch4, Switch2, and Mini depicting differences in FSize

3.4.6.2 Wi-Fi Classifier.

For Wi-Fi traffic, the classifier is operated via a Python script (see Appendix B) which utilizes each of the CSV files created during data preprocessing and the criteria found during training to classify devices and identify events. The command to operate the classifier is `$ python wifi.py -c` and it must be called after the preprocessor and from the same directory. First, for each device, the classifier analyzes traffic sent from the Raspberry Pi to that device to classify what kind of device it is (outlet, sensor, or camera). If the traffic meets one of the criteria found during training, then the device type is set accordingly, otherwise the type is unknown. Second, based on the device type determined in the first step, traffic is tested against the event criteria found during training to identify when an event occurs. For an outlet device, the traffic sent from the Raspberry Pi to the device is used to identify outlet events, while traffic sent from the device to the router is used to identify sensor and camera events. The type found for each device is stored in a CSV file listing the MAC address and type of device. The time, source, and destination for each event identified are also stored in a separate CSV file.

3.4.6.3 BLE Classifier.

For BLE traffic, a script (see Appendix C) is used to parse the packet captures created during passive sniffing for `ADV_IND`, `SCAN_RESP`, and `CONNECT_REQ` packets. The `ADV_IND` and `SCAN_RESP` packets provide advertisement information, such as device name, used by the classifier to classify devices. Instead of classifying devices into categories as in Wi-Fi, the classifier relies on the name provided within these packets. The discovered device name is stored along with the device BLE MAC address in a CSV file. Devices only exchange information after a `CONNECT_REQ` packet, therefore, these packets are selected as the indicator for device events. The classifier searches for

CONNECT_REQ packets, uses the device name found during the first step, and stores the time, source, and destination for each event into a CSV file. Only device events that match BLE devices found in reconnaissance are stored. Similar to Wi-Fi sensors and cameras, a single BLE event may cause devices to send multiple CONNECT_REQ packets that span a two-minute time period. This would cause CITIoT to mark one event as multiple events. Therefore, CITIoT identifies one event per device per two-minutes (i.e., the CONNECT_REQ packets sent within two-minutes of the first connection packet are ignored). This may cause the device to miss two events that occur less than a minute apart, but this event interval per device provides enough precision for the data leakage investigation.

3.4.7 Network Mapper.

The next component of CITIoT, the network mapper, creates a graphical representation of how Wi-Fi devices are connected within the smart home using the FSize of 802.11 data packets sent between devices. The network mapper unit is implemented within the Wi-Fi classifier script (see Appendix B) and operates as the classifier parses each CSV file created by the data preprocessing tool. The FSize of all packets sent between two devices are combined and stored in a new CSV file along with the device names. In the network map, the devices are the nodes, while the data sent between devices are the edges. For example, if the Raspberry Pi sends a total of three 500 byte packets to Switch1 then a three-tuple is written to a CSV that includes the Raspberry Pi and Switch1 as nodes and the cumulative number of bytes sent, 1500 bytes, as the edge between these two nodes. The R script provided in Appendix H uses the "igraph" R package to turn the edge file created by CITIoT into an undirected network map [45]. Figure 31 provides a sample network map created by the script—nodes represent devices and edges depict the cumulative frame size

sent between connected devices (i.e., thicker lines represent more data sent between devices).

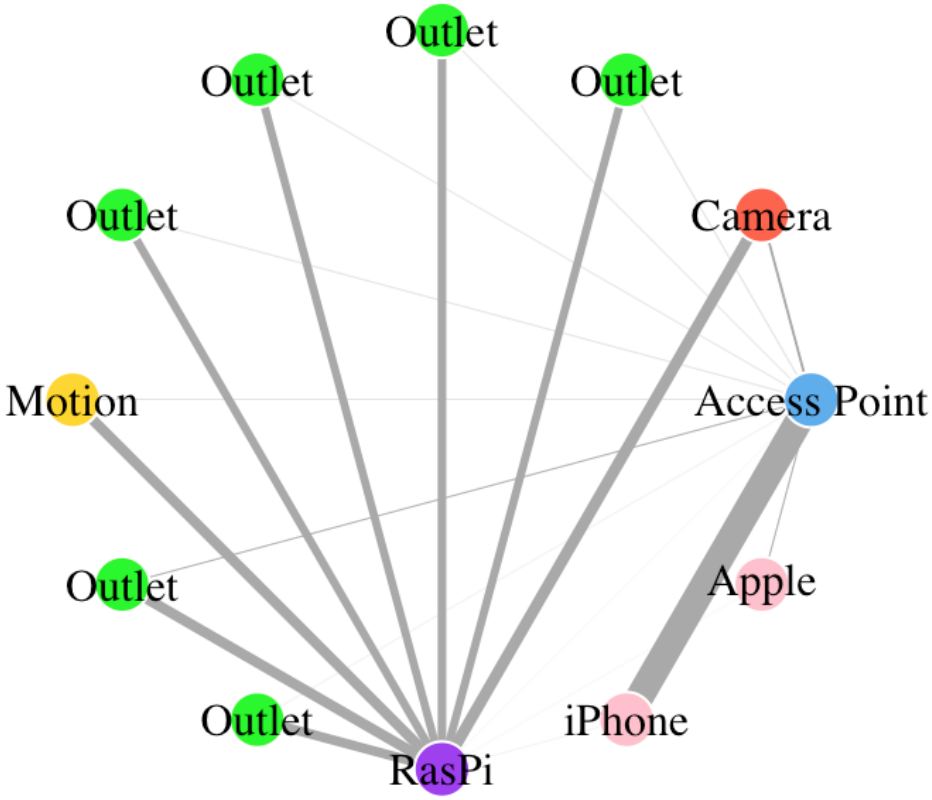


Figure 31. Network mapping of smart home architecture

3.4.8 Security.

Security of CITIoT is not key to this work, but still worth discussion. The tool's interaction with the smart home and any other Wi-Fi or BLE devices is completely passive. The sniffers do not associate with the smart home's access point nor do they transmit any data that is detectable. Smart home operation is completely unaffected by the tools presented. Therefore, the eavesdropper is virtually undetectable and the tools provided within CITIoT have little to no attack surface.

3.5 Mitigation of IoT Leakage (MIoTL)

The MIoTL tool provides methods for mitigating data leakage from IoT devices in smart homes. As shown in Figure 32, MIoTL has two components which operate within SHAA to negate capabilities provided by CITIoT using the concept of chaff presented in Section 2.5 to send fake IoT traffic among the real traffic. The first component, device shadow, spoofs IoT device traffic to make it more difficult to classify devices and identify device events. The second component, MAC shadow, spoofs traffic coming from a user's device (e.g., iPhone) to make it difficult to track when the user is home or away. The tool operates on a Raspberry Pi 3B with the Kali 4.1.19 operating system. The Raspberry Pi was used because the tool is always running and the Pi provides low power consumption and constant connectivity on the network.

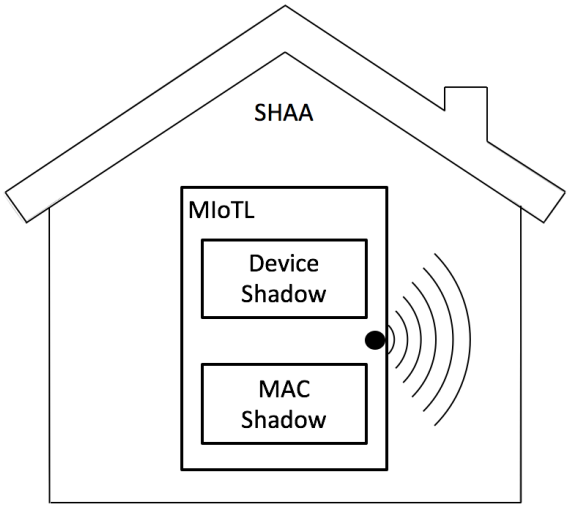


Figure 32. Diagram of MIoTL tool components

3.5.1 Device Shadow.

The device shadow script is provided in Appendix I and was written with Python using the Scapy network tool to randomly spoof packets sent between devices in the

smart home. Three different packet groups are randomly sent: (i) a packet of 620 bytes sent from the Raspberry Pi to every other device in a random order; (ii) a series of packets totaling 10,000 bytes sent from each device in a random order to the router; and (iii) a series of packets totaling 100,000 bytes sent from each device in a random order to the router. MIoTL randomly sends one of these packet groups at a random interval between ten and fifteen minutes. The random order and interval are used to make it difficult to differentiate real traffic from spoofed traffic sent from the tool. Packets are sent to and from each device to make it hard to identify devices. For example, per the device classification criteria used by CITIoT, a packet of size 620 bytes sent from the Raspberry Pi to a device indicates that the device is an outlet. This component spoofs 620-byte packets sent from the Raspberry Pi to each device, regardless of type, so the classifier tool would incorrectly classify a sensor as an outlet. The same concept is used to spoof events.

3.5.2 MAC Shadow.

The MAC shadow script is provided in Appendix J and was written with Python using Scapy to spoof packets sent from a device to a controller. To accomplish this, the script first checks to see if the device is on the smart home network every four minutes. If the device is not on the smart home network, then, at a random interval between 0 and 1 seconds, the script sends ten spoofed packets on behalf of the device (e.g., iPhone 6+) to a controller device (e.g., Apple TV). The tool checks to see if the device is not on the network to ensure the device's entry in the router's Address Resolution Protocol (ARP) table is not changed while the device is on the network and the user does not experience network degradation. The four minute interval was chosen as it sends packets often enough for it to appear that the device is on the network without sending too many packets to impact the performance of the wireless

network.

The process of checking if the device is in the smart home proved to be difficult as the Apple iPhone's network card goes into a low power mode when the phone is not in use and does not respond to TCP ping messages. To check if the device is on the network the script sends a series of 10 ARP requests and if the device responds to one of the requests then it is present. Documentation on how the iPhone's network card operates was not readily available so trial and error found that the ARP method was the most consistent to detect Apple devices.

3.6 Design Summary

This chapter describes each component of the SHAA, CITIoT, and MIoTL tools. The design presented is a unique approach to creating a smart home testbed that can be used to analyze IoT data leakage and test mitigation methods.

IV. Methodology

4.1 Problem/Objective

This research aims to demonstrate how data leakage in smart home environments enables an eavesdropper to classify IoT devices, track user movements, map networks, and identify events within the smart home. It also seeks to show how a smart home user can defend against these attacks. These goals are enabled through the implementation of the CITIoT and MIoTL tools respectively. The experiment presented in this section functions as an evaluation of these tools in a realistic smart home environment, testing how accurately CITIoT operates against the SHAA, and how well MIoTL mitigates these attacks. Specifically, the experimentation attempts to accomplish four objectives:

1. Determine the ability of an observer to accurately classify smart home devices.
2. Examine the percentage of events successfully identified.
3. Measure the capability to track when users are in the smart home.
4. Evaluate processing time and storage requirements.

The evaluation results provide consumers with an understanding of data leakage in smart home environments and a method to defend against these vulnerabilities.

4.2 System Under Test

Figure 33 displays the system under test (SUT) and component under test (CUT) diagram. Response variables, or metrics, described in Section 4.3, consist of classified devices, identified events, user tracking, processing time, and storage requirements. The actual Wi-Fi and BLE traffic collected by CITIoT is considered uncontrolled and

is examined in Section 4.5. Section 4.6 discusses the parameters that do not change throughout the experiment such as computing parameters and the number of devices. Section 4.7.3 describes the experiment treatments which include a user performing actions from a script to interact with the smart home environment and the operation of the MIoTL tool. The components tested include the preprocessor, MAC tracker, classifier, and network mapper.

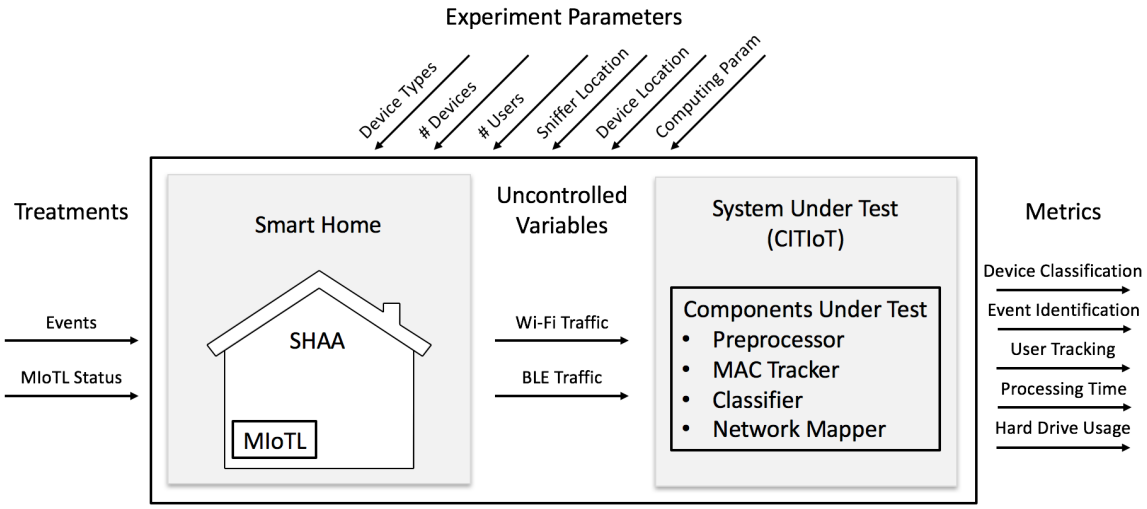


Figure 33. System Under Test and Component Under Test diagram

4.2.1 Assumptions.

The following assumptions are made when designing and executing experiments for CITIoT:

1. The actions performed within SHAA are representative of a real smart home environment.

2. The eavesdropper has already accomplished reconnaissance and scanning and has the required parameters to run CITIoT: AP MAC address, AP channel, Wi-Fi device MAC addresses, BLE device names, and controller MAC addresses.

3. During experimentation, CITIoT is positioned within SHAA, whereas in real-world operation it would be outside of the smart home. It is assumed that a directional antenna aimed at the smart home would have similar results to the antennae used within the smart home during this experimentation. This assumption is substantiated through Rose, et al.'s work cracking a gun safe from a quarter mile away [22].

4. The degree of precision for an identified event is one minute. This level of precision provides enough accuracy for the data leakage investigation presented in this thesis and allows for signal propagation and sniffer delays to be ignored.

4.3 Response Variables

The objectives of this experiment influence the response variables chosen to measure the accuracy and performance of CITIoT. While not directly measured, the effectiveness of MIoTL is quantified via the observed decrease in CITIoT's effectiveness when the mitigation tool is operating. Therefore, response variables (or performance metrics) tied to the four objectives help consider the overall operation of CITIoT. Response variables are observed in four configurations per trial: (i) BLE, (ii) Wi-Fi with no mitigation, (iii) combined BLE and Wi-Fi with no mitigation, and (iv) Wi-Fi with mitigation. Configuration three is used when analyzing CITIoT's overall accuracy and performance without mitigation; the mean of each metric is calculated across all trials. Configuration four is used to measure CITIoT's accuracy while MIoTL is operating; again, the average of each metric across all trials is used.

- **Objective 1:** Determine the ability of an observer to accurately classify smart home devices.

  Device Classification Success (DCS): The DCS response variable quan-

tifies the number of devices successfully identified by CITIoT. A device is considered successfully identified if the name of the BLE device matches the actual name or if the category of the Wi-Fi device matches the actual category.

- **Objective 2:** Examine the percentage of events successfully identified.

  Event Identification True Positives (EITP): The EITP response variable quantifies the number of true positives, or events identified that actually occurred. An event is considered successfully identified if the time and device of the event recognized by CITIoT matches an event in the log. The degree of precision for an identified event is one minute.

  Event Identification False Positives (EIFP): The EIFP response variable quantifies the number of false positives, or identified events that did not occur. An event is considered a false positive if it is identified by CITIoT and no corresponding event exists in the log.

  Event Identification False Negatives (EIFN): The EIFN response variable quantifies the number of false negatives, or events that CITIoT failed to identify. An event is considered a false negative if an event exists in the log and was not identified by CITIoT.

- **Objective 3:** Measure the capability to track when users are in the smart home.

  User Tracking Success (UTS): The UTS response variable quantifies the amount of time in which CITIoT successfully tracks when a user is home or away via Wi-Fi traffic. MIoTL attempts to make the user appear home at all times, so while mitigation is active, UTS quantifies the amount of time CITIoT is able to identify that a user is away from the home despite mitigation.

- **Objective 4:** Evaluate processing time and storage requirements.

Processing Time (PT): The PT response variable, measured in seconds, quantifies how long it takes CITIoT to operate for each trial.

Hard Drive Space (HDS): The HDS response variable, measured in bytes, quantifies the amount of hard drive space used by CITIoT after operation.

4.4 Control Variables

A primary goal of this experiment is to observe how CITIoT operates in a realistic smart home environment. Using COTS components restricts the number of factors that can be altered during experimentation. Event type and timing are the primary factors and are the main treatments in the experiment. A number of scripted events are performed in a random order and time interval throughout a trial. Additionally, the operating status of MIoTL is used to evaluate CITIoT's operation during mitigation.

4.5 Uncontrolled Variables

Another consequence of testing CITIoT against a realistic smart home environment is the introduction of uncontrollable factors. The use of COTS components and an open environment introduces wireless noise and the occurrence of unscripted events. This is beneficial to the evaluation of CITIoT as it should operate regardless of interference.

4.6 Experiment Parameters

Throughout the course of experimentation, several factors are held constant to limit the scope of the experiment:

- **Type of Devices:** The types of devices in the smart home do not change

throughout the experiment.

- **Number of Devices:** The number of devices in the smart home does not change throughout the experiment.
- **Number of Users:** The number of smart home occupants does not change throughout the experiment.
- **Location of Sniffers:** The location of the sniffers relative to the smart home devices does not change throughout the experiment.
- **Location of Devices:** The location of the devices relative to the sniffers does not change throughout the experiment.
- **Computing Parameters:** The operating systems, resources (memory, CPU, and disk space), script languages, and hardware are held constant.

4.7 Experimental Design

The purpose of this experiment is to meet the four objectives listed above. The experiment scenario is defined by a user performing actions from a script to interact with the smart home environment. These events occur while the user is both in and away from the smart home environment. Data logging provides truth data used to evaluate CITIoT's operation.

4.7.1 SHAA.

Figure 34 depicts how the devices, CITIoT tool, and MIoTL tool are laid out within SHAA. To provide consistency between trials, all devices, excluding the iPhone, are not moved throughout experimentation. Proximity between the devices and CITIoT tool provides greater chances of packet capture during experimentation.

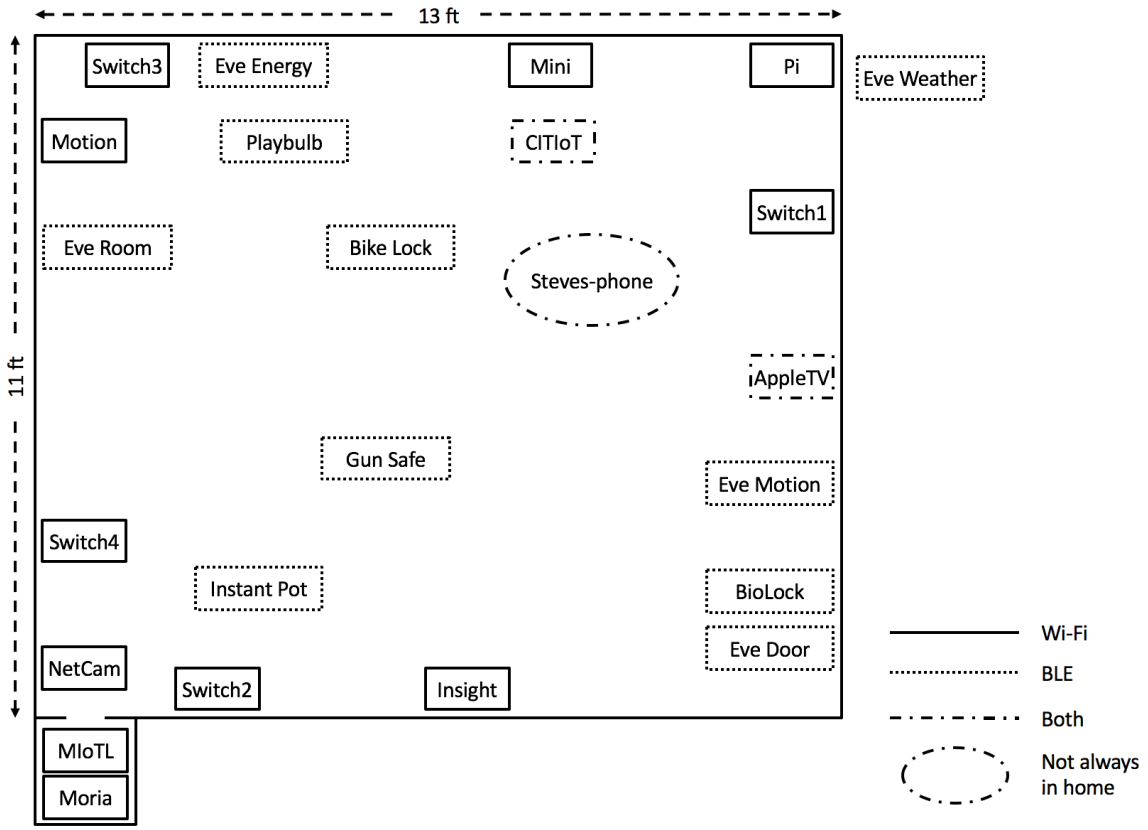


Figure 34. Approximate layout of devices within SHAA for experimentation (not to scale)

4.7.2 CITIoT.

Figure 34 shows where CITIoT is placed within SHAA. Each sniffer operates in the 2.4 GHz band and must be horizontally isolated to avoid interference. The distance between antennae, d, to provide horizontal isolation can be expressed by the equation

$$d \geq 2\frac{D^2}{\lambda} \tag{1}$$

where D is the length of the antenna in meters and λ is the wavelength of the device frequency band in Hz [46].

The Ubertooth One antennae are 3.5 inches long and operate with an average wavelength of 2441 MHz, while the Alfa Card antenna is 6.5 inches long and operates

with an average wavelength of 2412 MHz. Plugging these values into (1) provides a separation value of about 5 inches for the Ubertooth One antennae and 17 inches for the Alfa Card antenna to achieve isolation. Figure 35 shows how the individual sniffers are setup to avoid horizontal interference. The Ubertooth One sniffers are separated by 11 inches, while the Alfa Card is 20 to 23 inches from each of the Ubertooth One sniffers.

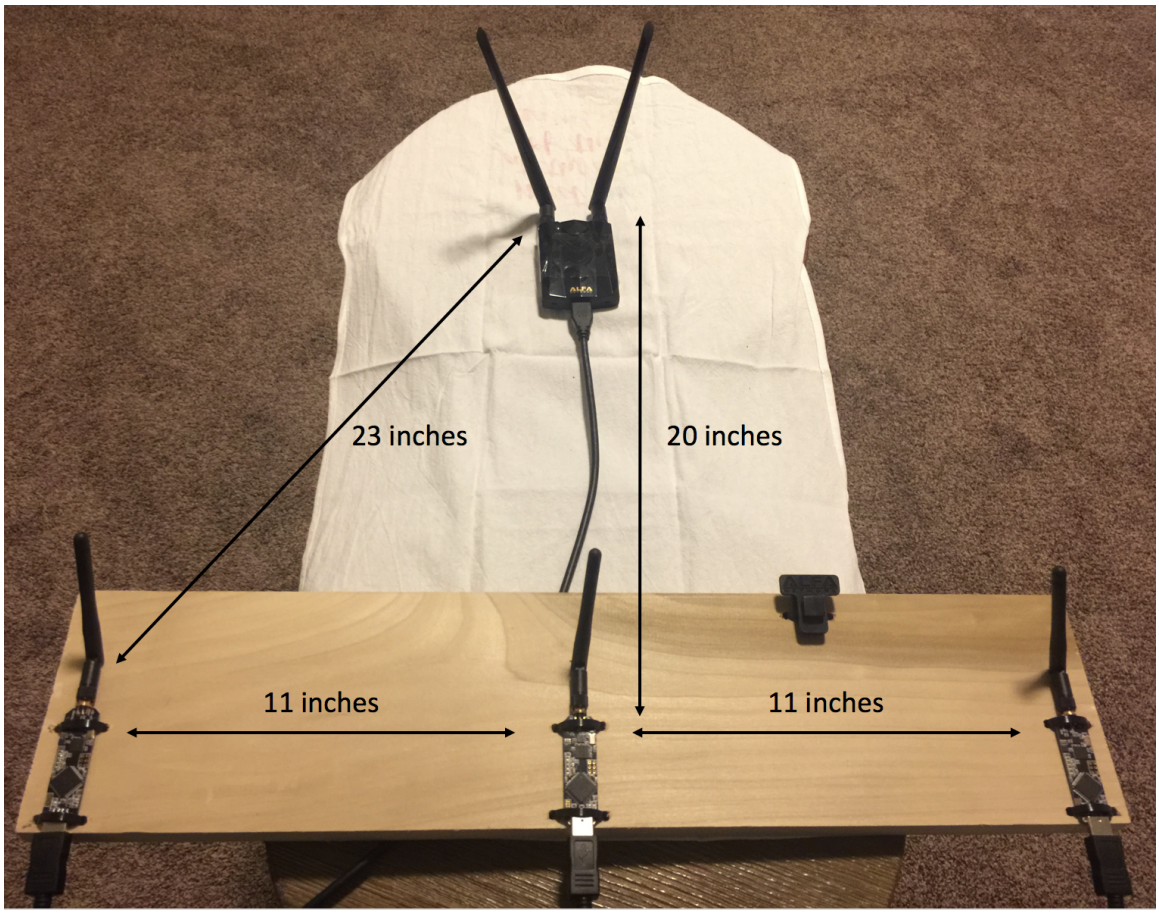


Figure 35. Layout of sniffer antennae for experimentation

4.7.3 Treatments.

Table 5 lists the thirty-one events used during experimentation. These events occur randomly with the restriction that a device must be turned on before it can be

turned off. Events happen multiple times during a trial at random intervals. Each event allows CITIoT to be evaluated against different devices and actions.

Table 5. Experiment events

	Device Name	**Action**	**Protocol**
1	Bike Lock	Unlock	BLE
2	BioLock	Unlock	BLE
3	Instant Pot	Turn on	BLE
4	Instant Pot	Turn off	BLE
5	Gunsafe	Open	BLE
6	Gunsafe	Close	BLE
7	Eve Room	Get temperature in living room	BLE
8	Eve Weather	Get temperature on patio	BLE
9	Eve Door	Open	BLE
10	Eve Door	Close	BLE
11	Eve Energy	Turn on	BLE
12	Eve Energy	Turn off	BLE
13	Eve Motion	Activate motion sensor	BLE
14	Playbulb	Turn on	BLE
15	Playbulb	Turn off	BLE
16	Switch1	Turn on	Wi-Fi
17	Switch1	Turn off	Wi-Fi
18	Switch2	Turn on	Wi-Fi
19	Switch2	Turn off	Wi-Fi
20	Switch3	Turn on	Wi-Fi
21	Switch3	Turn off	Wi-Fi
22	Switch4	Turn on	Wi-Fi
23	Switch4	Turn off	Wi-Fi
24	Mini	Turn on	Wi-Fi
25	Mini	Turn off	Wi-Fi
26	Insight	Turn on	Wi-Fi
27	Insight	Turn off	Wi-Fi
28	NetCam	Activate motion	Wi-Fi
29	Motion	Activate motion sensor	Wi-Fi
30	iPhone	Leave house	Wi-Fi and BLE
31	iPhone	Arrive house	Wi-Fi and BLE

The events are administered with SHAA operating in two states: with and without the MIoTL tool activated. Table 6 describes how the treatments are administered

among trials.

Table 6. Experiment treatments

Trial #	Events Administered	MIoTL Status
1-5	1-31	Off
6-10	16-31	On

4.7.4 Logging.

A user log is used to record the time, device name, and action of each BLE and Wi-Fi event carried out in SHAA during experimentation. Events recorded include turning on and off loggers, starting and stopping sniffers, arriving or leaving the smart home, system errors, and activating devices. Additionally, the Raspberry Pi records each Wi-Fi event processed by the Homebridge server. These two logs are considered truth data and used to calculate the DCS, EITP, EIFP, and EIFN response variables.

4.7.5 Testing Process.

Trials are carried out over ten days. The Homebridge logger and CITIoT sniffers are activated at the beginning of each trial. At least one minute is allowed between activating sniffers and the first event to permit the logger and sniffers time to normalize. For the first five trials, each event from Table 5 is carried out in a random order in the morning and again in the evening. Also, devices are randomly activated throughout the day while the user is away. The time of each treatment is recorded in the user log. At the end of the trial, CITIoT sniffers and the Homebridge logger are deactivated and the preprocessor and MAC track units of CITIoT are started. After these components are finished operating, the classifier and network mapper units are activated. The timing measurement is built into CITIoT using Python's time module to provide the wall-clock time used by the response variable, PT. Results are stored for statistical analysis and evaluation.

The testing process is repeated during trials 6-10 with the MIoTL tool operating. As MIoTL only creates Wi-Fi traffic to impede with CITIoT's operation, a subset of treatments, which only include Wi-Fi events, is used (events 16-31). The MIoTL tool is activated at least five minutes prior to the Homebridge logger and the CITIoT Wi-Fi sniffer to allow for normalization. Wi-Fi devices are activated and the user log is maintained as in the first trials. At the end of the trial, the CITIoT Wi-Fi sniffer, the Homebridge logger, and MIoTL are deactivated. CITIoT operates similarly to previous trials after that.

4.8 Statistical Analysis

Data is collected through three main components: (i) results from CITIoT, (ii) the logger on the Raspberry Pi recording events processed by the Homebridge server, and (iii) the user logs. The accuracy of CITIoT is measured using 5 metrics: Device Classification Success Rate (DCSR), Event Identification True Positives Rate (EITPR), Event Identification False Positives Rate (EIFPR), Event Identification False Negatives Rate (EIFNR), User Tracking Success Rate (UTSR), and the Positive Predictive Value (PPV). The performance of CITIoT is measured using the Normalized Processing Time (NPT) and Normalized Hard Drive Space (NHDS).

4.8.1 Device Classification Success Rate (DCSR).

The DCSR metric measures CITIoT's ability to classify devices. The DCSR metric can be expressed by the simple ratio measurement

$$DCSR = \frac{DCS}{TD} \times 100 \tag{2}$$

where DCS represents the number of successfully classified devices and TD represents the total number of devices within SHAA.

4.8.2 Event Identification True Positives Rate (EITPR).

The EITPR metric measures CITIoT's ability to accurately identify events. The EITPR metric can be expressed by the simple ratio measurement

$$EITPR = \frac{EITP}{TE} \times 100 \tag{3}$$

where $EITP$ represents the number of true positives, or successfully identified events, and TE represents the total number of events per trial.

4.8.3 Event Identification False Positives Rate (EIFPR).

The EIFPR metric measures the rate at which CITIoT falsely identifies events that did not actually occur. The EIFPR metric can be expressed by the simple ratio measurement

$$EIFPR = \frac{EIFP}{EI} \times 100 \tag{4}$$

where $EIFP$ represents the number of false positives, or identified events that did not occur, and EI represents the total number of events identified by CITIoT.

4.8.4 Event Identification False Negatives Rate (EIFNR).

The EIFNR rate measures the rate at which CITIoT fails to identify events that did actually occur. The EIFNR metric can be expressed by the simple ratio measurement

$$EIFNR = \frac{EIFN}{TE} \times 100 \tag{5}$$

where $EIFN$ represents the total number of false negatives, or events the tool failed to identify, and TE represents the total number events per test trial. The EIFNR

metric can be simplified to

$$EIFNR = 1 - EITPR \tag{6}$$

4.8.5 User Tracking Success Rate (UTSR).

The UTSR metric measures the rate at which a user's location is accurately tracked as home or away via Wi-Fi traffic. The UTSR metric can be expressed by the simple ratio measurement

$$UTSR = \frac{UTS}{TT} \times 100 \tag{7}$$

where UTS represents the total time (minutes) which the location of the user is successfully tracked and TT is the total time (minutes) of the trial.

4.8.6 Positive Predictive Value (PPV).

The PPV metric measures the probability that CITIoT identifies an event that actually occurred. The PPV can be calculated by the simple ratio measurement

$$PPV = \frac{EITP}{EITP + EIFP} \times 100 \tag{8}$$

where $EITP$ represents the number of true positives and $EIFP$ represents the number of false positives. A higher PPV indicates confidence in CITIoT's ability to identify events, whereas, a lower PPV suggests a lack of confidence.

4.8.7 Normalized Processing Time (NPT).

The NPT metric measures the normalized PT for CITIoT. Each configuration and experimental trial analyzes a different number of packets, therefore, the NPT, measured in seconds, is found by taking the PT per 25,000 packets. The NPT metric

can be expressed by the equation

$$NPT = 25000 \times \frac{T_n}{TP_n} \times 100 \tag{9}$$

where T_n is the total time of a trial, n and TP_n is the total number of packets in trial, n.

4.8.8 Normalized Hard Drive Space (NHDS).

The NHDS metric measures the normalized HDS for CITIoT. Each trial and configuration processes a different number of packets, therefore, the NHDS is found by taking the HDS per 25,000 packets.

4.8.9 Other Statistical Analysis Measures.

These results are imported into the statistical analysis tool R, a GNU project language for statistical computing, and are compared to the two truth data sources. The DCS, EITP, EIFP, UTS, NPT, and NHDS results are tested for mean validity by computing the standard deviation, mean, and 95% confidence interval. CITIoT's accuracy (EITP and EIFP) with and without MIoTL operating is compared using a two sample t-test with a null hypothesis which asserts that the difference in means is equal to zero. The null hypothesis can be rejected if the p-value is below an alpha level of 0.05 threshold—this would indicate strong evidence that mitigation has an impact on CITIoT's accuracy. A one tailed t-test is used to determine if the mean with mitigation is significantly greater for false positives and less for true positives than the means without mitigation.

Table 7 defines each performance metric's units of measurement, accepted range value, and expected range value. Expected values are derived from hypothesized performance levels.

Table 7. Performance metrics

Metric	Units	Accepted Range	Expected Value no Mitigation	Expected Value with Mitigation
DCSR	%	0 to 100	$> 75\%$	$< 75\%$
EITPR	%	0 to 100	$> 75\%$	$< 75\%$
EIFPR	%	0 to 100	$> 75\%$	$< 75\%$
EIFNR	%	0 to 100	$> 75\%$	$< 75\%$
UTSR	%	0 to 100	$> 75\%$	$< 75\%$
PPV	%	0 to 100	$> 85\%$	$< 15\%$
NPT	seconds	0 to ∞	< 30 seconds	< 30 seconds
NHDU	bytes	0 to ∞	< 10 MB	< 10 MB

4.9 Methodology Summary

This chapter describes the experimentation methodology used to measure the performance (NPT and NHDS) and accuracy (DCSR, EITPR, EIFPR, UTSR, and PPV) of CITIoT. The treatments allow for various devices and actions to determine the operational capabilities of the tool. The effectiveness of the MIoTL tool in mitigating some of CITIoT's features is measured through the accuracy of CITIoT while the MIoTL tool is operating.

V. Results and Analysis

5.1 Overview

This chapter describes the results obtained using CITIoT during the experimentation described in Chapter 4. Results are discussed in four configurations: (i) BLE, (ii) Wi-Fi with no mitigation, (iii) combined BLE and Wi-Fi with no mitigation, and (iv) Wi-Fi with mitigation. Configurations three and four are used when analyzing CITIoT's overall accuracy and performance. Section 5.2 discusses the accuracy of CITIoT by examining the applicable metrics. The performance of CITIoT, as defined by the NPT and NHDS response variables, is reported in Section 5.3. Alibis are provided for each failure, and results are discussed to provide insight into smart home leakage and its security ramifications. Trial 1 does not contain any BLE data due to the BLE sniffer malfunction described in Section 3.4.3.1 and, therefore, is not included in accuracy or performance results.

5.2 CITIoT Accuracy

This section analyzes CITIoT's accuracy against SHAA using the DCSR, EITPR, EIFPR, EIFNR, UTSR, and PPV metrics discussed in Section 4.8. Results are presented for all four configurations and are calculated using the script presented in Appendix K. The script compares a trial's event identification output from CITIoT with the truth data logs to provide a list of true positives, false positives, and false negatives. The script also calculates the values for each of the event response variables per trial and configuration. The script provided in Appendix L formats the truth data logs to assist in comparison with CITIoT's outputs. For Wi-Fi, the script parses the Homebridge logs to determine when events occur. For BLE, the script parses the manually created truth data logs and formats the date string and device names to

match the format used by CITIoT.

Tables 8, 9, 10, and 11 provide results from experiment trials for each configuration. The R script in Appendix M calculates the standard deviation and 95% confidence interval for each response variable. Table 12 summarizes the results of CITIoT's mean accuracy across all trials in each configuration. User tracking is only accomplished via Wi-Fi devices, therefore, the UTSR of BLE trials is not considered. Lastly, the confidence of CITIoT's ability to accurately identify events is determined using the PPV. The rest of this section discusses and analyzes the significance of each metric pertinent to the tool's accuracy.

Table 8. BLE results

Date	DC	TD	DCSR%	TP	FP	FN	TE	EI	EITPR%	EIFPR%	EIFNR%
8/16/17	Data not collected due to Ubertooth malfunction discussed in Section 3.4.3.1										
8/22/17	9	12	75.0	33	2	5	38	36	86.8	5.6	13.2
8/23/17	9	12	75.0	42	1	1	43	43	97.7	2.3	2.3
8/25/17	9	12	75.0	50	5	0	50	58	100.0	8.6	0.0
8/26/17	9	12	75.0	37	0	2	39	39	94.9	0.0	5.1

DC: devices classified; TD: total devices; DCSR: Device Classification Success Rate; TP: true positives, FP: false positives; FN: false negatives; TE: total events; EI: events identified; EITPR: Event Identification True Positive Rate; EIFPR: Event Identification False Positive Rate; EIFNR: Event Identification False Negative Rate

Table 9. Wi-Fi with no mitigation results

Date	DC	TD	DCSR%	TP	FP	FN	TE	EI	EITPR%	EIFPR%	EIFNR%
8/16/17	8	8	100.0	31	0	0	31	31	100.0	0.0	0.0
8/22/17	8	8	100.0	34	1	1	35	35	97.1	2.9	2.9
8/23/17	8	8	100.0	34	2	3	37	36	91.9	5.6	8.1
8/25/17	8	8	100.0	35	1	2	37	35	94.6	2.9	5.4
8/26/17	8	8	100.0	28	1	5	33	29	84.9	3.5	15.2

5.2.1 Device Classification Success Rate (DCSR).

The DCSR metric measures CITIoT's ability to accurately classify devices within SHAA. Table 13 provides the overall mean success rate for each configuration while

Table 10. Combined BLE and Wi-Fi without mitigation results

Date	DC	TD	DCSR%	TP	FP	FN	TE	EI	EITPR%	EIFPR%	EIFNR%
8/16/17	17	18	94.4	31	0	0	31	31	100.0	0.0	0.0
8/22/17	17	18	94.4	67	3	6	73	71	91.8	4.2	8.2
8/23/17	17	18	94.4	76	3	4	80	79	95.0	3.8	5.0
8/25/17	17	18	94.4	85	6	2	87	93	97.7	6.5	2.3
8/26/17	17	18	94.4	65	1	7	72	68	90.3	1.5	9.7

Table 11. Wi-Fi with mitigation results

Date	DC	TD	DCSR%	TP	FP	FN	TE	EI	EITPR%	EIFPR%	EIFNR%
12/19/17	6	8	75.0	28	255	5	33	285	84.9	89.5	15.2
12/22/17	6	8	75.0	27	257	5	32	285	84.4	90.2	15.6
12/23/17	6	8	75.0	25	225	7	32	250	78.1	90.0	21.9
12/26/17	6	8	75.0	25	170	8	33	196	75.8	86.7	24.2
12/27/17	6	8	75.0	28	199	9	37	227	75.7	87.7	24.3

Table 12. CITIoT mean accuracy results in each configuration across all trials

Configuration	DCSR%	EITPR%	EIFPR%	EIFNR%	UTSR%
BLE (Trials 1-5)	76.9	94.8	13.0	5.2	N/A
Wi-Fi no mitigation (Trials 1-5)	100.0	93.7	2.9	6.3	99.7
Combined no mitigation (Trials 1-5)	94.4	95.0	3.2	5.1	99.7
Wi-Fi with mitigation (Trials 6-10)	75.0	79.8	88.8	20.2	1.9
DCSR: Device Classification Success Rate; EITPR: Event Identification True Positive Rate; EIFPR: Event Identification False Positive Rate; EIFNR: Event Identification False Negative Rate; UTSR: User Tracking Success Rate					

Appendix N supplies the tool's device classification output for Wi-Fi devices. CITIoT classified BLE devices with a success rate of 75.0% per trial. The Bike Lock, Apple iPhone 6+, and Apple TV were not identifiable via BLE traffic from the smart home. The Bike Lock used a pseudonym as a device name "00000b67", while both Apple devices did not provide a device name and the BLE MAC addresses were randomized. Without mitigation active, the tool successfully classified all 8 Wi-Fi devices providing a 100% success rate. Of the total 18 IoT devices employed within SHAA, the tool was able to identify 17 using one of the wireless protocols for an overall success rate of 94.4%. With mitigation employed, however, the tool was only able

to classify 6 out of 8 Wi-Fi devices for a success rate of 75%. The traffic spoofed by MIoTL caused CITIoT to categorize each device as an outlet. This feature of MIoTL hides the existence of motion sensors and cameras within the smart home from an eavesdropper using CITIoT. The success rates were consistent across all trials for each configuration, therefore the standard deviation of this response variable is not significant.

Table 13. CITIoT mean DCSR results for each configuration

Configuration	DC	TD	$DCSR_{\%}$
BLE	9	12	75.0
Wi-Fi no mitigation	8	8	100.0
Combined no mitigation	17	18	94.4
Wi-Fi with mitigation	6	8	75.0
DC: devices classified; TD: total devices; DCSR: Device Classification Success Rate			

5.2.2 Event Identification True Positives Rate (EITPR).

The EITPR metric measures CITIoT's true positive rate, or the ability of the tool to successfully identify events that occur in SHAA. An event is correctly identified if the time and device of the event recognized matches an event in the log. For sensor, camera, and BLE events, an event is considered successful if the identified time is ± 1 minute from the truth data log time. This success interval provides enough precision for the problem presented in this thesis. Table 14 provides the overall mean EITPR results, while Figure 36 shows the spread of EITPR data for each configuration across all trials. Appendix O supplies the event identification output listing each event successfully identified.

Over all trials, CITIoT identified 162 of 170 BLE events for a mean EITPR of 94.9%. For trials without mitigation, CITIoT identified 162 of 173 Wi-Fi events for a mean EITPR of 93.7%. For BLE and Wi-Fi trials combined without mitigation,

Table 14. CITIoT mean EITPR results for each configuration

Configuration	Trials	TP	TE	EITPR%	σ	C.I. %
BLE	4	162	170	94.9	5.7	± 9.1
Wi-Fi no mitigation	5	162	173	93.7	5.8	± 7.2
Combined no mitigation	5	324	343	95.0	4.0	± 5.0
Wi-Fi with mitigation	5	133	167	79.8	4.5	± 5.6
TP: true positives; TE: total events; EITPR: Event Identification True Positive Rate; C.I.: confidence interval						

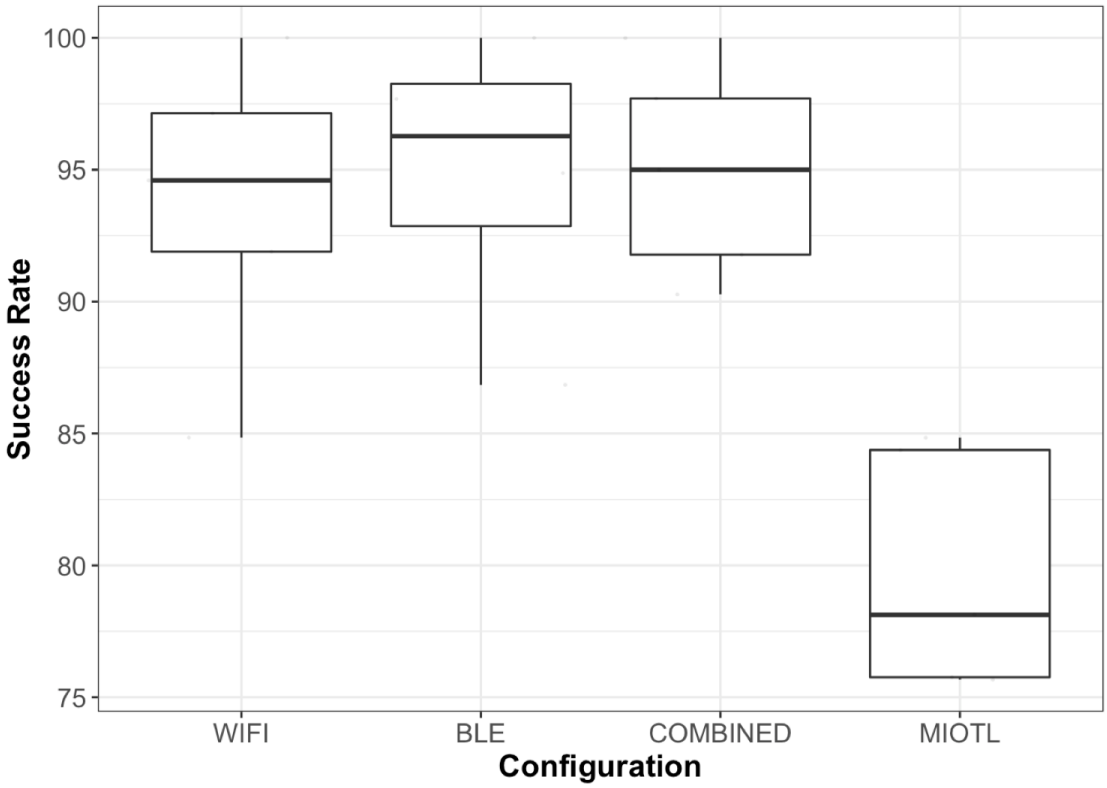


Figure 36. EITPR quartile ranges for each configuration

the tool identified a total 324 out of 343 events for a mean EITPR of 95.0%. For Wi-Fi trials with mitigation, the tool identified a total of 133 out of 167 events for a mean EITPR of 79.8%. The standard deviations and 95% confidence intervals, listed in Table 14, show that the EITPR for each trial was consistent to the mean of all trials per configuration. This provides a high confidence that the tool can consistently identify events at a combined rate greater than 90% when MIoTL is not operating.

CITIoT was not effective in identifying events during mitigation and operated at an average success rate of 79.8%. This lower level of event identification is expected as the NetCam and Motion devices are mis-categorized as outlets, therefore, no camera or sensor events would be identified. The standard deviation of the five mitigation trials is 4.1% indicating the average success rate is representative of all trials.

Next, the significance of MIoTL's impact on EITP is investigated. Using an alpha level of 0.05, the one tailed, two sample t-test indicates there is a statistically significant difference in EITPR when MIoTL is operating and when it is not ($p = 0.001426 < \alpha$). Therefore, this research clearly rejects the null hypothesis of equal means for EITP with and without MIoTL operating and accepts the alternative hypothesis that the difference in means is greater than 0. This indicates that mitigation has a statistically significant impact on CITIoT's EITPR—MIoTL decreases CITIoT's ability to successfully identify events.

5.2.3 Event Identification False Positives Rate (EIFPR).

The EIFPR metric measures CITIoT's false positive rate, or the rate at which the tool identifies events that did not occur in SHAA. An event is a false positive if there is no corresponding entry in the truth data logs; the false positive rate is represented as a ratio of falsely identified events to the total number events identified. Table 15 provides the overall mean EIFPR results, while Figure 37 provides the spread of EIFPR data for each configuration across all trials—the dots for the Wi-Fi configuration indicate two outliers in which the false positive rates were greater than the standard deviation from the mean. Appendix O supplies the tool's event identification false positives for each trial.

Of the 176 BLE events identified by CITIoT, 8 did not occur providing a mean EIFPR of 4.1%. BLE false positives occur because CITIoT identifies events based

Table 15. CITIoT mean EIFPR results for each configuration

Configuration	Trials	FP	EI	EIFPR%	σ	C.I. %
BLE	4	8	176	4.1	3.8	± 6.0
Wi-Fi no mitigation	5	5	166	2.9	.9	± 2.5
Combined no mitigation	5	13	342	3.2	2.5	± 3.1
Wi-Fi with mitigation	5	1106	1243	88.8	1.5	± 1.9
FP: false positives; EI: events identified; EIFPR: Event Identification False Positive Rate; C.I.: confidence interval						

on `CONNECT_REQ` packets sent from a master to a slave which may occur outside of smart home events. For example, a phone and a temperature sensor might create a connection to pass battery information that CITIoT would mistakenly identify as a smart home event. Of the false positives, all 8 were from one of the Eve devices and the wireless traffic is encrypted so the cause of these connections is not clear.

For trials without mitigation, five Wi-Fi events did not occur out of the 166 identified for a mean EIFPR of 2.9%. All five false positives were either motion or camera events. The CITIoT tool identifies these type of events by summing the total FSize of packets sent within a minute. If the total FSize of these packets reaches a threshold, then an event is identified. After traffic analysis, it was observed that these false positives occur when the two devices send enough packets in a minute to trigger CITIoT to falsely identify an event. The wireless traffic is encrypted so it is unclear if these false positives were caused by actual events that failed to report to the Homebridge log or other status traffic. The mean EIFPR of 2.9%, however, is well within the desired result of an EIFPR less than 10%.

Combined, 13 of the 342 BLE and Wi-Fi events identified did not occur, resulting in a total mean EIFPR of 3.2%. The standard deviations and 95% confidence intervals, provided in Table 15, show that the EIFPR for each trial was consistent to the mean of all trials per configuration. Therefore, without mitigation, CITIoT consistently produced false positives with a combined rate less than 5.3%. Mitigation

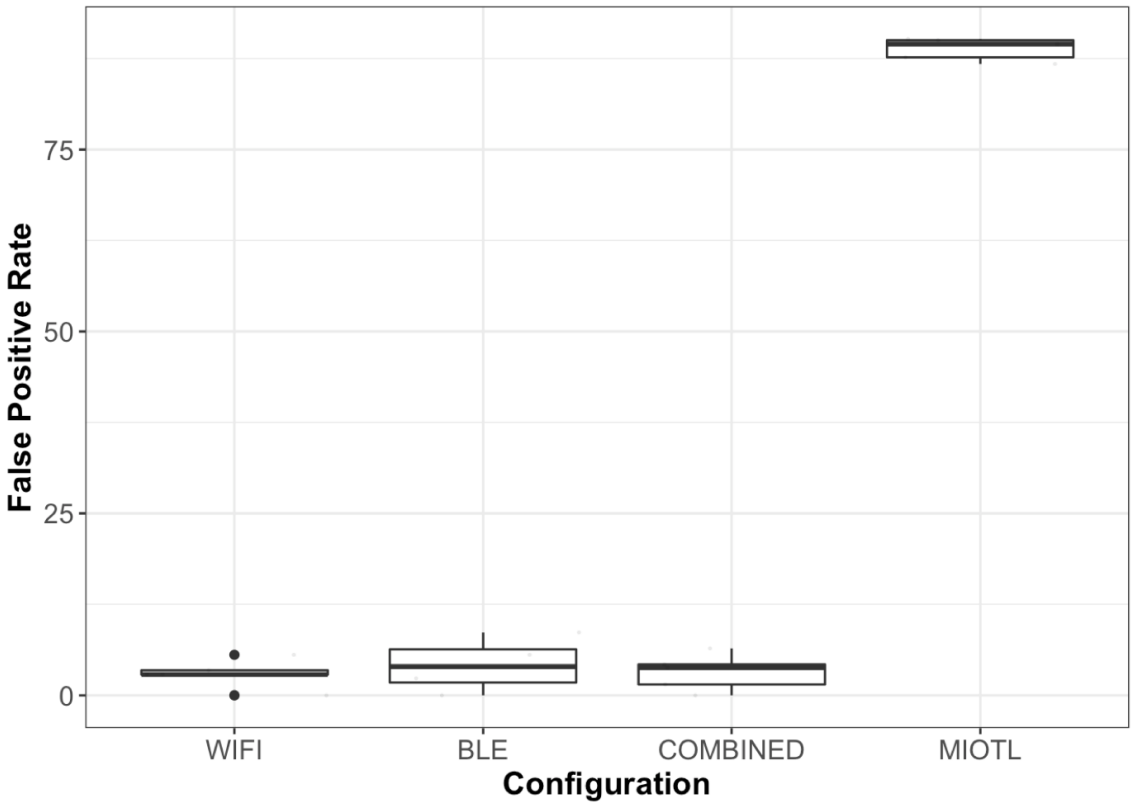


Figure 37. EIFPR quartile ranges for each configuration

increased the EIFPR for Wi-Fi devices to 88.8% and demonstrated that CITIoT is ineffective at differentiating real events from spoofed events when mitigation is active. With a standard deviation of only 1.4%, MIoTL is consistently successful in neutralizing CITIoT's event identification capability across all trials.

Next, the significance of MIoTL's impact on EIFP is investigated. Using an alpha level of 0.05, the one tailed, two sample t-test indicates that there is a statistically significant difference in EIFPR when MIoTL is operating and when it is not ($p = 4.675 \times 10^{-13} < \alpha$). Therefore, this research clearly rejects the null hypothesis of equal means for EIFP with and without MIoTL and accepts the alternative hypothesis that the difference in means is less than 0. This indicates that mitigation has a statistically significant impact on CITIoT's EIFPR—MIoTL increases the rate at which CITIoT identifies false positives.

5.2.4 Event Identification False Negatives Rate (EIFNR).

The EIFNR response variable, as provided in Section 4.3, measures CITIoT's false negative rate, or the rate at which the tool fails to identify events that occur in SHAA. Table 16 provides the EIFNR results, and Figure 38 depicts the spread of EIFNR data for each configuration across all trials. Appendix O supplies the tool's event identification output listing each false negative.

Table 16. CITIoT mean EIFNR results for each configuration

Configuration	Trials	FN	TE	EIFNR%	σ	C.I. %
BLE	4	8	170	5.2	5.0	± 9.1
Wi-Fi no mitigation	5	11	173	6.3	5.2	± 7.2
Combined no mitigation	5	19	342	5.1	3.6	± 5.0
Wi-Fi with mitigation	5	34	167	20.2	4.1	± 5.6
FN: false negatives; TE: total events; EIFNR: Event Identification False Negative Rate; C.I.: confidence interval						

CITIoT failed to identify 8 BLE events from the 170 total events resulting in a 5.2% mean EIFNR. There are three primary reasons the tool can fail to identify a BLE event: first, the `CONNECT_REQ` packet sent by the master may not be collected by the sniffers due to wireless interference; second, the `CONNECT_REQ` packet may have been collected by the sniffers, but corrupted; and third, the sniffer set to listen on the advertisement channel that a `CONNECT_REQ` packet was sent may already be following a different connection and, therefore, would not collect the `CONNECT_REQ` packet. While not conclusive, the cause of a false negative can be speculated via traffic analysis. The lack of connection events around the time of an event can indicate the first cause. A malformed `CONNECT_REQ` packet at the time of an event can point towards the second cause. A sniffer following a connection at the time of an unidentified event suggests the third reason. Traffic analysis of the 8 false negatives indicates that four of the events were likely caused by interference, while the other four could have been a result of the Ubertooth One sniffers following other connections.

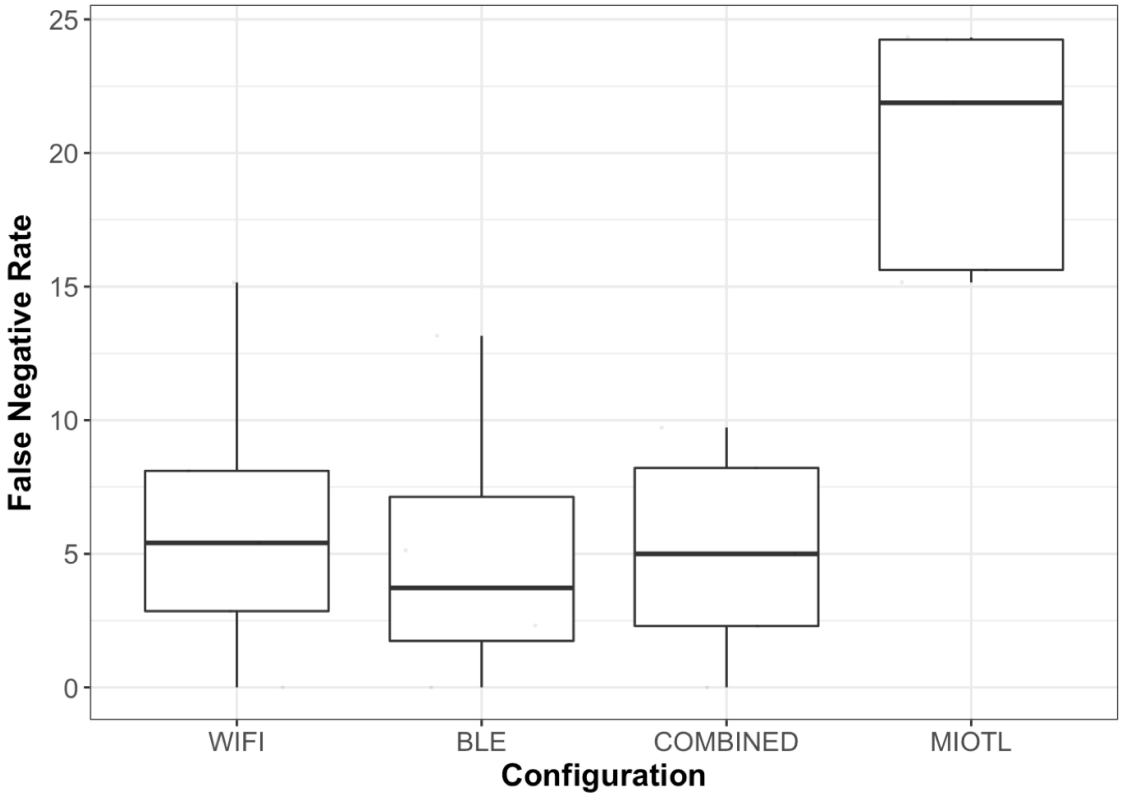


Figure 38. EIFNR quartile ranges for each configuration

Without mitigation, a total of 11 out of 173 Wi-Fi events were not identified for a mean EIFNR of 6.3%. The main reason the tool may fail to identify a Wi-Fi event is the Alfa Card sniffer fails to capture the packet sent from the Raspberry Pi to a device due to congestion. There is no way to guarantee the cause, but high levels of traffic around missed events suggests this was the cause for all 11 false negatives.

Combined, the tool failed to identify 19 of 342 BLE and Wi-Fi events with no mitigation for a mean EIFNR of 5.0% across all trials. The standard deviations and 95% confidence interval, provided in Table 16, suggest that the EIFNR for each trial was consistent to the mean of all trials per configuration. Therefore, without mitigation, CITIoT consistently fails to identify events at a rate less than 10%.

With mitigation however, CITIoT failed to identify 34 of 167 events for a mean EIFNR of 20.2%. Of the 34 events the tool failed to identify, 29 were either NetCam

or Motion events. MIoTL caused CITIoT to identify these camera and sensor devices as outlets. Consequently, CITIoT was not able to identify the sensor or camera events. The 5 outlet events that were not identified were due to dropped packets. Of the 5 outlet false negatives, 3 occurred at the same time that MIoTL sent a burst of spoofed packets which could have resulted in CITIoT experiencing a greater dropped packet rate during those times. The standard deviation of 4.1% indicates that MIoTL consistently increased the EIFNR by about 15% across all trials.

5.2.5 Positive Predictive Value (PPV).

The PPV measures the probability that an identified event actually occurred in the smart home and is a ratio of true and false positives. CITIoT had a PPV of 96.1% without mitigation suggesting that results can be trusted. With mitigation, however, the PPV was 10.7% indicating CITIoT experienced so many false positives that there is little confidence in CITIoT's operation.

5.2.6 User Tracking Success Rate (UTSR).

The UTSR response variable measures the rate at which CITIoT accurately determines if a user is in the smart home or not. Tracking is accomplished via Wi-Fi wireless traffic, so there are no results from the BLE configuration. Table 17 summarizes the tracking results, while Appendix P lists the tracking information provided by CITIoT for each trial. Without mitigation, the tool successfully tracked the user's location for all but 9 minutes out of 51 hours and 49 minutes. This resulted in a mean UTSR of 99.7% across trials. Inaccuracies in timing can be explained by the Apple iPhone 6+ connecting to the AP as the user is walking up to or away from the smart home and prior to or after the times reflected in the logs.

User tracking while MIoTL is active was much less effective. The goal of MIoTL

is to spoof messages sent from the user's device so it appears the device is always in the smart home making it impossible for CITIoT to track if a user is home or away. Therefore, UTSR during mitigation measures CITIoT's ability to identify if a user is away despite mitigation. With MIoTL active, CITIoT only identified 7.5 minutes that the user was away from the home for a mean UTSR of 1.9%. CITIoT was able to identify the device's absence during these 7.5 minutes because the spoofed packets sent by MIoTL during this period were dropped by the sniffer. Otherwise, MIoTL was successful in inhibiting CITIoT's capability to track when the user was home or away.

Table 17. CITIoT mean UTSR results

Configuration	**Trials**	**ST**$_{min}$	**TT**$_{min}$	**UTSR**$_{\%}$	σ
Wi-Fi no mitigation	5	3100	3109	99.7	0.2
Wi-Fi with mitigation	5	7.5	1473	1.9	3.9
ST: successfully tracked time; TT: total trial time; UTSR: User Tracking Success Rate					

5.3 CITIoT Performance

This section analyzes CITIoT's performance while operating against SHAA using the NPT and NHDS metrics discussed in Section 4.8. The performance parameters are presented for each configuration.

5.3.1 Normalized Processing Time (NPT).

The NPT response variable measures the average normalized processing time for each configuration. The NPT values for BLE and Wi-Fi separately were calculated while those components were operating alone. In normal operation, the BLE and Wi-Fi components operate simultaneously in two different processes. Therefore, the combined NPT with no mitigation was calculated when both components were op-

erating concurrently. Table 18 provides the NPT for each trial and the mean NPT for each configuration. The first trial does not have a BLE component, so that trial's timing is not considered in the combined average.

The mean NPT for the BLE component of CITIoT operating alone is 24.1 seconds per 25,000 packets. On average, one hour of captured BLE traffic resulted in about 441,750 packets and took the BLE component of the tool approximately 7 minutes to process. The standard deviation of 0.2 seconds indicates that each trial's NPT did not vary considerably.

The average NPT for the Wi-Fi component of CITIoT is 22.0 seconds per 25,000 packets. On average, one hour of captured Wi-Fi traffic resulted in about 417,703 packets and took the Wi-Fi component of the tool approximately 6 minutes to process. The standard deviation of 0.7 seconds indicates that the each trial's NPT did not vary considerably.

The average NPT presented for the combined Wi-Fi and BLE without mitigation takes into account the fact that the components are operating concurrently. The mean NPT was 14.6 seconds per 25,000 packets. On average, one hour of captured BLE and Wi-Fi traffic resulted in about 859,480 packets and took CITIoT approximately 8 minutes to process.

The average NPT of Wi-Fi with mitigation is very similar to Wi-Fi without mitigation as all that changes between configurations is the number of packets processed by the tool and the number of packets are normalized for this metric. The average NPT across all trials for Wi-Fi with mitigation is 22.1 seconds per 25,000 packets. On average, one hour of captured Wi-Fi packets with mitigation resulted in about 535,680 packets and took the Wi-Fi component of the tool approximately 8 minutes to process.

Table 18. CITIoT mean NPT results, in seconds, for each configuration

Component	Trial1	Trial2	Trial3	Trial4	Trial5	NPT_s	σ
BLE	N/A	24.0	24.4	24.0	24.0	24.117	0.2
Wi-Fi no mitigation	21.8	21.4	21.1	23.2	22.4	22.0	0.7
Combined no mitigation	N/A	17.4	12.4	13.9	14.6	14.6	1.8
Wi-Fi with mitigation	20.9	22.8	24.2	22.3	20.1	22.1	1.4
NPT: Normalized Processing Time							

5.3.2 Normalized Hard Drive Space (NHDS).

The NHDS metric measures the average normalized hard drive space usage for each configuration. Table 19 provides the NHDS per trial and the average NHDS per configuration. The first trial does not have a BLE component, so that trial's space usage is not considered in the combined average.

The average NHDS used for Wi-Fi and BLE without mitigation is 7.0 MB per 25,000 packets. On average, 8 million packets were captured per trial which resulted in a corresponding mean NHDS of about 1216 MB per trial.

Table 19. CITIoT mean NHDS results, in MB, for each configuration

Configuration	Trial1	Trial2	Trial3	Trial4	Trial5	$NHDS_{MB}$	σ
BLE	N/A	1.9	1.9	1.9	1.8	1.9	0.0
Wi-Fi no mitigation	5.8	3.4	3.3	7.0	6.7	5.3	1.6
Combined no mitigation	N/A	5.3	5.1	8.8	8.6	7.0	1.8
Wi-Fi with mitigation	1.8	3.7	12.6	3.7	2.4	4.8	4.0
NHDU: Normalized Hard Drive Space							

5.4 Results Summary

This section summarizes the results of all evaluation tests and provides CITIoT's overall accuracy and performance with and without mitigation. When considering CITIoT's overall accuracy and performance, the combined BLE and Wi-Fi with no mitigation configuration is used. Without mitigation, CITIoT was able to categorize

17 out of 18 devices. On average, it was able to identify 95.0% of the events within SHAA, while only identifying false positives at an average rate of 3.2%. This resulted in a PPV of 96.1% when MIoTL was inactive, indicating a high confidence in CITIoT's ability to identify events. CITIoT was able to track if a user was in the home or away with an average success rate of 99.7%. With mitigation, however, CITIoT was unable to identify motion sensor and security camera devices, identified an average of 79.8% of events, and falsely identified events at a mean rate of 88.8%. These values resulted in a PPV of 10.7% when MIoTL was active, suggesting a low confidence in CITIoT's ability to identify events. Also, CITIoT could only recognize 7.5 minutes of 24 hours that the user was away from the smart home. These results show that without mitigation, CITIoT can effectively classify devices, identify events, and track users within the home. With mitigation, however, the tool is overwhelmed by false positives, failed to identify key devices, and cannot tell when a user is away from the home.

VI. Conclusion

6.1 Overview

This chapter summarizes the research and results found during experimental evaluation. Section 6.2 reiterates notable conclusions derived from experimentation and statistical analysis. Section 6.3 synthesizes findings to underline security and privacy risks IoT devices present to the home and provides practical recommendations for future IoT security. Lastly, Section 6.4 provides possibilities for future work with the CITIoT tool.

6.2 Research Conclusions

This research was successful in analyzing data leakage through four contributions: first, designing SHAA to test and analyze IoT data leakage; second, demonstrating how vulnerabilities and characteristic data exchanges can be used to fingerprint devices; third, presenting CITIoT to show how data leakage can enable an observer to map a smart home network, classify devices within the home, identify events, and track whether a user is home or away; and fourth, providing MIoTL to help mitigate smart home device leakage.

As hypothesized, an eavesdropper was able to collect traffic from outside a smart home network to successfully identify devices, track user's presence, and deduce events such as when a door is opened or when a camera senses motion inside the home.

Device classification was 94.4% successful with all 8 Wi-Fi devices and 9 out of 12 BLE devices categorized each day. On average, CITIoT was able to identify 95.0% of the events that occurred within the smart home and only identified 3 events per day that did not actually occur. This low false positive rate provided confidence in CITIoT's ability to accurately identify events. The tool was also 99.7% successful in

determining if the user was in the home or not.

When the MIoTL tool was operating, however, CITIoT was much less accurate. All of the Wi-Fi devices were categorized as outlets which concealed the motion sensor and security camera. An average of 79.8% events were successfully identified, but MIoTL caused CITIoT to falsely identify about 221 events per day that did not occur. This high false positive rate resulted in a lack of confidence that CITIoT could accurately identify events during mitigation. Additionally, the tool could only recognize 7.5 minutes of 24 hours that the user was away from the home when MIoTL was operating.

These results provide statistical evidence supporting the original hypothesis: without mitigation, CITIoT was able to classify devices, identify events, and track users, but with mitigation, the number of events identified was decreased and the number of false positives was increased making CITIoT ineffective at distinguishing real events from spoofed events.

A total of 11.2 GB, 63.2 million packets, and 94.8 hours of traffic was captured throughout the 10 trials accomplished during experimentation. At an average rate of 860,000 packets captured per hour, CITIoT was able to process one hour's worth of captured traffic in about 8 minutes. CITIoT used a mean 7 MB of storage per 25,000 packets captured and, therefore, a one hour traffic capture took up about 240 MB of hard drive space.

6.3 Research Significance and Synthesis

As the modern home gets smarter it also becomes more vulnerable to attacks that were previously reserved to computers and networks. IoT devices constantly communicate data that enable an eavesdropper to infer information about people and devices within a smart home. Users must be aware of what their devices are

advertising and how this information can be used against them.

For example, using CITIoT's output from experimentation a few observations can be made about the user and smart home that have significant security implications: (i) the user was away from the home between 0800-1100 four days out of the week, (ii) the user used a BLE lock to secure their home, and (iii) the user employs a Wi-Fi based security camera and motion sensor in the home. It was also observed that the communication between the iPhone and BLE lock was not encrypted and passwords were sent in the clear. Using the sniffed passwords, the times when the user was away from the home, and a replay attack, the door was unlocked by the eavesdropper at will. The eavesdropper can also change the passwords locking the user out of the home. The adversary knows to be aware of a Wi-Fi camera and motion sensor. Furthermore, an eavesdropper could deploy multiple Raspberry Pi devices with CITIoT running on each to collect data simultaneously from numerous smart homes, something not possible with conventional reconnaissance methods. If MIoTL was operating in the smart home, however, the adversary would not have been able to track when the user was away nor be aware of the existence of the Wi-Fi security camera and motion sensor.

This example is directly relevant to the military or DoD and has implications outreaching that of the Strava incident due to the threat on physical security presented in this thesis. The smart environment may be implemented in a DoD leader's office or home approved to store classified or sensitive information. Every government official carries at least one mobile phone that, using the presented methods, can enable an eavesdropper to track movement or determine when officials have left the office or home. Lastly, device vulnerabilities found in BLE locks allow unauthorized access to homes, offices, or storage containers that could contain sensitive information. These findings stress the importance of ongoing evaluation and assessment of the security

risk of IoT in the DoD.

Many of the vulnerabilities used in this work take advantage of information that is not encrypted at the lower levels of the Wi-Fi and BLE protocols, therefore, to create more secure smart devices, developers must consider security from the physical layer on up. For Wi-Fi, this includes periodically changing MAC addresses, randomizing FSize of event packets, and encrypting lower-layer data packets. In BLE, devices need to make their advertisements private. Also, common operational security methods can help prevent against smart home device attacks. For example, users should be aware that routine schedules leave them vulnerable to pattern-of-life modeling—a threat which is increased by smart devices. Maintaining unpredictable schedules will help prevent these types of attacks. Similarly, turning on or off lights while away from home can trick an observer into thinking someone is home. It is also important to have situational awareness of potential eavesdroppers or suspicious devices around when accessing smart locks or other devices.

While these recommendations can improve the security of smart home environments, none of these ideas are new. Why, then, have these fixes not been implemented to secure against privacy leakage? In response to rapid growth of the IoT market, efforts to limit power, develop devices quickly, and other design constraints are driving developers toward poor security implementation, leaving devices vulnerable. Also, while the areas of network and computer security have seen more adversarial pressure, the smart home is relatively new. Until recently, outlets, locks, and light-bulbs were not connected to networks. This is the same evolution vehicles have seen over the past five years as they have become connected to the Internet and, therefore, open to attack. The privacy implications demonstrated in this work, however, require that developers of IoT technologies consider security in design and engage with the computer security community to create more secure smart homes.

6.4 Future Work

There are a number of avenues to take in extending the CITIoT system as the field of IoT devices and security is constantly growing. The following suggests seven future work options based off this research:

1. The number, type, and manufacturer of devices used within SHAA can be expanded to test if the fingerprinting methods presented in the classifier extend to other IoT devices. Only a limited selection of Wi-Fi devices were used in training the classifier.

2. The MAC tracker unit only observes when a user is home or away and could be expanded to track device locations within the home for better reconnaissance and tracking. A more significant tracking capability would greatly increase CITIoT's ability to stress the security implications of smart home devices.

3. The classifier currently uses a hands-on method for training. Machine learning may be able to assist in training the classifier more efficiently and accurately.

4. Program execution can be improved. Processing time and hard drive usage are limiting factors for a small battery powered device such as a Raspberry Pi and could limit the capabilities of remotely operating CITIoT.

5. The Python wrapper, Pyshark has the capability to process packets captured in real-time. Adding a real-time mode to CITIoT can increase the effectiveness of demonstrating smart home vulnerabilities.

6. Mitigation is only effective against CITIoT's Wi-Fi capabilities. Introducing BLE mitigation techniques similar to those presented in this work can expand MIoTL's ability to defend against data leakage in smart homes.

7. The impact of MIoTL's operation on SHAA was not evaluated in this thesis. The amount of traffic sent by mitigation may have a negative effect on Wi-Fi network performance and should be enumerated. Additionally, the power used to operate mitigation may be cost prohibiting and should also be evaluated.

Appendix A. BLE Sniffer Script

```
1  #!/bin/bash
2  # Ubertooth scan with three Ubertooth One sniffers
3  # Each sniffer will listen for connection events on one of three
4  # advertisement channels (37, 38, 39)
5  # will save the combined pcap into a file
6
7  function pause(){ read -p "$*" }
8
9  echo "Type desired output name for PCAP (no spaces), followed by [
       ENTER]:"
10
11 read name
12
13 # If the files exist, delete them
14 if [ -e cap0.pcap ]; then rm cap0.pcap
15 fi
16 if [ -e cap1.pcap ]; then rm cap1.pcap
17 fi
18 if [ -e cap2.pcap ]; then rm cap2.pcap
19 fi
20 # if the output capture filename already exists, ask to overwrite
21 if [ -e $name.pcap ]; then
22   read -p "File already exists, overwrite (y/n)? : " -n 1 -r
23   echo
24   if [[ $REPLY =~ ^[Yy]$ ]]
25     then rm $name.pcap; echo 'removed'
26   else
27     [[ "$0" = "$BASH_SOURCE" ]] && exit 1 || return 1
28   fi
29 fi
30 # start BLE capture on 3 Ubertooth One sniffers with each set to
       follow connections, on a different advertising channel
31 # and ouptut files to a capture
32 ubertooth-btle -f -U0 -A37 -qcap0.pcap & ubertooth-btle -f -U1 -A38 -
       qcap1.pcap & ubertooth-btle -f -U2 -A39 -qcap2.pcap
33
34 pause 'Press [Enter] key to continue...'
35
36 # merge the captures to one file with the name provided by the user
37 mergecap cap0.pcap cap1.pcap cap2.pcap -w $name.pcap
38 rm cap0.pcap; rm cap1.pcap; rm cap2.pcap
```

Appendix B. Wi-Fi Script

```
#!/usr/bin/python
# wifi.py
# Script that includes the Wi-Fi preprocessor, mac tracker, trainer,
# classifier, network mapper components of CITIoT. Includes timing
# information for each component and the tool as a whole.
# Required: helpers.py
import helpers
# https://github.com/KimiNewt/pyshark or via pip install pyshark
import pyshark
import matplotlib.dates as m_dates
import sys, getopt, csv, os, datetime, time, logging

######################################################################
# GLOBAL VARIABLES
# defined MAC addresses
ROUTER = 'ec:4f:82:73:d1:1c'
RASPI = 'b8:27:eb:09:1a:81'

# list of Wi-Fi devices (IoT devices, router, and Raspberry Pi)
WIFI_DEVICES = {'b8:27:eb:09:1a:81', '14:91:82:24:dd:35',
                '60:38:e0:ee:7c:e5', 'a0:18:28:33:34:f8',
                '08:66:98:ed:1e:19', 'b4:75:0e:0d:33:d5',
                'b4:75:0e:0d:94:65', '94:10:3e:2b:7a:55',
                '14:91:82:c8:6a:09', 'ec:1a:59:f1:fb:21',
                'ec:1a:59:e4:fd:41', 'ec:4f:82:73:d1:1c'}

# list of IoT devices ascertained via the OUI results
IOT_DEVICES = {'14:91:82:24:dd:35', '60:38:e0:ee:7c:e5',
               'b4:75:0e:0d:33:d5', 'b4:75:0e:0d:94:65',
               '94:10:3e:2b:7a:55', '14:91:82:c8:6a:09',
               'ec:1a:59:f1:fb:21', 'ec:1a:59:e4:fd:41'}

# list of devices and associated IDs used for the network mapper
DEVICE_ID = {'ec:4f:82:73:d1:1c': 's01', 'ec:1a:59:e4:fd:41': 's02',
             'b4:75:0e:0d:33:d5': 's03', 'b4:75:0e:0d:94:65': 's04',
             '94:10:3e:2b:7a:55': 's05', '14:91:82:c8:6a:09': 's06',
             'ec:1a:59:f1:fb:21': 's07', '14:91:82:24:dd:35': 's08',
             '60:38:e0:ee:7c:e5': 's09', 'b8:27:eb:09:1a:81': 's10',
             'a0:18:28:33:34:f8': 's11', '08:66:98:ed:1e:19': 's12'}

DEVICE_NAME = {'ec:4f:82:73:d1:1c': 'Router',
               'ec:1a:59:e4:fd:41': 'NetCam',
```

```
43              'b4:75:0e:0d:33:d5': 'Switch1',
44              'b4:75:0e:0d:94:65': 'Switch2',
45              '94:10:3e:2b:7a:55': 'Switch3',
46              '14:91:82:c8:6a:09': 'Switch4',
47              'ec:1a:59:f1:fb:21': 'Motion',
48              '14:91:82:24:dd:35': 'Insight',
49              '60:38:e0:ee:7c:e5': 'Mini',
50              'b8:27:eb:09:1a:81': 'Pi',
51              'a0:18:28:33:34:f8': 'iPhone',
52              '08:66:98:ed:1e:19': 'AppleTV'}
53
54  # directory to store and read csv files for devices
55  SRC_DIR = './Source/'
56  DST_DIR = './Destination/'
57
58  # seconds elapsed until a device is considered not in the home
59  MAC_TRACK_TIME = 300
60  # global variables used for timing
61  path_name = os.getcwd()
62  DATE = path_name[path_name.rindex('/')+1:]
63  PROC_TIME = "wifi_processing_time_" + DATE + ".csv"
64  # number of packets to normalize timing
65  TIMING_PKT_NUMBER = 25000
66  ##########################################################################
67
68  def main(argv):
69      """
70      Main function that calls appropriate functions depending
71      on operation mode selected by user input
72
73      The main function is called in four modes:
74      wifi.py -h/--help: Provides usage.
75      wifi.py -p <pcap file>: Begins preprocessing on the file capture
76          provided and tracks devices.
77      wifi.py -t: Provides graphs to help training classifier.
78          Preprocessing mode must be ran first.
79      wifi.py -c: Classifies devices, identifies events, and maps
80          the network.
81      """
82      try:
83          opts, args = getopt.getopt(argv, "hp:tc", ["help"])
84      except getopt.GetoptError:
85          print 'usage error'
```

```
86          print 'for preprocessing/tracking: wifi.py -p <pcap file>'
87          print 'for training: wifi.py -t'
88          print 'for classification/network mapping: wifi.py -c'
89          print 'exiting'
90          sys.exit(2)
91      for opt, arg, in opts:
92          if opt in ("-h", "--help"):
93              print 'for preprocessing /tracking: wifi.py -p <pcap file>'
94              print 'for training: wifi.py -t'
95              print 'for classification/network mapping: wifi.py -c'
96              sys.exit()
97
98          # preprocessing unit
99          elif opt == '-p':
100             start_time = time.time()
101             preprocessor(arg)
102             print "Finish preprocessor:", time.time() - start_time
103
104         # training unit for classifier
105         elif opt == '-t':
106             start_time = time.time()
107             trainer()
108             print "Finish trainer:", time.time() - start_time
109
110         # classifier unit
111         elif opt == '-c':
112             start_time = time.time()
113             classifier()
114             print "Finish classifier:", time.time() - start_time
115
116 ######################################################################
117 # PREPROCESSOR UNIT
118 ######################################################################
119
120 def preprocessor(file_name):
121     """
122     Unit that parses each packet in file capture and stores
123     packets into CSV files. Also provides device tracking information.
124
125     Parses capture file and stores 4-tuples in the form [time, frame
126     size, source, destination] for each packet into two files for
127     each device. Device source files include 4-tuples in which every
128     tuple has the device as the source MAC address. Device destination
```

```
129     files include 4-tuples in which every tuplehas the device as the
130     destination MAC address. Also provides a CSV file with device
131     addresses, arrival time, and departure time.
132
133     Parameters
134     ----------
135     file_name: (string) File name of capture provided by user.
136
137     Returns
138     ----------
139
140     """
141     # preprare timing
142     helpers.delete_file(PROC_TIME)
143     pt_file = open(PROC_TIME, 'w')
144     csv.writer(pt_file).writerow(["Unit", "Total Packets Processed", "
            Total Process Time", "Average Process Time"])
145     pt_file.close()
146     pkt_cntr = 0
147     total_time_preproc = 0
148     total_time_mac = 0
149
150     total_time_start = time.time()
151     # initialize dictionaries to store file object for each device
152     tgt_files_by_src = {}
153     tgt_files_by_dst = {}
154
155     # initialize dictionary for tracking of each device
156     macs = {}
157
158     # initialize file names
159     cap = pyshark.FileCapture(file_name)
160     mac_track_file = "mac_track_" + DATE + ".csv"
161     helpers.delete_file(mac_track_file)
162     helpers.init_dirs()
163
164     # obtain time for first packet
165     prev_pkt_time = cap[0].frame_info.time_epoch
166
167     # open target files to write output to
168     for device in WIFI_DEVICES:
169         tgt_files_by_src[device] = open(SRC_DIR + device.replace(':', '
                .') + ".csv", 'a')
```

```
170         tgt_files_by_dst[device] = open(DST_DIR + device.replace(':', '
                .') + ".csv", 'a')
171     tgt_mac_track_file = open(mac_track_file, 'a')
172 
173     # counter used by MAC Track unit to keep track of pkts
174     # observed with non-sequential timing
175     timing_cntr = 0
176 
177     # iterate through each packet in the capture, store tuples
178     # to files, and track when devices are on the network
179     for pkt in cap:
180         pkt_cntr += 1
181 
182         # mac track
183         time_start = time.time()
184         prev_pkt_time, macs, timing_cntr = mac_track(pkt,
                tgt_mac_track_file, prev_pkt_time, macs, timing_cntr)
185         total_time_mac += (time.time() - time_start)
186 
187         time_start = time.time()
188         if pkt.highest_layer == 'DATA':
189             parse_pkt(pkt, tgt_files_by_src, tgt_files_by_dst)
190             total_time_preproc += time.time() - time_start
191 
192     time_start = time.time()
193 
194     # using the last packet in the capture,
195     # check which devices are still on the network
196     mac_track_final(tgt_mac_track_file, prev_pkt_time, macs)
197 
198     total_time_mac += time.time() - time_start
199 
200     total_time = time.time() - total_time_start
201 
202     # close files
203     for k, v in tgt_files_by_src.iteritems():
204         v.close()
205     for k, v in tgt_files_by_dst.iteritems():
206         v.close()
207     tgt_mac_track_file.close()
208 
209     # run the classifier and add time to total Wi-Fi timing
210     classifier()
```

```
211     final_time = time.time()
212
213     # calculate times
214     normalized_total_time = (TIMING_PKT_NUMBER * total_time) /
            pkt_cntr
215     normalized_mac_time = (TIMING_PKT_NUMBER * (total_time -
            total_time_preproc)) / pkt_cntr
216     normalized_preproc_time = (TIMING_PKT_NUMBER * (total_time -
            total_time_mac)) / pkt_cntr
217
218     # write timing information to file
219     with open(PROC_TIME, 'a') as pt_file:
220         csv.writer(pt_file).writerow(["MAC Track+Preproc", pkt_cntr,
                total_time, normalized_total_time])
221         csv.writer(pt_file).writerow(["MAC Track", pkt_cntr,
                total_time_mac, normalized_mac_time])
222         csv.writer(pt_file).writerow(["Preprocesser", pkt_cntr,
                total_time_preproc, normalized_preproc_time])
223         csv.writer(pt_file).writerow(["Start and finish time",
                total_time_start, final_time, final_time-total_time_start])
224
225
226 def parse_pkt(pkt, tgt_files_by_src, tgt_files_by_dst):
227     """
228     Parses each 802.11 Data packet within the provided capture file.
229
230     Extracts the time, frame size, source, and destination of each
231     802.11 Data packet. Stores the resulting 4-tuple into two files:
232     one for the source and one for the destination of the packet.
233
234     Parameters
235     ----------
236     pkt: (pyshark packet) Pyshark packet object containing packet
237         information from capture.
238     tgt_files_by_src: (file object) Uses the device address as keys
239         and file objects as values. Every packet with a source
240         address corresponding to the key will be appended to the file
241         object.
242     tgt_files_by_dst: (dictionary) Uses the device address as keys
243         and file objects as values. Every packet with a destination
244         address corresponding to the key will be appended to the file
245         object.
246
```

```
247     Returns
248     ----------
249     void
250     """
251     try:
252         pkt_dst = pkt.wlan.da
253         pkt_src = pkt.wlan.sa
254         if (pkt_src in WIFI_DEVICES) and (pkt_dst in WIFI_DEVICES):
255             pkt_len = pkt.length
256             pkt_time = helpers.pretty_time(pkt.frame_info.time_epoch)
257             file_input = [pkt_time, pkt_len, pkt_src, pkt_dst]
258             csv.writer(tgt_files_by_src[pkt_src]).writerow(file_input)
259             csv.writer(tgt_files_by_dst[pkt_dst]).writerow(file_input)
260 
261     except AttributeError:
262         # packet does not contain an 802.11 attribute
263         # or is corrupt so ignore it
264         print "ignored: ", pkt.number
265         pass
266 
267 
268 def mac_track(pkt, tgt_mac_track_file, prev_pkt_time, macs,
        timing_cntr):
269     """
270     Records devices with no network traffic for more than
271     five minutes.
272 
273     Compares the time of each packet with the last time a device
274     sent a message and, if greater tha five minutes, marks the
275     device as no longer present and stores the device MAC address,
276     first time the device sent a packet (arrival time), and last
277     time the device sent a packet (departure time) in a csv file.
278 
279     Parameters
280     ----------
281     pkt: (pyshark packet) Pyshark packet object containing packet
282         information from capture.
283     tgt_mac_track_file: (file object) CSV file to append tracking
284         data.
285     prev_pkt_time: (frame_info.time_epoch) Time value obtained
286         from the previous packet.
287     macs: (dictionary) Uses the device address as keys and a 2-tuple
288         containing the arrival time and departure time for each device.
```

```
289     timing_cntr: (int) Counter used to keep track of number of packets
290         with non-sequential timestamps
291
292     Returns
293     ----------
294     pkt_time: (frame_info.time_epoch) Current packet time.
295     macs: (dictionary) Updated list of device addresses with 2-tuple
296         arrival and departure time.
297     """
298     # get packet time of every packet to compare time
299     pkt_time = pkt.frame_info.time_epoch
300     diff = float(pkt_time) - float(prev_pkt_time)
301
302     # check if two consecutive packets are less than 5 seconds apart
303     # and the new packet is not negative to ensure corrupt or out of
304     # sequence packets do not provide invalid time
305     if float(pkt_time) < 0 or diff > 5:
306         timing_cntr += 1
307         # provide error information to assist in troubleshooting
308         print "packet number for time error: ", pkt.number
309         print "prev pkt time: ", prev_pkt_time
310         print "current pkt time: ", pkt_time
311         return prev_pkt_time, macs
312         # sometimese timing jumps occur due to pauses in sniffer,
313         # if it appears that a new time needs to be set as the
314         # previous pkt time (more than 5 pkts with a new time)
315         # then update prev_pkt_time to current pkt_time
316         if timing_cntr > 5:
317             timing_cntr = 0
318             return pkt_time, macs, timing_cntr
319         else:
320             return prev_pkt_time, macs, timing_cntr
321     else:
322         timing_cntr = 0
323
324     try:
325         pkt_src = pkt.wlan.sa
326         if pkt_src in WIFI_DEVICES:
327
328             # if first time seeing the packet the packet source then
329             # add it with the time set as the arrive and depart
330             # if the packet is still in the collection, then update
331             # the depart time
```

```
332             if pkt_src not in macs:
333                 macs[pkt_src] = [pkt_time, pkt_time]
334             else:
335                 macs[pkt_src][1] = pkt_time
336 
337             # check each device in the collection of macs to see
338             # if it has been more than five minutes since the last
339             # time seeing it
340             for k, v in macs.items():
341                 if (float(pkt_time) - float(v[1])) > MAC_TRACK_TIME:
342                     file_input = [k, helpers.pretty_time(v[0]), helpers.
                            pretty_time(v[1])]
343                     csv.writer(tgt_mac_track_file).writerow(file_input)
344                     del macs[k]
345 
346     except AttributeError:
347         # ignore packets that aren't 802.11
348         pass
349 
350     return pkt_time, macs
351 
352 
353 def mac_track_final(tgt_mac_track_file, pkt_time, macs):
354     """
355     Records status of devices after last packet in capture.
356 
357     After the last packet is parsed, store each device MAC along with
358     arrival time and time of last packet. This provides timing
359     information for devices still on the network at the end of the
360     packet capture.
361 
362     Parameters
363     ----------
364     tgt_mac_track_file: (file object) CSV file to append tracking
365         data.
366     pkt_time: (frame_info.time_epoch) Time value obtained from
367         the last packet.
368     macs: (dictionary) Uses the device address as keys and a 2-tuple
369         containing the arrival time and departure time for each device.
370 
371     Returns
372     ----------
373     Void
```

```
374     """
375     for k, v in macs.items():
376         file_input = [k, helpers.pretty_time(v[0]), helpers.pretty_time
            (pkt_time)]
377         csv.writer(tgt_mac_track_file).writerow(file_input)
378         del macs[k]
379 
380 ######################################################################
381 # TRAINING UNIT
382 ######################################################################
383 
384 
385 def trainer():
386     """
387     Unit that provides graphs to help train classifier.
388 
389     Provides user with two graphs for each device showing each packet
390     sent to a device and each packet sent from device. Stores these
391     graphs into files.
392 
393     Parameters
394     ----------
395 
396     Returns
397     ----------
398 
399     """
400     total_time_start = time.time()
401 
402     helpers.init_training_dirs()
403 
404     pkt_cntr = training_by_dst()
405 
406     pkt_cntr += training_by_src()
407 
408     total_time = time.time() - total_time_start
409     print "Total number of packets: ", pkt_cntr
410     print "Total time: ", total_time
411     print "Average time: ", (TIMING_PKT_NUMBER*total_time)/pkt_cntr
412 
413     with open(PROC_TIME, 'a') as pt_file:
414         csv.writer(pt_file).writerow(["MAC Track", pkt_cntr, total_time
            , (TIMING_PKT_NUMBER * total_time) / pkt_cntr])
```

```
415
416
417 def training_by_dst():
418     """
419     Provide graphs to help train classifier.
420
421     Provides user with a graphical representation of each packet
422     sent to a device and saves each graph to a file.
423
424     Parameters
425     ----------
426
427     Returns
428     ----------
429     packets: (int) number of packets processed
430
431     """
432     grph_plots = []
433     grph_names = []
434
435     packets = 0
436
437     # gather all packets sent to a device by using the destination
438     # csv files from preprocessing that contain packets sent to a
439     # particular device
440     for filename in os.listdir(DST_DIR):
441         device_by_dst = []
442         device = filename.replace('.csv', '').replace('.', ':')
443
444         if device in IOT_DEVICES:
445             # load all packets into a list
446             with open(DST_DIR + filename, 'rb') as curr_file:
447                 reader = csv.reader(curr_file)
448                 contents = list(reader)
449
450             # create list of packets from Raspi to device
451             for line in con
452                 packets += 1
453                 pkt_time = line[0]
454                 pkt_size = int(line[1])
455                 pkt_src = line[2]
456                 pkt_dst = line[3]
457                 if pkt_src == RASPI:
```

```
458                     device_by_dst.append([pkt_time, pkt_size, pkt_src,
                            pkt_dst])
459
460             # setup formatting and graph values
461             if len(device_by_dst) != 0:
462                 dates = [datetime.datetime.strptime(d[0], '%Y-%m-%d %H
                        :%M:%S') for d in device_by_dst]
463                 dates = m_dates.date2num(dates)
464                 values = [d[1] for d in device_by_dst]
465                 grph_plots.append([dates, values])
466                 grph_names.append("Packets sent from Raspberry Pi to "
                        + DEVICE_NAME[device])
467
468     # create the graphs, show them, and save them
469     my_graph_2 = helpers.Graph(grph_plots, grph_names, '%m-%d %H:%M:%S
            ', "Pkts from Raspberry Pi to device")
470     my_graph_2.graph()
471     my_graph_2.save_files()
472     my_graph_2.delete()
473     return packets
474
475
476 def training_by_src():
477     """
478     Provide graphs to help train classifier.
479
480     Provides user with a graphical representation of each packet sent
481     from a device and saves each graph to a file.
482
483     Parameters
484     ----------
485
486     Returns
487     ----------
488     packets: (int) number of packets processed
489
490     """
491     grph_plots = []
492     grph_names = []
493     packets = 0
494
495     # gather all packets sent from a device by using the source csv
496     # files from preprocessing that contain packets sent from a
```

```
497     # particular device
498     for filename in os.listdir(SRC_DIR):
499         device_by_src = []
500         device = filename.replace('.csv', '').replace('.', ':')
501 
502         if device in IOT_DEVICES:
503             # load all packets into a list
504             with open(SRC_DIR+filename, 'rb') as curr_file:
505                 reader = csv.reader(curr_file)
506                 contents = list(reader)
507 
508                 # create list of packets from device to router
509                 for line in contents:
510                     packets += 1
511                     pkt_time = line[0]
512                     pkt_size = int(line[1])
513                     pkt_src = line[2]
514                     pkt_dst = line[3]
515 
516                     if pkt_dst == ROUTER:
517                         device_by_src.append([pkt_time, pkt_size,
                                pkt_src, pkt_dst])
518 
519                 # setup formatting and graph values
520                 if len(device_by_src) != 0:
521                     dates = [datetime.datetime.strptime(d[0], '%Y-%m-%d
                            %H:%M:%S') for d in device_by_src]
522                     dates = m_dates.date2num(dates)
523                     values = [d[1] for d in device_by_src]
524                     grph_plots.append([dates, values])
525                     grph_names.append("Packets sent from " + DEVICE_NAME
                            [device] + " to Router")
526 
527     # create the graphs, show them, and save them
528     my_graph = helpers.Graph(grph_plots, grph_names, '%m-%d %H:%M', "
            Pkts from device to Router")
529     my_graph.graph()
530     my_graph.save_files()
531     my_graph.delete()
532     return packets
533 
534 
535 ######################################################################
```

```
536 # CLASSIFIER UNIT
537 ##############################################################################
538
539
540 def classifier():
541     """
542     Unit that classifies devices, identifies events, and information
543     to create a network map.
544
545     Provides user with three CSV files:
546         (i) A CSV file which contains each device and corresponding
547             classification type.
548         (ii) A CSV file which contains the time, source, and
549             destination for each event.
550         (iii) A CSV file which contains the total size of all
551             packets sent between two devices.
552
553     Parameters
554     ----------
555
556     Returns
557     ----------
558
559     """
560     # setup timing information
561     total_time_class = 0
562     total_time_map = 0
563
564
565     total_time_start = time.time()
566
567     # initialize file names
568     dev_cat_file = "wifi_devices_" + DATE + ".csv"
569     helpers.delete_file(dev_cat_file)
570     event_id_file = "wifi_events_" + DATE + ".csv"
571     helpers.delete_file(event_id_file)
572     network_edge_file = "network_edge_" + DATE + ".csv"
573     helpers.delete_file(network_edge_file)
574
575     start_time = time.time()
576     # open network edge csv file to write nodes and edges into
577     with open(network_edge_file, 'a') as network_file:
578         # initialize first line of network edge
```

```
579         csv.writer(network_file).writerow(['from', 'to', 'weight'])
580     total_time_map += time.time() - start_time
581 
582     start_time = time.time()
583     # classify devices and identify events by destination
584     device_categorization, event_identification, pkt_cntr =
            events_by_dst()
585     total_time_class += time.time() - start_time
586 
587     # identify events by destination
588     event_identification, pkts, t_class, t_map = events_by_src(
            network_edge_file, device_categorization, event_identification,
             total_time_class, total_time_map)
589 
590     # count the number of packets processed by classifier
591     pkt_cntr += pkts
592     total_time_class += t_class
593     total_time_map += t_map
594 
595     start_time = time.time()
596     # write device categories to file
597     with open(dev_cat_file, 'a') as curr_file:
598         for k, v in device_categorization.iteritems():
599             csv.writer(curr_file).writerow([DEVICE_NAME[k], v])
600 
601     # write events to file
602     with open(event_id_file, 'a') as curr_file:
603         for event in sorted(event_identification):
604             csv.writer(curr_file).writerow(event)
605     total_time_class += time.time() - start_time
606 
607     # calculating timing information
608     total_time = time.time() - total_time_start
609     normalized_total_time = (TIMING_PKT_NUMBER * total_time)/pkt_cntr
610     normalized_class_time = (TIMING_PKT_NUMBER * (total_time -
            total_time_map))/pkt_cntr
611     normalized_map_time = (TIMING_PKT_NUMBER * (total_time -
            total_time_class))/pkt_cntr
612 
613     with open(PROC_TIME, 'a') as pt_file:
614         csv.writer(pt_file).writerow(["Class+NtwkMapper", pkt_cntr,
                total_time, normalized_total_time])
```

```
615         csv.writer(pt_file).writerow(["Classifier", pkt_cntr,
                total_time_class, normalized_class_time])
616         csv.writer(pt_file).writerow(["Network Mapper", pkt_cntr,
                total_time_map, normalized_map_time])
617
618
619 def events_by_dst():
620     """
621     Identify devices and identify events by destination.
622
623     Uses the files created in preprocessing which contain packets with
624     the device as the destination to classify devices and identify
625     events according to classifier parameters.
626
627     Parameters
628     ----------
629
630     Returns
631     ----------
632     device_categorization: (dictionary) Uses device addresses as keys
633         and assigned category as values.
634     event_identification: (list) 2-D list containing the time, source,
635         and destination of each identified event
636     pkt_cntr: (int) the number of packets processed by classifier
637     """
638     device_categorization = {}
639     event_identification = []
640     pkt_cntr = 0
641     # analyze all of the packets sent to a device using the
642     # dest csv files from preprocessing that contain packets
643     # destined for a particular device
644     for filename in os.listdir(DST_DIR):
645         device_by_dst = []
646         device = filename.replace('.csv', '').replace('.', ':')
647         events = []
648
649         # if the file contains packets destined to an IoT device,
650         # then read the file into a list
651         if device in IOT_DEVICES:
652             device_category = 'UNKNOWN'
653             with open(DST_DIR + filename, 'rb') as curr_file:
654                 reader = csv.reader(curr_file)
655                 contents = list(reader)
```

```
656
657             # for each packet in the file, obtain the time,
658             # frame size, source, and destination and store
659             for line in contents:
660                 pkt_cntr += 1
661                 pkt_time = line[0]
662                 pkt_size = int(line[1])
663                 pkt_src = line[2]
664                 pkt_dst = line[3]
665                 if pkt_src == RASPI:
666                     device_by_dst.append([pkt_time, pkt_size, pkt_src,
                            pkt_dst])
667                     device_category = helpers.categorize_device_by_dst(
                            pkt_size, device_category)
668
669                     # if the device is an outlet,
670                     # then use outlet event criteria
671                     if device_category == 'OUTLET':
672                         event_identification, events = helpers.
                                id_events_by_dst(pkt_time, pkt_size, pkt_src,
                                 pkt_dst, event_identification, events)
673             device_categorization[device] = device_category
674
675     return device_categorization, event_identification, pkt_cntr
676
677
678 def events_by_src(network_edge_file, device_categorization,
        event_identification, total_time_class, total_time_map):
679     """
680     Identify events by source.
681
682     Uses the files created in preprocessing which contain packets with
683     the device as the source to identify events according to
684     classifier parameters. Simultaneously records the amount of data
685     sent between two devices.
686
687     Parameters
688     ----------
689     network_edge_file: (file object) File used to record amount of
690         data sent between two devices.
691     device_categorization: (dictionary) Uses device addresses as keys
692         and assigned category as values.
693     event_identification: (list) 2-D list containing the time, source,
```

```
694         and destination of each identified event
695     total_time_class: (time) total classifier time
696     total_time_map: (time) total mapper time
697 
698     Returns
699     ----------
700     event_identification: (list) 2-D list containing the time, source,
701         and destination of each identified event
702     pkt_cntr: (int) number of packets processed by events by source
703     total_time_class: (time) total classifier time
704     total_time_map: (time) total mapper time
705     """
706 
707     pkt_cntr = 0
708     # analyze packets sent from a device using src csv files
709     # from preprocessing that contain packets from a particular device
710     for filename in os.listdir(SRC_DIR):
711         device_by_src = {}
712         device = filename.replace('.csv', '').replace('.', ':')
713         pkt_src = ''
714 
715         # initialize dictionary for mapping devices; key is the dst
716         # device and the value is the total size of packets sent to
717         # that device
718         map_dst_device = {}
719 
720         with open(SRC_DIR+filename, 'rb') as curr_file:
721             reader = csv.reader(curr_file)
722             contents = list(reader)
723 
724             # obtain the time, frame size, src, dst for each packet
725             # discard seconds
726             for line in contents:
727                 pkt_cntr += 1
728                 pkt_time = line[0]
729                 pkt_time = pkt_time[:pkt_time.rindex(':')]
730                 pkt_size = int(line[1])
731                 pkt_src = line[2]
732                 pkt_dst = line[3]
733 
734                 start_time = time.time()
735                 # classification
736                 if pkt_dst == ROUTER:
```

```
737                     # sum packet size of pkts sent in same min
738                     if pkt_time in device_by_src:
739                         device_by_src[pkt_time] = device_by_src[pkt_time
                                ] + pkt_size
740                     else:
741                         device_by_src[pkt_time] = pkt_size
742                         total_time_class += time.time() - start_time
743 
744                 start_time = time.time()
745                 # mapping
746                 # find the total frame size of all pkts
747                 # sent to a device
748                 if pkt_dst in map_dst_device:
749                     map_dst_device[pkt_dst] = map_dst_device[pkt_dst] +
                            pkt_size
750                 else:
751                     map_dst_device[pkt_dst] = pkt_size
752                 total_time_map += time.time() - start_time
753 
754             start_time = time.time()
755             # attempt to identify events based off of packets
756             # sent from a device
757             if (device in IOT_DEVICES) and (len(device_by_src) != 0):
758                 event_identification = helpers.id_events_by_src(
                        device_by_src, pkt_src, ROUTER,
                        device_categorization, event_identification)
759             total_time_class += time.time() - start_time
760 
761         start_time = time.time()
762         # write the src, dst, and total frame size to the
763         # network edge csv file
764         # The R-Script requires IDs instead of names, so store with ID
765         with open(network_edge_file, 'a') as network_file:
766             for k, v in map_dst_device.iteritems():
767                 csv.writer(network_file).writerow([DEVICE_ID[device.
                        replace('.', ':')], DEVICE_ID[k], v])
768         total_time_map += time.time() - start_time
769 
770     return event_identification, pkt_cntr, total_time_class,
            total_time_map
771 
772 # call main
773 if __name__ == "__main__":
```

```
774     main(sys.argv[1:])
```

Appendix C. BLE Script

```
#!/usr/bin/python
# ble.py
# Script that includes the BLE classifier component of CITIoT.
# Includes timing information.
# Required: helpers.py
import helpers
import pyshark
import sys, getopt, csv, time, os

#########################################################################
# GLOBAL VARIABLES
ADV_IND = '0'
SCAN_RESP = '4'
CONNECT_REQ = '5'
TIMING_PKT_NUMBER = 25000
path_name = os.getcwd()
DATE = path_name[path_name.rindex('/')+1:]
BLE_DEVICES = {'Eve Door 91B3', 'Eve Room 4A04', 'Eve Weather 943D'
               'Eve Motion 31A7', 'Eve Energy 556E', 'Gunbox',
               'BLELock', '00000b67', 'Instant Pot Smart',
               'PLAYBULB',}
#########################################################################

def main(argv):
    try:
        opts, args = getopt.getopt(argv, "hc:", ["help"])
    except getopt.GetoptError:
        print 'parse_controller.py -c <pcap file>'
        sys.exit(2)
    for opt, arg, in opts:
        if opt in ("-h", "--help"):
            print 'parse_controller.py -c <pcap file>'
            sys.exit()
        elif opt == '-c':
            classifier(arg)


def classifier(file_name):
    """
    Classifying unit for Bluetooth Low Energy.

    Utilizes the provided packet capture to parse data looking for
```

```
    connection events.
    Provides two files: one with each device's MAC address and
    name and one with the time, access address, master, and slave
    of each connection event.

    Parameters
    ----------
    file_name: (string) File name of capture provided by user.

    Returns
    ----------

    """
    proc_time = "ble_processing_time_" + DATE + ".csv"
    helpers.delete_file(proc_time)
    pt_file = open(proc_time, 'w')
    csv.writer(pt_file).writerow(["Unit", "Total Packets Processed", "
        Total Process Time", "Average Process Time"])
    pkt_cntr = 0

    total_time_start = time.time()
    # initialize files
    cap = pyshark.FileCapture(file_name)
    btle_output_file = "ble_events_" + DATE + ".csv"
    helpers.delete_file(btle_output_file)
    btle_device_file = "ble_devices_" + DATE + ".csv"
    helpers.delete_file(btle_device_file)

    # initialize dictionary to store device information
    devices = {}
    # initialize list to store connection information
    connections = []

    # parse each packet in the capture looking for advertising
    # indication, scan response and connect request packets
    for pkt in cap:
        pkt_cntr += 1
        try:
            if pkt.btle.advertising_header_pdu_type == ADV_IND:
                devices = ble_adv_ind(pkt, devices)

            elif pkt.btle.advertising_header_pdu_type == SCAN_RESP:
                devices = ble_scan_response(pkt, devices)
```

```
85
86              elif pkt.btle.advertising_header_pdu_type == CONNECT_REQ:
87                  connections = ble_connect_req(pkt, devices, connections
                        )
88
89          except AttributeError:
90              # ignore packets that are malformed
91              pass
92
93      # write results to files
94      with open(btle_output_file, 'a') as curr_file:
95          for file_input in connections:
96              csv.writer(curr_file).writerow(file_input)
97
98      with open(btle_device_file, 'a') as curr_file:
99          for k, v in devices.iteritems():
100             csv.writer(curr_file).writerow([k, v])
101
102     final_time = time.time()
103     total_time = final_time - total_time_start
104     normalized_total_time = (TIMING_PKT_NUMBER * total_time) /
            pkt_cntr
105
106     print "total number of packets: ", pkt_cntr
107     print "Total time: ", total_time
108     print "Normalized total time per 25000 packets: ",
            normalized_total_time
109
110     csv.writer(pt_file).writerow(["BLE", pkt_cntr, total_time,
            normalized_total_time])
111     csv.writer(pt_file).writerow(["Start and finish time",
            total_time_start, final_time, final_time - total_time_start])
112     pt_file.close()
113
114
115 def ble_adv_ind(pkt, devices):
116     """
117     Parse Advertising Indication packets.
118
119     Extract device name provided in Advertising Indication
120     packets.
121
122     Parameters
```

```
123     ----------
124     pkt: (pyshark packet) Pyshark packet object containing
125         packet information from capture.
126     devices: (dictionary) Uses the device address as the key
127         and the name of the device as the value.
128 
129     Returns
130     ----------
131     devices: (dictionary) Updated dictionary of device addresses and
132         corresponding names
133     """
134     # if a device name has not been found, then extract the name from
135     # the Advertising Indication packet and store it
136 
137     adv_addr = pkt.btle.advertising_address
138     if adv_addr not in devices:
139         device_name = pkt.btle.btcommon_eir_ad_entry_device_name
140         # from trial and error some weird device names can appear,
141         # so ignore some that have been encountered
142         if (len(device_name) > 2) and (device_name in BLE_DEVICES):
143             devices[adv_addr] = device_name
144 
145 
146     return devices
147 
148 
149 def ble_scan_response(pkt, devices):
150     """
151     Parse Scan Response packets.
152 
153     Extract device name provided in Scan Response packets. These
154     packet types provide more information than an Advertising
155     Packet, so overwrite if a device name was found from an
156     Advertising Indication packets.
157 
158     Parameters
159     ----------
160     pkt: (pyshark packet) Pyshark packet object containing packet
161         information from capture.
162     devices: (dictionary) Uses the device address as the key and
163         the name of the device as the value.
164 
165     Returns
```

```
    ----------
    devices: (dictionary) Updated dictionary of device addresses
        and corresponding names
    """
    # extract a device name from the Scan Response packet
    # Scan Response packets provide better naming information so
    # overwrite previously found names
    try:
        device_name = pkt.btle.btcommon_eir_ad_entry_device_name
        if device_name in BLE_DEVICES:
            devices[pkt.btle.advertising_address] = device_name
    except AttributeError:
        # ignore packets with no device name
        pass

    return devices


def ble_connect_req(pkt, devices, connections):
    """
    Parse Connection Request packets.

    Find Connection Request packets and record the time, access
    address, master device name, and slave device name.

    Parameters
    ----------
    pkt: (pyshark packet) Pyshark packet object containing packet
        information from capture.
    devices: (dictionary) Uses the device address as the key and
        the name of the device as the value.
    connections: (list) 2-D list containing the following
        information for each connection event: packet time,
        access address, master device name, and slave device name.

    Returns
    ----------
    connections: (list) Updated list of connection events
    """
    # extract required information from Connection Request
    # packets (packet time, master/slave mac addresses, and
    # access address)
    pkt_time = pkt.frame_info.time_epoch
```

```
209     pkt_time = helpers.pretty_time(pkt_time)
210     pkt_time = pkt_time[:pkt_time.rindex(':')]
211     pkt_mstr_mac = pkt.btle.initiator_address
212     pkt_slv_mac = pkt.btle.advertising_address
213     pkt_lladdr = pkt.btle.link_layer_data_access_address
214     pkt_slv_id = "No ID"
215 
216     # obtain device names using MAC addresses found in the
217     # Connection Request packet and device names found in
218     # Advertising Indication and Scan Response packets
219     if pkt_slv_mac in devices: pkt_slv_id = devices[pkt_slv_mac]
220 
221     fields = [pkt_time, pkt_slv_id, '1']
222 
223     # if the slave address cannot be resolved to a device name,
224     # ignore it as it does not provide valuable information
225     # also, sometimes multiple connections occur during one event
226     # so only acknowledge one event per device per minute.
227     # This sample interval provides enough precision for the
228     # problem at hand
229     if (pkt_slv_id in BLE_DEVICES) and (fields not in connections):
230         connections.append(fields)
231 
232     return connections
233 
234 
235 # call main
236 if __name__ == "__main__":
237     main(sys.argv[1:])
```

Appendix D. Helper Script

```
#!/usr/bin/python
# helpers.py
# provides a collection of functions used by wifi.py and ble.py
# includes the device categorization and event identification
# criteria, graphing capabilities, and file control
import pyshark
import datetime, os, errno, shutil, sys
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

######################################################################
# GLOBAL VARIABLES
DEVICE_NAME = {'ec:4f:82:73:d1:1c': 'Router',
               'ec:1a:59:e4:fd:41': 'NetCam',
               'b4:75:0e:0d:33:d5': 'Switch1',
               'b4:75:0e:0d:94:65': 'Switch2',
               '94:10:3e:2b:7a:55': 'Switch3',
               '14:91:82:c8:6a:09': 'Switch4',
               'ec:1a:59:f1:fb:21': 'Motion',
               '14:91:82:24:dd:35': 'Insight',
               '60:38:e0:ee:7c:e5': 'Mini',
               'b8:27:eb:09:1a:81': 'Pi',
               'a0:18:28:33:34:f8': 'iPhone',
               '08:66:98:ed:1e:19': 'AppleTV'}

SRC_DIR = './Source/'
DST_DIR = './Destination/'
GRPH_DIR = './Graphs/'
TIME_FORMAT = '%Y-%m-%d %H:%M:%S'
UNKNOWN = 0
OUTLET = 1
SENSOR = 2
CAMERA = 3
######################################################################


def init_dirs():
    """
    Initializes directory for storing files. If the directory exists
    delete it then create an empty dir.
    """
    src_dir = os.path.dirname(SRC_DIR)
```

```
43      dst_dir = os.path.dirname(DST_DIR)
44
45      if os.path.exists(src_dir):
46          try:
47              shutil.rmtree(src_dir)
48          except OSError:
49              print "Issue removing files within ", src_dir, " check if
                    files are read only."
50              sys.exit()
51      os.makedirs(src_dir)
52      if os.path.exists(dst_dir):
53          try:
54              shutil.rmtree(dst_dir)
55          except OSError:
56              print "Issue removing files within ", dst_dir, " check if
                    files are read only."
57              sys.exit()
58      os.makedirs(dst_dir)
59
60
61  def init_training_dirs():
62      """
63      Initializes directory for graph files. If the directory exists
64      delete it then create an empty dir.
65      """
66      grph_dir = os.path.dirname(GRPH_DIR)
67      if os.path.exists(grph_dir):
68          try:
69              shutil.rmtree(grph_dir)
70          except OSError:
71              print "Issue removing files within ", grph_dir, " check if
                    files are read only."
72              sys.exit()
73      os.makedirs(grph_dir)
74
75
76  def delete_file(filename):
77      """
78      Deletes file.
79
80      Parameters
81      ----------
82      filename: (string) file to delete.
```

```
83      """
84      try:
85          os.remove(filename)
86      except OSError as e:
87          if e.errno != errno.ENOENT:
88              raise
89
90
91  def pretty_time(pkt_time):
92      """
93      Takes epoch time and transforms it into a better format.
94
95      Parameters
96      ----------
97      pkt_time: (pkt.frame_info.time_epoch) Time of packet in
98          epoch format.
99
100     Returns
101     ----------
102     time: (datetime) Time of packet in datetime format.
103     """
104     # have to account for timestamp issue on host computer during
            first four days of trial
105     if float(pkt_time) < 1503370800:
106         return datetime.datetime.fromtimestamp(float(pkt_time) + 42).
                strftime(TIME_FORMAT)
107     elif float(pkt_time) < 1503702000:
108         return datetime.datetime.fromtimestamp(float(pkt_time)+48).
                strftime(TIME_FORMAT)
109     else:
110         return datetime.datetime.fromtimestamp(float(pkt_time)).
                strftime(TIME_FORMAT)
111
112
113 class Graph(object):
114     """
115     Helper function to create graphs using the matplotlib
116     library
117     """
118     def __init__(self, plots= [], names= [], date_format = '%m-%d %H:%
            M:%S',
119                  title= "Figure"):
120         # get the datetime and value for each device and packet
```

```
121         self.plots = plots
122         # get the name for each device
123         self.names = names
124         self.title = title
125         # self.id = id
126         self.date_format = date_format
127         # create figure and axes
128         self.fig = plt.figure()
129         self.ax = plt.subplot(111)
130 
131         # setup formatting for datetime axes
132         self.seconds = mdates.SecondLocator()
133         self.hours = mdates.HourLocator()
134         self.minutes = mdates.MinuteLocator()
135         self.hourFmt = mdates.DateFormatter('%H')
136 
137         # create the initial graph
138         self.curr_pos = 0
139 
140         # call self.update everytime a 'key_press_event' happens
141         self.cid = self.fig.canvas.mpl_connect('key_press_event', self.
                update)
142 
143     def graph(self):
144         """
145         Presents graph of device packets
146         """
147         self.set_axes_parameters()
148 
149         # set the plot data and x-axis range
150         self.ax.plot_date(self.dates, self.values)
151 
152         plt.show()
153 
154     def update(self, e):
155         """
156         Updates graphs when moving between the
157         different devices
158         e: (event) function called on arrow key event
159         """
160         if e.key == "right":
161             self.curr_pos += 1
162         elif e.key == "left":
```

```
163             self.curr_pos -= 1
164         else:
165             return
166 
167         # allow it to loop
168         self.curr_pos = self.curr_pos %len(self.plots)
169 
170         self.ax.cla()
171 
172         self.set_axes_parameters()
173 
174         self.ax.plot_date(self.dates, self.values)
175 
176         # regraph
177         self.fig.canvas.draw()
178 
179     def save_files(self):
180         """
181         Saves graphs into files
182         """
183         self.curr_pos = 0
184         for name in self.names:
185             self.fig = plt.figure()
186             self.ax = plt.subplot(111)
187 
188             self.set_axes_parameters()
189 
190             self.ax.plot_date(self.dates, self.values)
191 
192             self.fig.savefig(GRPH_DIR + name + '.png')
193             self.ax.cla()
194             self.curr_pos += 1
195 
196     def set_axes_parameters(self):
197         """
198         Setup the axes parameters
199         """
200         # update date, values, and name to the current device
201         self.dates = self.plots[self.curr_pos][0]
202         self.values = self.plots[self.curr_pos][1]
203 
204         # set the title of the figure and axes
205         self.fig.canvas.set_window_title(self.title)
```

```
206         name = self.names[self.curr_pos]
207         self.ax.set_title(name)
208         self.ax.set_xlabel('Time (hour)')
209         self.ax.set_ylabel('Frame Size (bytes)')
210 
211         # setup formatting for datetime axes
212         self.ax.xaxis.set_major_locator(self.hours)
213         self.ax.xaxis.set_major_formatter(self.hourFmt)
214         self.ax.format_xdata = mdates.DateFormatter(self.date_format)
215         self.ax.set_xlim(self.dates.min()-.001, self.dates.max()+.001)
216         self.ax.grid(True)
217         self.fig.autofmt_xdate()
218 
219     def delete(self):
220         plt.close('all')
221         self.fig.canvas.mpl_disconnect(self.cid)
222 
223 
224 # write category of device to csv
225 def categorize_device_by_dst(pkt_size, device_category):
226     """
227     Categorize devices using traffic destined to device and criteria
228     found during training.
229 
230     Parameters
231     ----------
232     pkt_size: (pkt.length) Contains the frame size of a packet sent
233         to the device.
234 
235     Returns
236     ----------
237     category: (string) Return the category of the device based on
238         which criteria is met by the packet.
239     """
240     if device_category != 'OUTLET':
241         # criteria for an outlet
242         if 619 <= pkt_size <= 632:
243             return 'OUTLET'
244         # criteria for a camera
245         elif pkt_size == 281:
246             return 'CAMERA'
247         # criteria for a sensor
248         elif pkt_size == 269:
```

```
249             return 'SENSOR'
250
251     return device_category
252
253
254 def id_events_by_dst(pkt_time, pkt_size, pkt_src, pkt_dst,
        event_identification, events):
255     """
256     Identify events using traffic from the Raspberry Pi to device and
257     criteria found during training.
258
259     Parameters
260     ----------
261     pkt_time: (datetime) Timestamp of packet.
262     pkt_size: (pkt.length) Uses device address as keys and the
263         assigned category as values.
264     pkt_src: (pkt.wlan.sa) Source address of the packet.
265     pkt_dst: (pkt.wlan.da) Destination address of the packet.
266     event_identification: (list) 2-D list with each entry containing
267         the time, source, and destination of an event.
268     events: (list) Contains a time to the minute of events that
269         occurred.
270
271     Returns
272     ----------
273     event_identification: (list) Updated event identification list.
274     """
275     if 619 <= pkt_size <= 632:
276         pkt_time = pkt_time[:pkt_time.rindex(':')]
277         # sometimes multiple packets are sent due to retransmission
278         # attempts so only acknowledge one event per device per
279         # minute. This sample interval provides enough precision
280         # for the problem at hand
281         if pkt_time not in events:
282             events.append(pkt_time)
283             event_identification.append([pkt_time, DEVICE_NAME[pkt_dst
                ], '1'])
284
285     return event_identification, events
286
287
288 def id_events_by_src(device_by_src, pkt_src, pkt_dst,
        device_categorization, event_identification):
```

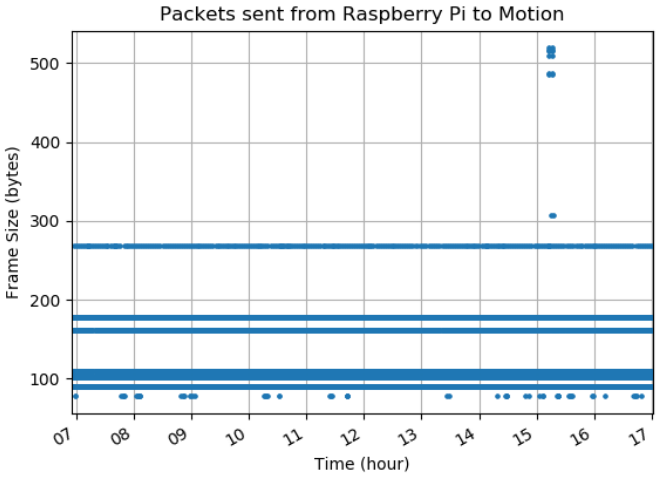
```
289     """
290     Identify events using traffic from a device to the router and
291     criteria found during training.
292 
293     Parameters
294     ----------
295     device_by_src: (dictionary) Keys are the time of the packet time
296         to the minute and values are the sum of all packets sent in
297         one minute.
298     pkt_src: (pkt.wlan.sa) Source address of the packet.
299     pkt_dst: (pkt.wlan.da) Destination address of the packet.
300     device_categorization: (dictionary) Uses device addresses as keys
301         and assigned category as values.
302     event_identification: (list) 2-D list containing the time, source,
303         and destination of each identified event.
304 
305     Returns
306     ----------
307     event_identification: (list) Updated 2-D list containing the time,
308         source, and destination of each identified event.
309     """
310     if device_categorization[pkt_src] == 'CAMERA':
311         for k, v in sorted(device_by_src.iteritems()):
312             if v > 100000:
313                 if check_motion_event(k, pkt_src, event_identification)
                        :
314                     event_identification.append([k, DEVICE_NAME[pkt_src
                        ], '1'])
315     elif device_categorization[pkt_src] == 'SENSOR':
316         for k, v in sorted(device_by_src.iteritems()):
317             if v > 10000:
318                 if check_motion_event(k, pkt_src, event_identification)
                        :
319                     event_identification.append([k, DEVICE_NAME[pkt_src
                        ], '1'])
320 
321     return event_identification
322 
323 
324 def check_motion_event(pkt_time, pkt_src, event_identification):
325     """
326     Because motion events take time to send, if one was sent the
327     minute before then ignore a potential new event could cause
```

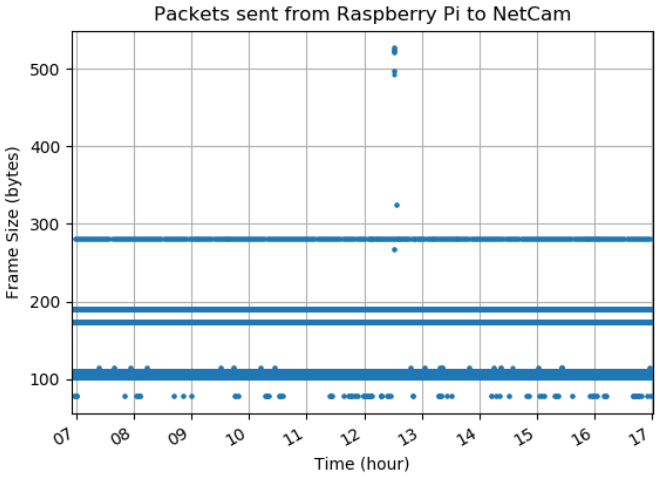
```
328     it to miss two events in a row, but most motion devices
329     have at least a 60 second no motion sensor before restarting,
330     so this should not be an issue.
331
332     Parameters
333     ----------
334     pkt_time: (datetime) Timestamp of packet.
335     pkt_src: (pkt.wlan.sa) Source address of the packet.
336     event_identification: (list) 2-D list with each entry containing
337         the time, source, and destination of an event.
338     """
339     date = datetime.datetime.strptime(pkt_time, '%Y-%m-%d %H:%M')
340     check_date = date - datetime.timedelta(minutes=1)
341     new_date = check_date.strftime('%Y-%m-%d %H:%M')
342     if [new_date, DEVICE_NAME[pkt_src], '1'] in event_identification:
343         return False
344     return True
```

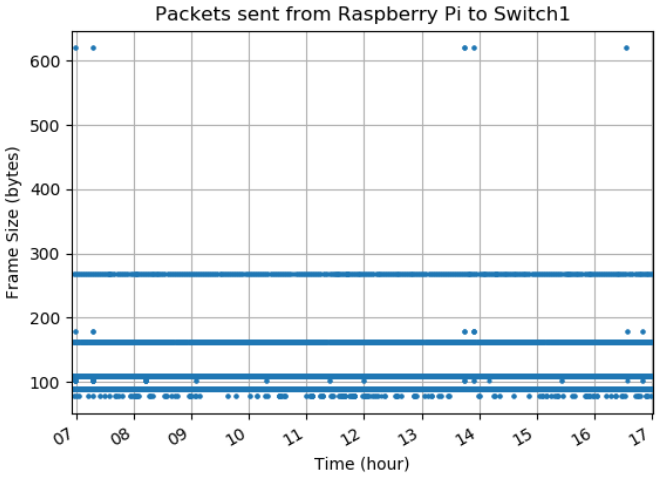
Appendix E. Training Event Log

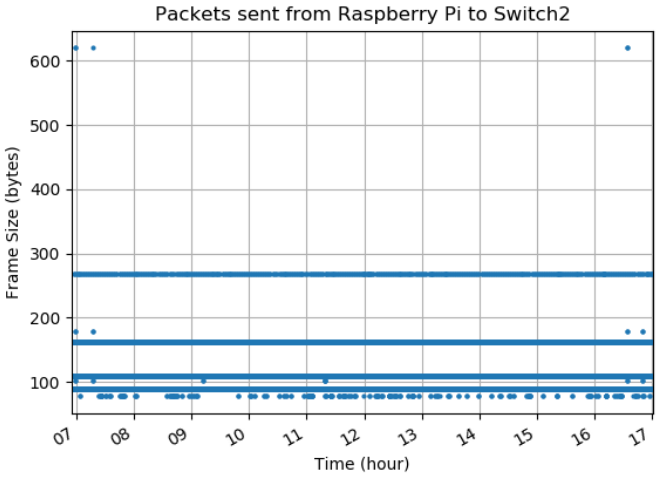
Time	MAC	Device	Action
8/14/17 6:55		Homebridge	Start
8/14/17 6:57		Alfa Card	Start
8/14/17 6:57	B4:75:0E:0D:33:D5	Switch 1	ON
8/14/17 6:58	B4:75:0E:0D:94:65	Switch 2	ON
8/14/17 6:59	94:10:3E:2B:7A:55	Switch 3	ON
8/14/17 7:00	14:91:82:C8:6A:09	Switch 4	ON
8/14/17 7:01	60:38:E0:EE:7C:E5	Mini	ON
8/14/17 7:09	14:91:82:24:DD:35	Insight	ON
8/14/17 7:11	EC:1A:59:E4:FD:41	NetCam	MOTION
8/14/17 7:16	EC:1A:59:F1:FB:21	Motion	MOTION
8/14/17 7:16	B4:75:0E:0D:33:D5	Switch 1	OFF
8/14/17 7:17	B4:75:0E:0D:94:65	Switch 2	OFF
8/14/17 7:17	94:10:3E:2B:7A:55	Switch 3	OFF
8/14/17 7:17	14:91:82:C8:6A:09	Switch 4	OFF
8/14/17 7:17	60:38:E0:EE:7C:E5	Mini	OFF
8/14/17 7:17	14:91:82:24:DD:35	Insight	OFF
8/14/17 13:43	B4:75:0E:0D:33:D5	Switch 1	ON
8/14/17 13:45	B4:75:0E:0D:33:D5	Switch 1	OFF
8/14/17 16:33	B4:75:0E:0D:33:D5	Switch 1	ON
8/14/17 16:34	B4:75:0E:0D:94:65	Switch 2	ON
8/14/17 16:35	94:10:3E:2B:7A:55	Switch 3	ON
8/14/17 16:36	14:91:82:C8:6A:09	Switch 4	ON
8/14/17 16:37	60:38:E0:EE:7C:E5	Mini	ON
8/14/17 16:43	14:91:82:24:DD:35	Insight	ON
8/14/17 16:45	EC:1A:59:E4:FD:41	NetCam	MOTION
8/14/17 16:48	EC:1A:59:F1:FB:21	Motion	MOTION
8/14/17 16:49	B4:75:0E:0D:33:D5	Switch 1	OFF
8/14/17 16:49	B4:75:0E:0D:94:65	Switch 2	OFF
8/14/17 16:49	94:10:3E:2B:7A:55	Switch 3	OFF
8/14/17 16:49	14:91:82:C8:6A:09	Switch 4	OFF
8/14/17 16:49	60:38:E0:EE:7C:E5	Mini	OFF
8/14/17 16:49	14:91:82:24:DD:35	Insight	OFF
8/14/17 16:59		Alfa Card	Stop
8/14/17 17:00		Homebridge	Stop

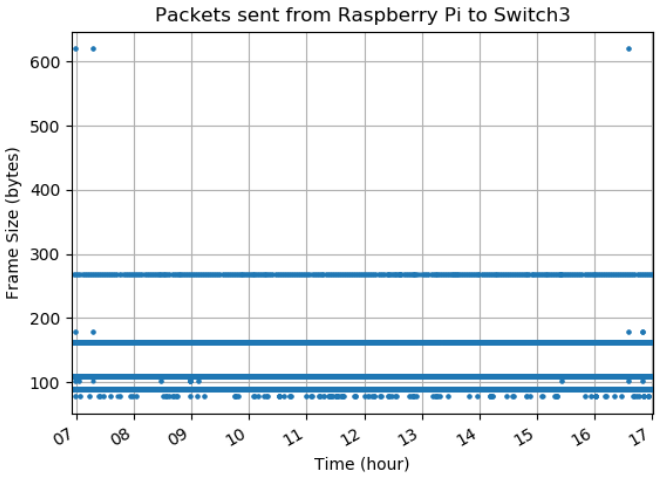
Appendix F. Training Plots from Raspberry Pi to Device

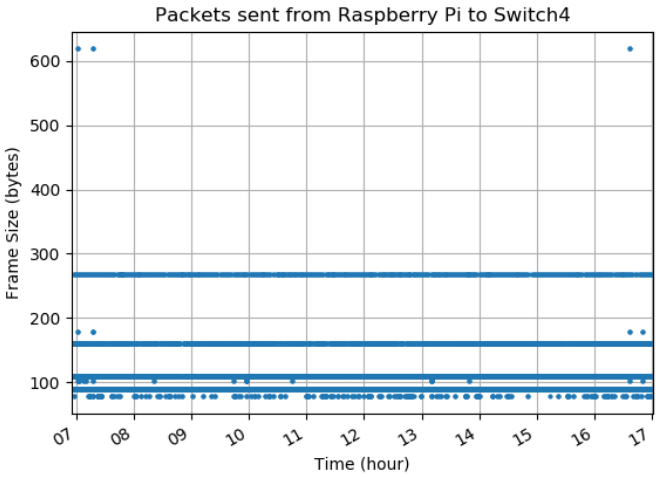


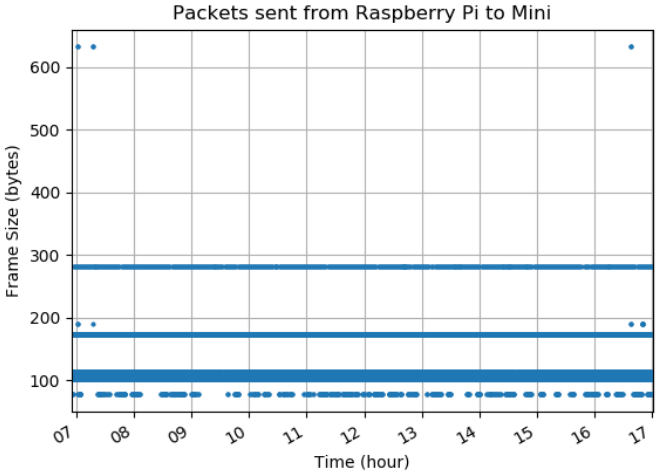


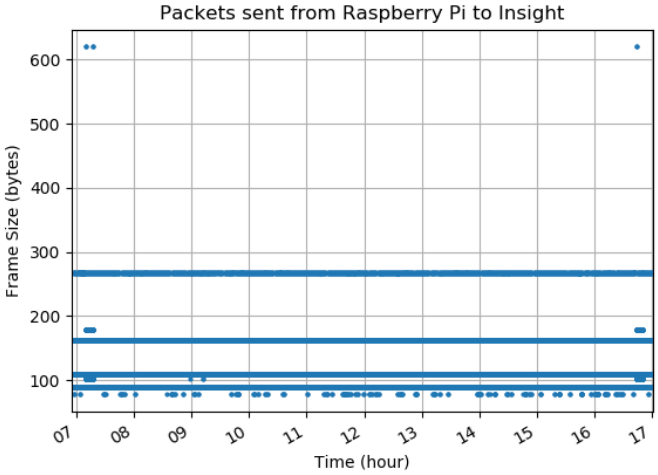




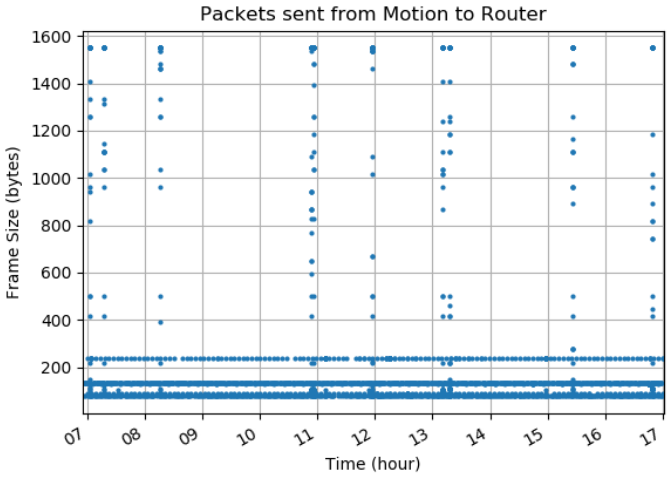

Packets sent from Raspberry Pi to Switch3
Frame Size (bytes)
600
500
400
300
200
100
07
08
09
10
11
12
13
14
15
16
17
Time (hour)

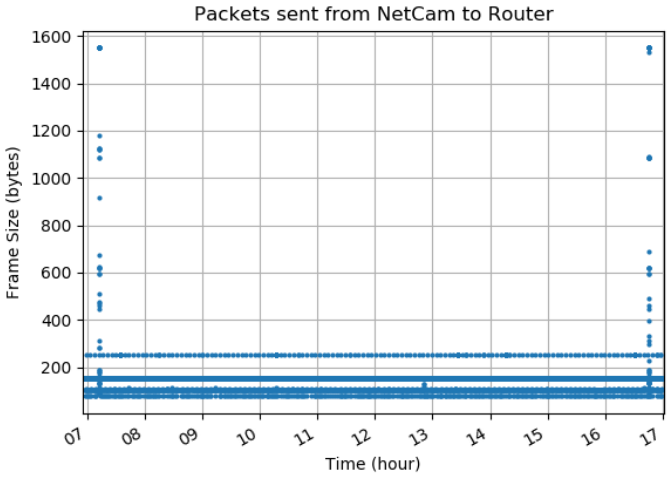

Packets sent from Raspberry Pi to Switch4
Frame Size (bytes)
600
500
400
300
200
100
07
08
09
10
11
12
13
14
15
16
17
Time (hour)

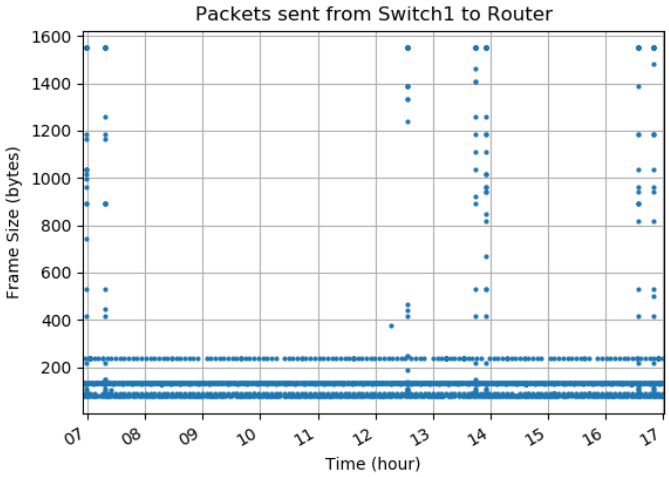

Packets sent from Raspberry Pi to Mini
Frame Size (bytes)
600
500
400
300
200
100
07
08
09
10
11
12
13
14
15
16
17
Time (hour)

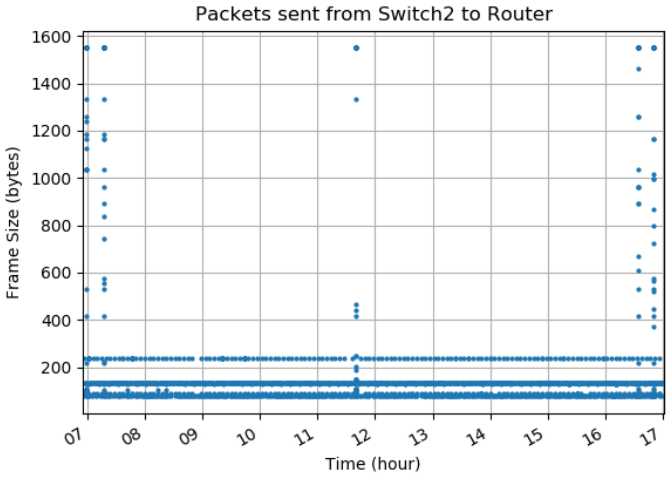

Packets sent from Raspberry Pi to Insight
Frame Size (bytes)
600
500
400
300
200
100
07
08
09
10
11
12
13
14
15
16
17
Time (hour)

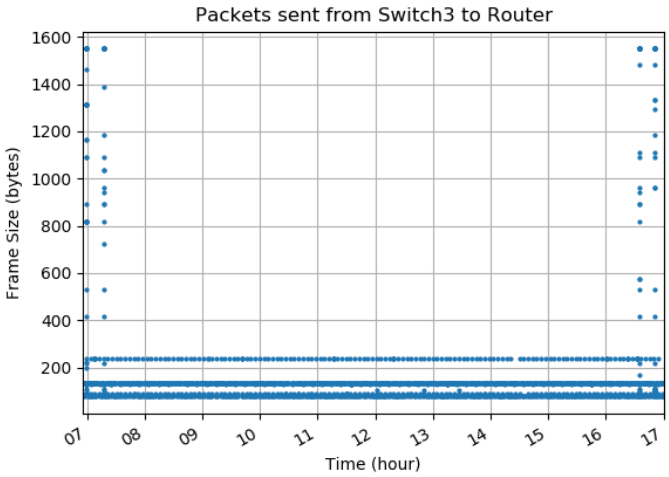
Appendix G. Training Plots from Device to Router

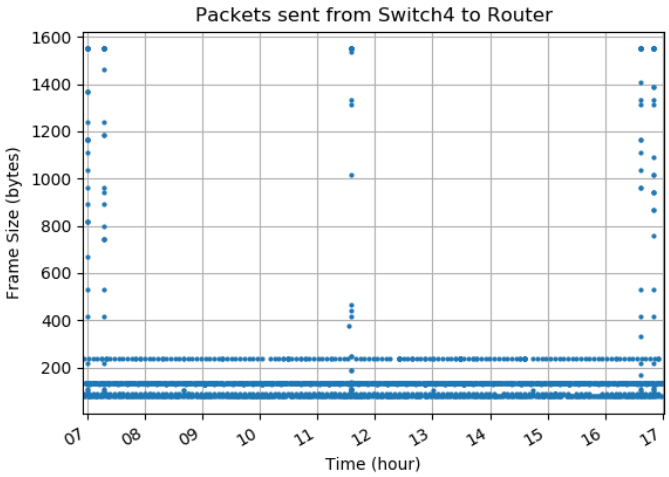


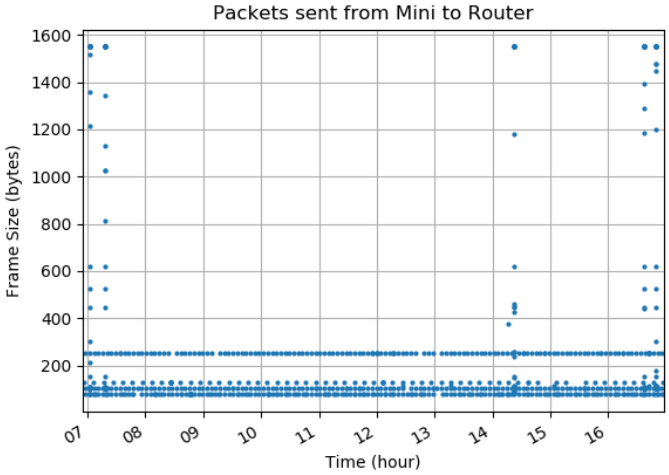


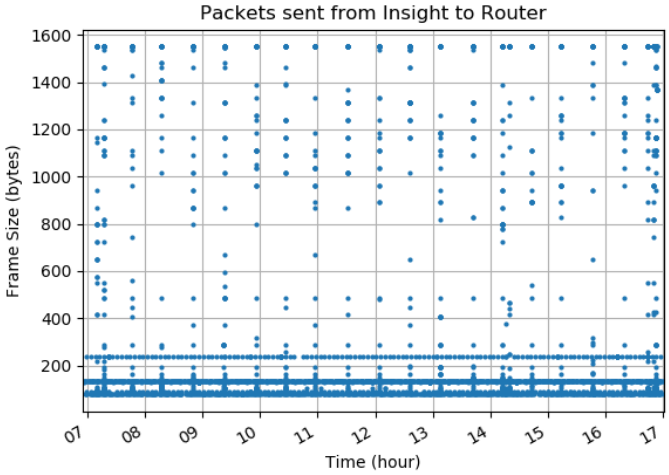





Packets sent from Switch3 to Router
Frame Size (bytes)
Time (hour)


Packets sent from Switch4 to Router
Frame Size (bytes)
Time (hour)


Packets sent from Mini to Router
Frame Size (bytes)
Time (hour)


Packets sent from Insight to Router
Frame Size (bytes)
Time (hour)

Appendix H. Network Mapping Script

```
library('igraph')

# -- CREATE GRAPHS ------------------------------------------------
nodes <- read.csv("network_node_file.csv", header=T, as.is=T)
links <- read.csv("network_edge_file.csv", header=T, as.is=T)
net <-graph_from_data_frame(d=links, vertices=nodes, directed=T)
colrs <- c("steelblue2", "tomato", "gold", "green")
V(net)$color <- colrs[V(net)$device.type]
V(net)
E(net)$width <- E(net)$weight/4000000
l <- layout_in_circle(net)
plot(net, layout=l, vertex.label=V(net)$device.label,
     edge.arrow.size=0, vertex.frame.color="#ffffff",
     vertex.label.color="black")
```

Appendix I. Device Shadow Script

```
#!/usr/bin/python
# deviceShadow.py
# Uses Scapy to spoof packets sent from the Raspberry Pi to the
# outlets and from the camera/sensor to the router
# include randomization to emulate real traffic
import sys, time, datetime, multiprocessing
from scapy.all import *

#####################################################################
# GLOBAL VARIABLES
DEVICES = {'B4:75:0E:0D:33:D5': '192.168.1.40',
           'B4:75:0E:0D:94:65': '192.168.1.41',
           '94:10:3E:2B:7A:55': '192.168.1.42',
           '14:91:82:C8:6A:09': '192.168.1.7',
           '14:91:82:24:DD:35': '192.168.1.47',
           '60:38:E0:EE:7C:E5': '192.168.1.51',
           'EC:1A:59:F1:FB:21': '192.168.1.43',
           'EC:1A:59:E4:FD:41': '192.168.1.44'}
RASPI_MAC = 'B8:27:EB:09:1A:81'
RASPI_IP = '192.168.1.50'
ROUTER_MAC = 'ec:4f:82:73:d1:1c'
ROUTER_IP = '192.168.1.1'
# tcp destination port
tcp = TCP(dport=4242)
OUTLET_DATA = '\xccWhat is the answer to the Ultimate Question of' \
              ' Life, The Universe, and Everything? The answer is' \
              ' 42!' + ('a' * 431)
SENSOR_DATA = '\xccWhat is the answer to the Ultimate Question of' \
              ' Life, The Universe, and Everything? The answer is' \
              ' 42!' + ('a' * 1351)
CAMERA_DATA = '\xccWhat is the answer to the Ultimate Question of' \
              ' Life, The Universe, and Everything? The answer is' \
              ' 42!' + ('a' * 1351)
TIME_FORMAT = '%Y-%m-%d %H:%M:%S'
#####################################################################


def main():
    """
    Randomly spoof outlet, sensor, and camera traffic at a random
    interval between one and two minutes. Each cycle of all three
    types of traffic are sent randomly every 10 to 15 minutes
```

```
43      """
44      functions = [outlet_devices, sensor_devices, camera_devices]
45      while 1:
46          random.shuffle(functions)
47          for function in functions:
48              function()
49              sleep_time = random.randint(60,120)
50              print 'sleeping for ', sleep_time, ' seconds.'
51              time.sleep(sleep_time)
52          sleep_time = random.randint(600,900)
53          print 'sleeping for ', sleep_time, ' seconds.'
54          time.sleep(sleep_time)
55
56
57  def outlet_devices():
58      """
59      Spoof the traffic signature of packets sent from the Raspberry
60      Pi to an outlet. For each device in a random order send a 620
61      byte TCP packet from the raspberry pi to each device
62      """
63      print 'outlet device'
64      outlets = list(DEVICES.keys())
65      random.shuffle(outlets)
66      for outlet in outlets:
67          for _ in range(3):
68              print "sending packet on behalf of ", RASPI_MAC, ' to ',
                    outlet, ' at ', pretty_time(time.time())
69              ether = Ether(src=RASPI_MAC, dst=outlet)
70              ip = IP(src=RASPI_IP, dst=DEVICES[outlet])
71              sendp(ether / ip / tcp / OUTLET_DATA, iface='eth0', verbose
                    =0)
72          time.sleep(1)
73
74
75  def sensor_devices():
76      """
77      Spoof the traffic signature of packets sent from a sensor to
78      the router. For each device in a random order send a series of
79      TCP packets totaling a size of at least 10000 bytes from the
80      device to the router in one minute
81      """
82      print 'sensor device'
83      sensors = list(DEVICES.keys())
```

```
84      random.shuffle(sensors)
85      for sensor in sensors:
86          for _ in range(10):
87              print "sending packet on behalf of ", sensor, ' to ',
                    ROUTER_MAC, ' at ', pretty_time(time.time())
88              ether = Ether(src=sensor, dst=ROUTER_MAC)
89              ip = IP(src=DEVICES[sensor], dst=ROUTER_IP)
90              sendp(ether / ip / tcp / SENSOR_DATA, iface='wlan0',
                    verbose=0)
91          time.sleep(1)
92
93
94  def camera_devices():
95      """
96      Spoof the traffic signature of packets sent from a camera to
97      the router. For each device in a random order send a series of
98      TCP packets totaling a size of at least 100000 bytes from
99      the device to the router in one minute
100     """
101     print 'camera device'
102     cameras = list(DEVICES.keys())
103     random.shuffle(cameras)
104     # spawn 5 subprocesses with eac sending 5 pkts
105     jobs = []
106     for camera in cameras:
107         for _ in range(5):
108             p = multiprocessing.Process(target=camera_msg, args =(
                    camera,))
109             jobs.append(p)
110             p.start()
111     # block until all processes are done
112     for job in jobs:
113         job.join()
114
115
116 def camera_msg(camera):
117     """
118     Process to send five packets on behalf of the camera
119     """
120     for _ in range(5):
121         print "sending packet on behalf of ", camera, ' to ',
                ROUTER_MAC, ' at ', pretty_time(time.time())
122         ether = Ether(src=camera, dst=ROUTER_MAC)
```

```
123         ip = IP(src=DEVICES[camera], dst=ROUTER_IP)
124         sendp(ether / ip / tcp / CAMERA_DATA, iface='wlan0', verbose=0)
125
126
127 def pretty_time(pkt_time):
128     """
129     Helper function to get better time format
130     """
131     return datetime.fromtimestamp(float(pkt_time)).strftime(
            TIME_FORMAT)
132
133 if __name__ == "__main__":
134     main()
```

Appendix J. MAC Shadow Script

```
#!/usr/bin/python
# macShadow.py
# Uses Scapy to spoof packets sent from user devices to make it
# appear that the device is within the home. Does not spoof the
# device if it is actually home to ensure that ARP tables are not
# changed
import sys, time, datetime
from scapy.all import *

#########################################################################
# GLOBAL VARIABLES
# Device IPs to shadow
DEVICES= ["192.168.1.4"]
# Device MACs to shadow
MACS= ["a0:18:28:33:34:f8"]
# Device that is always on network so spoofed packets appear real
DST_IP = "192.168.1.54"
# Device MAC to send packets to
DST_MAC = "08:66:98:ed:1e:19"
# Network Interface to use to send packets
INTERFACE = 'wlan0'
# Time format
TIME_FORMAT = '%Y-%m-%d %H:%M:%S'
#########################################################################


def main():
    """
    Determine which devices are in the home. For those not in the
    home send spoofed packets from the absent devices IP/MAC at a
    random interval between 3 and 4 minutes. If all devices are
    present, then sleep for 4 minutes then check again.
    """
    # holds all of the src/dst ethernet tuples for scapy
    ethers = []
    # holds all of the src/dst IP tuples for scapy
    ips = []
    # tcp destination port
    tcp = TCP(dport=4242)
    # message to send for fun
    data = '\xccWhat is the answer to the Ultimate Question of Life,
       The Universe, and Everything?'
```

```
42
43      # for each ip and mac address provided above,
44      # create the tuples for scapy
45      for ip, mac in zip(DEVICES, MACS):
46          ethers.append(Ether(src=mac, dst=DST_MAC))
47          ips.append(IP(src=ip, dst=DST_IP))
48
49      # infinite loop
50      while 1:
51          # initialize variables
52          all_present = True
53          absent_devices = []
54          # for each device, check if it is present. If it is not
55          # present add device IP and ethernet tuples to the list
56          #of absent devices
57          for ip, ether, dev in zip(ips, ethers, DEVICES):
58              device_present = check_device(dev)
59              if device_present != True:
60                  all_present = False
61                  absent_devices.append([ether,ip, dev])
62
63          # if all devices are present, sleep for 5 min, then check again
64          if all_present == True:
65              print 'all devices present; sleeping for 4 min'
66              time.sleep(240)
67
68          # if they are not all present, send 10 packets at random
69          # intervals from the device then sleep between one and 3
70          # minutes before checking for devices again
71          else:
72              print 'All devices not present'
73              for dev in absent_devices:
74                  print 'Sending ten messages on behalf of:', dev[2], 'at
                        ', pretty_time(time.time())
75                  for _ in range(10):
76                      sendp(dev[0]/dev[1]/tcp/data, iface=INTERFACE,
                            verbose=0)
77                      sleep_time = random.randint(1,100)
78                      time.sleep(sleep_time/100)
79              sleep_time = random.randint(180,240)
80              print 'Done sending, going to sleep for', sleep_time, '
                    seconds then restart.'
81              time.sleep(sleep_time)
```

```
82
83          del absent_devices[:]
84
85
86  def check_device(dev):
87      """
88      Checks if a device is presnt using ARP pings
89      """
90      print 'Checking to see if device at ', dev, ' is on the network at
            ', pretty_time(time.time())
91      # Send 10 ARP pings to see if the device is present
92      ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=dev),
           timeout=2, verbose=0, retry=10)
93      for s, r in ans:
94          if r[ARP].psrc == dev:
95              print 'Device responded to ARP'
96              print 'Device is present'
97              return True
98      print 'Device did not respond to ARP'
99      print 'Device is not present'
100     return False
101
102
103 def pretty_time(pkt_time):
104     """
105     Helper function to get better time format
106     """
107     return datetime.fromtimestamp(float(pkt_time)).strftime(
           TIME_FORMAT)
108
109 if __name__ == "__main__":
110     main()
```

Appendix K. Results Script

```
#!/usr/bin/python
# results.py
# Compares BLE/Wi-Fi logs with events identified by CITIoT. Provides
# True Positive, False Negative, and False Positive rates. Also,
# provides a .csv file with events categorized as TP, FN, or FP.
# If a motion/camera/BLE event is identified +-1 minute from a
# logged event, then classify as a TP. This is because CITIoT is
# identifying a real event, but the timing may be too close that it
# identifies the time off by a minute.
import sys, getopt, csv, datetime
######################################################################
# GLOBAL VARIABLES
BLE_DEVICES = {'Eve Door 91B3', 'Eve Room 4A04', 'Eve Weather 943D'
               'Eve Motion 31A7', 'Eve Energy 556E', 'Gunbox',
               'BLELock', '00000b67', 'Instant Pot Smart',
               'PLAYBULB',}
######################################################################

def main(argv):
    """
    Compares logs to CITIoT event identification. Can compare
    BLE and Wi-Fi logs and events or just one of the components
    """
    ble_log_file_name = ""
    wifi_log_file_name = ""
    ble_events_file_name = ""
    wifi_events_file_name = ""

    try:
        opts, args = getopt.getopt(argv, "ha:b:c:d:", ["help"])
    except getopt.GetoptError:
        print 'results.py -a <ble_log.csv> -b <wifi_log> -c <ble_vents.
            csv> -d <wifi_events'
        sys.exit(2)
    for opt, arg, in opts:
        if opt in ("-h", "--help"):
            print "Help:"
            print 'results.py -a <ble_log.csv> -b <wifi_log> -c <
                ble_vents.csv> -d <wifi_events'
            sys.exit()
        elif opt == '-a':
            ble_log_file_name = arg
```

```
41          elif opt == '-b':
42              wifi_log_file_name = arg
43          elif opt == '-c':
44              ble_events_file_name = arg
45          elif opt == '-d':
46              wifi_events_file_name = arg
47
48      # variables to store results
49      total_true_pos = 0
50      total_false_pos = 0
51      total_false_neg = 0
52      total_num_events = 0
53      total_num_events_id = 0
54
55      #r results file information
56      date = ""
57      results_file_name = ""
58
59      # list to store logs and events
60      ble_logs = []
61      ble_events = []
62      wifi_logs = []
63      wifi_events = []
64
65      # lists to store output vlaues
66      ble_csv_entries = []
67      wifi_csv_entries = []
68      total_csv_entries = []
69      total_true_pos_list = []
70      total_false_pos_list = []
71      total_false_neg_list = []
72
73      # if the user provided BLE log and event filenames analyze
74      # BLE results
75      if (ble_log_file_name != "") and (ble_events_file_name != ""):
76          with open(ble_log_file_name, 'rb') as curr_file:
77              reader = csv.reader(curr_file)
78              ble_logs = list(reader)
79
80          with open(ble_events_file_name, 'rb') as curr_file:
81              reader = csv.reader(curr_file)
82              ble_events = list(reader)
83
```

```
84          # get information used for the output file
85          pkt_time = ble_logs[0][0]
86          date = datetime.datetime.strptime(pkt_time, '%Y-%m-%d %H:%M').
                strftime('%d%b%y')
87          results_file_name = "results_" + date + ".csv"
88
89          # start the output for the BLE results
90          ble_csv_entries.append(["BLE Results:",""])
91
92          # compare id'd events to the logs
93          true_pos, false_pos, false_neg, num_events, num_events_id,
                csv_entries, true_pos_list, false_pos_list, false_neg_list
                = calculate_results(ble_logs, ble_events)
94
95          # add BLE results to running totals
96          total_true_pos += true_pos
97          total_false_pos += false_pos
98          total_false_neg += false_neg
99          total_num_events += num_events
100         total_num_events_id += num_events_id
101         ble_csv_entries += csv_entries
102         total_true_pos_list += true_pos_list
103         total_false_pos_list += false_pos_list
104         total_false_neg_list += false_neg_list
105
106     # if the user provided Wi-Fi log and event filenames analyze
107     # Wi-Fi results
108     if (wifi_log_file_name != "") and (wifi_events_file_name != ""):
109         with open(wifi_log_file_name, 'rb') as curr_file:
110             reader = csv.reader(curr_file)
111             wifi_logs = list(reader)
112
113         with open(wifi_events_file_name, 'rb') as curr_file:
114             reader = csv.reader(curr_file)
115             wifi_events = list(reader)
116
117         # get information used for the output file
118         pkt_time = wifi_logs[0][0]
119         date = datetime.datetime.strptime(pkt_time, '%Y-%m-%d %H:%M').
                strftime('%d%b%y')
120         results_file_name = "results_" + date + ".csv"
121
122         # start the output for the BLE results
```

```
123         wifi_csv_entries.append(["Wi-Fi Results:",""])
124
125         # compare id'd events to the logs
126         true_pos, false_pos, false_neg, num_events, num_events_id,
               csv_entries, true_pos_list, false_pos_list, false_neg_list
               = calculate_results(wifi_logs, wifi_events)
127
128         # add BLE results to running totals
129         total_true_pos += true_pos
130         total_false_pos += false_pos
131         total_false_neg += false_neg
132         total_num_events += num_events
133         total_num_events_id += num_events_id
134         wifi_csv_entries += csv_entries
135         total_true_pos_list += true_pos_list
136         total_false_pos_list += false_pos_list
137         total_false_neg_list += false_neg_list
138
139     # prepare complete list of results
140     total_csv_entries.append(["Total Results:", ""])
141     total_csv_entries.append(["True Positives", total_true_pos])
142     total_csv_entries.append(["False Positives", total_false_pos])
143     total_csv_entries.append(["False Negatives", total_false_neg])
144     total_csv_entries.append(["Number of events logged",
           total_num_events])
145     total_csv_entries.append(["Number of events identified",
           total_num_events_id])
146     total_csv_entries.append(["Event identification success rate",
147                       float(total_true_pos) / float(total_num_events)
                             * 100])
148     total_csv_entries.append(["Event identification false positive
           rate",
149                       float(total_false_pos) / float(
                             total_num_events_id) * 100])
150     total_csv_entries.append(["Event identification false negative
           rate",
151                       float(total_false_neg) / float(total_num_events)
                              * 100])
152
153     # include the results for the files provided by the user
154     if(ble_log_file_name != "") and (wifi_log_file_name != ""):
155         # combine columns for BLE and Wi-Fi results
```

```
156         combined_csv_entries = [x+y+z for x, y, z in zip (
                ble_csv_entries, wifi_csv_entries, total_csv_entries)]
157     elif ble_log_file_name == "":
158         combined_csv_entries = wifi_csv_entries
159     elif wifi_log_file_name == "":
160         combined_csv_entries = ble_csv_entries
161 
162     # write results to a file
163     with open(results_file_name, 'w') as curr_file:
164         for line in combined_csv_entries:
165             csv.writer(curr_file).writerow(line)
166 
167         csv.writer(curr_file).writerow(["True Positives:"])
168         for line in total_true_pos_list:
169             csv.writer(curr_file).writerow(line)
170 
171         csv.writer(curr_file).writerow(["False Positives:"])
172         for line in total_false_pos_list:
173             csv.writer(curr_file).writerow(line)
174 
175         csv.writer(curr_file).writerow(["False Negatives:"])
176         for line in total_false_neg_list:
177             csv.writer(curr_file).writerow(line)
178 
179     # combine log files and events file to use in R script
180     combined_logs_file = "logs_" + date + ".csv"
181     combined_logs = ble_logs + wifi_logs
182     combined_events_file = "events_" + date + ".csv"
183     combined_events = ble_events + wifi_events
184 
185     # write combined log/events to files
186     with open(combined_logs_file, 'w') as curr_file:
187         for line in combined_logs:
188             csv.writer(curr_file).writerow(line)
189 
190     with open(combined_events_file, 'w') as curr_file:
191         for line in combined_events:
192             csv.writer(curr_file).writerow(line)
193 
194 
195 def calculate_results(logs, events):
196     """
197     Compares log to id'd events. For NetCam/Motion and BLE events
```

```
198     acknowledges a success if the id'd event was found +- 1 min of
199     the log.
200     """
201     true_pos = 0
202     false_pos = 0
203     false_neg = 0
204     true_pos_list =[]
205     false_pos_list = []
206     false_neg_list = []
207
208     # if an event is in the logs and in the event list then TP
209     # if an event is in the logs but not in the event list then FN
210     for line in logs:
211         if line in events:
212             true_pos += 1
213             true_pos_list.append(line)
214         if line not in events:
215             if line[1] == "NetCam" or line[1] == "Motion" or line[1] in
                    BLE_DEVICES:
216                 if check_event(line, events) != False:
217                     true_pos += 1
218                     true_pos_list.append(line)
219                 else:
220                     false_neg += 1
221                     false_neg_list.append(line)
222             else:
223                 false_neg += 1
224                 false_neg_list.append(line)
225
226     # if an event is in the events list but not in log then FP
227     for line in events:
228         if line not in logs:
229             if line[1] == "NetCam" or line[1] == "Motion" or line[1] in
                    BLE_DEVICES:
230                 time = check_event(line, logs)
231                 if time == False:
232                     false_pos += 1
233                     false_pos_list.append(line)
234                 else:
235                     # line up times so look better in plots
236                     line[0] = time
237             else:
238                 false_pos += 1
```

```
239                 false_pos_list.append(line)
240
241     num_events = len(logs)
242     num_events_id = len(events)
243     csv_entries = []
244
245     # results per protocol
246     csv_entries.append(["True Positives", true_pos])
247     csv_entries.append(["False Positives", false_pos])
248     csv_entries.append(["False Negatives", false_neg])
249     csv_entries.append(["Number of events logged", num_events])
250     csv_entries.append(["Number of events identified", num_events_id])
251     csv_entries.append(["Event identification success rate",
252                                 float(true_pos)/float(num_events)
                                        *100])
253     csv_entries.append(["Event identification false positive rate",
254                                 float(false_pos) / float(
                                        num_events_id)*100])
255     csv_entries.append(["Event identification false negative rate",
256                                 float(false_neg) / float(num_events)
                                        *100])
257
258     return true_pos, false_pos, false_neg, num_events, num_events_id,
            csv_entries, true_pos_list, false_pos_list, false_neg_list
259
260
261 def check_event(line, check_list):
262     """
263     Because motion/camera BLE events take time to send, if one was
264     sent the minute before then ignore a potential new event could
265     cause us to miss two events in a row, but most motion devices
266     have at least a 60 second no motion sensor before restarting, so
267     this should not be an issue.
268     """
269     pkt_time = line[0]
270     device = line[1]
271     date = datetime.datetime.strptime(pkt_time, '%Y-%m-%d %H:%M')
272     date_minus_1 = date - datetime.timedelta(minutes=1)
273     date_minus_1 = date_minus_1.strftime('%Y-%m-%d %H:%M')
274     date_plus_1 = date + datetime.timedelta(minutes=1)
275     date_plus_1 = date_plus_1.strftime('%Y-%m-%d %H:%M')
276     if [date_minus_1, device, '1'] in check_list:
277         print "got minus 1"
```

```
278         return date_minus_1
279     elif [date_plus_1, device, '1'] in check_list:
280         print "got plus 1"
281         return date_plus_1
282     else:
283         return False
284 
285 
286 # call main
287 if __name__ == "__main__":
288     main(sys.argv[1:])
```

Appendix L. Log Script

```
#!/usr/bin/python
# logParser.py
# Helper file to parse BLE and Wi-Fi logs. Very specific to the
# logs produced by Homebridge and the user. Provides two new
# logs containing the time, device, and '1' for each event.
# The '1' is used by the R script that plots the log values.
import sys, getopt, csv, datetime

#########################################################################
# GLOBAL VARIABLES
DEVICES = {'Switch 1': 'b4:75:0e:0d:33:d5',
           'Switch 2': 'b4:75:0e:0d:94:65',
           'Switch 3': '94:10:3e:2b:7a:55',
           'Switch 4': '14:91:82:c8:6a:09',
           'Mini': '60:38:e0:ee:7c:e5',
           'Insight': '14:91:82:24:dd:35',
           'WeMo Motion': 'ec:1a:59:f1:fb:21',
           'NetCam': 'ec:1a:59:e4:fd:41',
           'NetCam Motion,smbeyer8': 'ec:1a:59:e4:fd:41',
           'Motion': 'ec:1a:59:f1:fb:21'}
ACTIONS = {'Set state: On',
           'Set state: Off',
           'Motion Sensor: Detected'}
DEVICE_NAME = {'ec:1a:59:e4:fd:41': 'NetCam',
               'b4:75:0e:0d:33:d5': 'Switch1',
               'b4:75:0e:0d:94:65': 'Switch2',
               '94:10:3e:2b:7a:55': 'Switch3',
               '14:91:82:c8:6a:09': 'Switch4',
               'ec:1a:59:f1:fb:21': 'Motion',
               '14:91:82:24:dd:35': 'Insight',
               '60:38:e0:ee:7c:e5': 'Mini'}
TIME_FORMAT = '%Y-%m-%d %H:%M'
#########################################################################


def main(argv):
    """
    Create properly formatted logs for BLE and/or Wi-Fi
    """
    try:
        opts, args = getopt.getopt(argv, "hw:b:", ["help"])
    except getopt.GetoptError:
```

```
43          print 'parse_log.py -w <wifi log file> -b <ble log file>'
44          sys.exit(2)
45      for opt, arg, in opts:
46          if opt in ("-h", "--help"):
47              print 'parse_log.py -w <wifi log file> -b <ble log file>'
48              sys.exit()
49          elif opt == '-w':
50              print "arg:", arg
51              file_name = arg
52              parse_wifi_log(file_name)
53          elif opt == '-b':
54              file_name = arg
55              parse_ble_log(file_name)
56
57
58  def parse_wifi_log(file_name):
59      """
60      Parses the homebridge event log looking for devices and
61      events. Creates a .csv file with the time and device of
62      each event. A '1' is added as the third value to be used
63      in R script for plotting.
64      """
65
66      with open(file_name, 'r') as f:
67          content = f.readlines()
68      lines = []
69      # pull information from log
70      for line in content:
71          a = stringbtwn(line, '[', 2, ']', 2).replace(',', '')
72          b = stringbtwn(line, ']', 3, '-', 1).strip()
73          c = stringbtwn(line, '-', 1, '\n', 1).strip()
74          lines.append([a,b,c])
75
76      events = []
77      # for each
78      for line in lines:
79          device = line[1]
80          action = line[2]
81          if (device in list(DEVICES.keys())) and (action in ACTIONS):
82              date = line[0]
83              # reformat the time string
84              date = datetime.datetime.strptime(date, "%m/%d/%Y %I:%M:%S
                    %p").strftime(TIME_FORMAT)
```

```
85              event = [date, DEVICE_NAME[DEVICES[device]], '1']
86              if event not in events:
87                  events.append(event)
88
89      # save the results into a new file
90      file_date = datetime.datetime.strptime(date, TIME_FORMAT).strftime
            ("%d%b%y")
91      csv_file = "wifi_log_" + file_date + ".csv"
92      with open(csv_file, 'w') as curr_file:
93          for event in events:
94              csv.writer(curr_file).writerow(event)
95
96
97  def findnth(string, char, n):
98      """
99      Helper function that finds the location of the nth instance of a
100     character in a string
101     """
102     parts=string.split(char, n)
103     if len(parts)<=n:
104         return -1
105     return len(string)-len(parts[-1])-len(char)
106
107 def stringbtwn(string, char1, n1, char2, n2):
108     """
109     Helper function that finds the value between the given nth
110     instance of a character. If an input string of "[test] [test2]"
            and values [, 2, ], 2 were passed to the function, the function
111     would return "test2"
112     """
113     a = findnth(string, char1, n1)
114     b = findnth(string, char2, n2)
115     if (a == -1) or (b == -1):
116         return ""
117     return string[a+1:b]
118
119
120 def parse_ble_log(file_name):
121     """
122     Parses the BLE user log. Reformats the datetime and saves back
123     to the file.
124     """
125     with open(file_name, 'rb') as curr_file:
```

```
126         reader = csv.reader(curr_file)
127         contents = list(reader)
128 
129     lines = []
130     for line in contents:
131         date = line[0]
132         date = datetime.datetime.strptime(date, "%m/%d/%Y %H:%M").
                strftime(TIME_FORMAT)
133         device = line[1]
134         value = line[2]
135         lines.append([date,device,value])
136 
137     with open(file_name, 'w') as curr_file:
138         for line in sorted(lines):
139             csv.writer(curr_file).writerow(line)
140 
141 
142 # call main
143 if __name__ == "__main__":
144   main(sys.argv[1:])
```

Appendix M. R Script

```
library(tidyverse)
library(plotly)
library(scales)

# -- PLOTTING FUNCTIONS ------------------------------------------
## Combined with no mitigation
# Repeat for trials #1-5
log_file_1 = 'logs_16Aug17.csv'
event_file_1 = 'events_16Aug17.csv'

myCols <- c("Time", "Device", "Event", "Data_Source")

log_plot_1 <- read_csv(log_file_1,
    col_names = c("Time", "Device", "Event"), na = ".")
log_plot_1$Data_Source <- rep("Log", nrow(log_plot_1))
log_plot_1_final <- log_plot_1[myCols]

event_plot_1 <- read_csv(event_file_1,
    col_names = c("Time", "Device", "Event"), na = ".")
event_plot_1$Data_Source <- rep("CITIoT", nrow(event_plot_1))
event_plot_1_final <- event_plot_1[myCols]

data_1 <- rbind(log_plot_1_final, event_plot_1_final)
data_1$Time <- as.POSIXct(data_1$Time, format = "%m/%d/%y %H:%M",
    tz = "America/New_York")
date_1 <- format(min(data_1$Time), '%d %b %y')

plot_1 <- ggplot(data_1) +
    aes(Time, Event, color=Data_Source, shape=Data_Source,
        size = Data_Source, label = Time,
        text = paste("Device: ", Device)) +
    geom_point() + scale_shape_manual(values=c(8, 1)) +
    scale_size_manual(values=c(2, 4)) + ggtitle(date_1) +
    labs( x= "Time (minutes)", y = "Event") +
    theme(axis.text.y=element_blank())

ggplotly(plot_1 +
    scale_x_datetime(breaks = date_breaks("5 min"),
        labels = date_format ("%H%M")),
        tooltip = c('text','label'))
ggplotly(plot_1 +
    scale_x_datetime(breaks = date_breaks("1 hour"),
```

```
43        labels = date_format ("%H%M")),
44        tooltip = c('text','label'))
45
46 ## Combined with mitigation
47 # Repeat for trials #6-10
48 log_file_6 = 'logs_19Dec17.csv'
49 event_file_6 = 'events_19Dec17.csv'
50
51 myCols <- c("Time", "Device", "Event", "Data_Source")
52
53 log_plot_6 <- read_csv(log_file_6,
54     col_names = c("Time", "Device", "Event"), na = ".")
55 log_plot_6$Data_Source <- rep("Log", nrow(log_plot_6))
56 log_plot_6_final <- log_plot_6[myCols]
57
58 event_plot_6 <- read_csv(event_file_6,
59     col_names = c("Time", "Device", "Event"), na = ".")
60 event_plot_6$Data_Source <- rep("CITIoT", nrow(event_plot_6))
61 event_plot_6_final <- event_plot_6[myCols]
62
63 data_6 <- rbind(log_plot_6_final, event_plot_6_final)
64 data_6$Time <- as.POSIXct(data_6$Time, format = "%m/%d/%y %H:%M",
65     tz = "America/New_York")
66 min(data_6$Time)
67 date_6 <- format(min(data_6$Time), '%d %b %y')
68
69 plot_6 <- ggplot(data_6) +
70     aes(Time, Event, color=Data_Source, shape=Data_Source,
71         size = Data_Source, label = Time,
72         text = paste("Device: ", Device)) +
73     geom_point() + scale_shape_manual(values=c(8, 1)) +
74     scale_size_manual(values=c(2, 4)) + ggtitle(date_6) +
75     labs( x= "Time (minutes)", y = "Event") +
76     theme(axis.text.y=element_blank())
77
78 ggplotly(plot_6 +
79     scale_x_datetime(breaks = date_breaks("1 hour"),
80       labels = date_format ("%H%M")),
81       tooltip = c('text','label'))
82
83
84 # -- RESPONSE VARIABLE FUNCTIONS ----------------------------------
85 # load and prepare data
```

```
ble_results <- read.csv("BLE_Results.csv")
ble_results$Config <- "BLE"
wifi_results <- read.csv("WIFI_Results.csv")
wifi_results$Config <- "WIFI"
combined_results <- read.csv("COMBINED_results.csv")
combined_results$Config <- "COMBINED"
miotl_results <- read.csv("MIOTL_results.csv")
miotl_results$Config <- "MIOTL"
total_results = rbind(wifi_results, ble_results,
                      combined_results, miotl_results)
total_results$Config <- as.character(total_results$Config)
total_results$Config <- factor(total_results$Config,
                      levels=unique(total_results$Config))

# Calculate EITP results
bleESIR.SE = summarySE(ble_results, measurevar = "EITP",
    groupvars = c("Config"), na.rm=TRUE)
wifiEITP.SE = summarySE(wifi_results, measurevar = "EITP",
    groupvars = c("Config"), na.rm=TRUE)
combinedEITP.SE = summarySE(combined_results, measurevar = "EITP",
    groupvars = c("Config"), na.rm=TRUE)
miotlEITP.SE = summarySE(miotl_results, measurevar = "EITP",
    groupvars = c("Config"), na.rm=TRUE)
totalEITP.SE = rbind(bleESIR.SE, wifiEITP.SE,
                     combinedEITP.SE, miotlEITP.SE)
totalEITP.SE
ggplot(total_results, aes(x=Config,y=EITP)) +
    geom_boxplot() + geom_jitter(alpha=.1, shape=16, size=.5) +
    labs(size=4, x ="Configuration", y="Success Rate") +
    theme_bw() +
    theme(axis.text=element_text(size=12),
      axis.title=element_text(size=14,face="bold"))

# Calculate EIFP results
bleEIFP.SE = summarySE(ble_results, measurevar = "EIFP",
    groupvars = c("Config"), na.rm=TRUE)
wifiEIFP.SE = summarySE(wifi_results, measurevar = "EIFP",
    groupvars = c("Config"), na.rm=TRUE)
combinedEIFP.SE = summarySE(combined_results, measurevar = "EIFP",
    groupvars = c("Config"), na.rm=TRUE)
miotlEIFP.SE = summarySE(miotl_results, measurevar = "EIFP",
    groupvars = c("Config"), na.rm=TRUE)
```

```
totalEIFP.SE = rbind(bleEIFP.SE, wifiEIFP.SE, combinedEIFP.SE,
    miotlEIFP.SE)
totalEIFP.SE
ggplot(total_results, aes(x=Config,y=EIFP)) +
    geom_boxplot() + geom_jitter(alpha=.1, shape=16, size=.5) +
    labs(size=4, x ="Configuration", y="False Positive Rate") +
    theme_bw() +
    theme(axis.text=element_text(size=12),
      axis.title=element_text(size=14,face="bold"))

# Calculate EIFN results
bleEIFN.SE = summarySE(ble_results, measurevar = "EIFN",
    groupvars = c("Config"), na.rm=TRUE)
wifiEIFN.SE = summarySE(wifi_results, measurevar = "EIFN",
    groupvars = c("Config"), na.rm=TRUE)
combinedEIFN.SE = summarySE(combined_results, measurevar = "EIFN",
    groupvars = c("Config"), na.rm=TRUE)
miotlEIFN.SE = summarySE(miotl_results, measurevar = "EIFN",
    groupvars = c("Config"), na.rm=TRUE)
totalEIFN.SE = rbind(bleEIFN.SE, wifiEIFN.SE,
                     combinedEIFN.SE, miotlEIFN.SE)
totalEIFN.SE
ggplot(total_results, aes(x=Config,y=EIFN)) +
    geom_boxplot() + geom_jitter(alpha=.1, shape=16, size=.5) +
    labs(size=4, x ="Configuration", y="False Negative Rate") +
    theme_bw() +
    theme(axis.text=element_text(size=12),
      axis.title=element_text(size=14,face="bold"))

# Two sample t-tests to compare Wi-Fi no mitigation and Wi-Fi with
    mitigation
t.test(wifi_results$EITPR, miotl_results$EITPR, var.equal=T,
    alternative = "greater")
t.test(wifi_results$EIFPR, miotl_results$EIFPR, var.equal = T,
    alternative = "less")
```

Appendix N. Device Classification Results

Actual Classifications	
Device	**Classification**
Switch1	Outlet
Switch2	Outlet
Switch3	Outlet
Switch4	Outlet
Insight	Outlet
Mini	Outlet
Motion	Sensor
NetCam	Camera

CITIoT Results Without Mitigation					
Device	**16 Aug 17**	**22 Aug 17**	**23 Aug 17**	**25 Aug 17**	**26 Aug 17**
Switch1	Outlet	Outlet	Outlet	Outlet	Outlet
Switch2	Outlet	Outlet	Outlet	Outlet	Outlet
Switch3	Outlet	Outlet	Outlet	Outlet	Outlet
Switch4	Outlet	Outlet	Outlet	Outlet	Outlet
Insight	Outlet	Outlet	Outlet	Outlet	Outlet
Mini	Outlet	Outlet	Outlet	Outlet	Outlet
Motion	Sensor	Sensor	Sensor	Sensor	Sensor
NetCam	Camera	Camera	Camera	Camera	Camera
CITIoT Results With Mitigation					
Device	**19 Dec 17**	**22 Dec 17**	**23 Dec 17**	**26 Dec 17**	**27 Dec 17**
Switch1	Outlet	Outlet	Outlet	Outlet	Outlet
Switch2	Outlet	Outlet	Outlet	Outlet	Outlet
Switch3	Outlet	Outlet	Outlet	Outlet	Outlet
Switch4	Outlet	Outlet	Outlet	Outlet	Outlet
Insight	Outlet	Outlet	Outlet	Outlet	Outlet
Mini	Outlet	Outlet	Outlet	Outlet	Outlet
Motion	Outlet	Outlet	Outlet	Outlet	Outlet
NetCam	Outlet	Outlet	Outlet	Outlet	Outlet

Appendix O. Event Identification Results

16 Aug 17			
	BLE Results	**Wi-Fi Results**	**Total Results**
True Positives	N/A	31	31
False Posisitives	N/A	0	0
False Negatives	N/A	0	0
Events logged	N/A	31	31
Events ID'd	N/A	31	31
EITP	N/A	100.000 %	100.000 %
EIFP	N/A	0.000 %	0.000 %
EIFN	N/A	0.000 %	0.000 %

BLE Events		**Wi-Fi Events**	
True Positives:			
		8/16/17 7:05	Switch1
		8/16/17 7:05	Switch2
		8/16/17 7:07	Switch3
		8/16/17 7:08	Switch4
		8/16/17 7:09	Mini
		8/16/17 7:14	Insight
		8/16/17 7:15	NetCam
		8/16/17 7:18	Motion
		8/16/17 7:22	Switch1
		8/16/17 7:23	Switch2
		8/16/17 7:23	Switch3
		8/16/17 7:23	Mini
		8/16/17 7:23	Switch4
		8/16/17 7:23	Insight
		8/16/17 7:32	NetCam
		8/16/17 15:33	NetCam
		8/16/17 15:42	NetCam
		8/16/17 16:24	Switch1
		8/16/17 16:25	Switch2
		8/16/17 16:26	Switch3
		8/16/17 16:27	Switch4
		8/16/17 16:28	Mini
		8/16/17 16:32	Insight
		8/16/17 16:33	NetCam
		8/16/17 16:35	Motion
		8/16/17 16:39	Switch1

		8/16/17 16:39	Switch2
		8/16/17 16:39	Switch3
		8/16/17 16:39	Switch4
		8/16/17 16:39	Mini
		8/16/17 16:39	Insight
False Positives:			
False Negatives:			

22 Aug 17			
	BLE Results	**Wi-Fi Results**	**Total Results**
True Positives	33	34	67
False Posisitives	2	1	3
False Negatives	5	1	6
Events logged	38	35	73
Events ID'd	36	35	71
EITP	86.842 %	97.143 %	91.781 %
EIFP	5.556 %	2.857 %	4.225 %
EIFN	13.158 %	2.857 %	8.219 %

BLE Events		**Wi-Fi Events**	
True Positives:			
8/22/17 5:58	Eve Motion 31A7	8/22/17 5:57	Motion
8/22/17 5:59	Instant Pot Smart	8/22/17 6:00	Insight
8/22/17 6:00	Eve Motion 31A7	8/22/17 6:01	Mini
8/22/17 6:04	PLAYBULB	8/22/17 6:02	Switch2
8/22/17 6:06	Eve Energy 556E	8/22/17 6:03	NetCam
8/22/17 6:10	Eve Weather 943D	8/22/17 6:05	Switch1
8/22/17 6:14	Instant Pot Smart	8/22/17 6:08	Switch3
8/22/17 6:14	PLAYBULB	8/22/17 6:13	Motion
8/22/17 6:15	Gunbox	8/22/17 6:16	Mini
8/22/17 6:18	Eve Motion 31A7	8/22/17 6:16	Switch1
8/22/17 6:19	Eve Room 4A04	8/22/17 6:16	Switch4
8/22/17 6:20	Eve Motion 31A7	8/22/17 6:17	NetCam
8/22/17 6:23	Eve Motion 31A7	8/22/17 6:18	Insight
8/22/17 6:23	Eve Room 4A04	8/22/17 6:18	Switch2
8/22/17 6:24	00000b67	8/22/17 6:18	Switch3
8/22/17 6:25	BLELock	8/22/17 8:18	Motion
8/22/17 6:26	Eve Door 91B3	8/22/17 14:16	Switch1
8/22/17 16:18	Eve Weather 943D	8/22/17 14:19	Switch1
8/22/17 16:23	Eve Room 4A04	8/22/17 16:19	Switch4
8/22/17 16:24	Gunbox	8/22/17 16:20	Insight

8/22/17 16:25	Instant Pot Smart	8/22/17 16:26	Switch1
8/22/17 16:27	Eve Motion 31A7	8/22/17 16:28	Motion
8/22/17 16:35	PLAYBULB	8/22/17 16:29	Switch2
8/22/17 16:36	Eve Energy 556E	8/22/17 16:30	NetCam
8/22/17 16:40	Eve Motion 31A7	8/22/17 16:31	Mini
8/22/17 16:40	Gunbox	8/22/17 16:37	Switch3
8/22/17 16:41	Eve Energy 556E	8/22/17 16:39	Switch3
8/22/17 16:41	Eve Room 4A04	8/22/17 16:39	Motion
8/22/17 16:42	Eve Energy 556E	8/22/17 16:39	Insight
8/22/17 16:42	PLAYBULB	8/22/17 16:40	Switch2
8/22/17 16:42	Eve Motion 31A7	8/22/17 16:41	Switch4
8/22/17 16:43	Eve Weather 943D	8/22/17 16:42	Switch1
8/22/17 16:43	Instant Pot Smart	8/22/17 16:42	NetCam
		8/22/17 16:43	Mini
False Positives:			
8/22/17 14:48	Eve Door 91B3	8/22/17 15:18	Motion
8/22/17 16:07	Eve Motion 31A7		
False Negatives:			
8/22/17 6:07	Eve Weather 943D	8/22/17 6:13	Switch4
8/22/17 6:09	Eve Room 4A04		
8/22/17 6:15	Eve Energy 556E		
8/22/17 14:17	Eve Room 4A04		
8/22/17 14:18	Eve Weather 943D		

23 Aug 17			
	BLE Results	**Wi-Fi Results**	**Total Results**
True Positives	42	34	76
False Posisitives	1	2	3
False Negatives	1	3	4
Events logged	43	37	80
Events ID'd	43	36	79
EITP	97.674 %	91.892 %	95.000 %
EIFP	2.326 %	5.556 %	3.797 %
EIFN	2.326 %	8.108 %	5.000 %

BLE Events		**Wi-Fi Events**	
True Positives:			
8/23/17 7:05	Eve Weather 943D	8/23/17 7:04	NetCam
8/23/17 7:06	Eve Room 4A04	8/23/17 7:08	Switch2
8/23/17 7:09	PLAYBULB	8/23/17 7:10	Motion

8/23/17 7:12	Instant Pot Smart	8/23/17 7:11	Switch4
8/23/17 7:14	Eve Energy 556E	8/23/17 7:13	Switch3
8/23/17 7:17	Eve Motion 31A7	8/23/17 7:15	Mini
8/23/17 7:19	Eve Motion 31A7	8/23/17 7:16	Insight
8/23/17 7:20	Eve Energy 556E	8/23/17 7:19	Switch1
8/23/17 7:21	Gunbox	8/23/17 7:21	Switch2
8/23/17 7:21	Instant Pot Smart	8/23/17 7:22	Mini
8/23/17 7:22	Eve Motion 31A7	8/23/17 7:22	Switch3
8/23/17 7:22	Gunbox	8/23/17 7:24	Motion
8/23/17 7:23	Eve Room 4A04	8/23/17 7:24	Switch1
8/23/17 7:24	Eve Motion 31A7	8/23/17 7:25	NetCam
8/23/17 7:25	Eve Weather 943D	8/23/17 7:25	Insight
8/23/17 7:26	PLAYBULB	8/23/17 13:35	Motion
8/23/17 7:31	PLAYBULB	8/23/17 13:38	Motion
8/23/17 7:32	00000b67	8/23/17 13:52	NetCam
8/23/17 7:33	BLELock	8/23/17 16:26	NetCam
8/23/17 7:34	Eve Door 91B3	8/23/17 16:27	NetCam
8/23/17 14:33	Eve Door 91B3	8/23/17 16:28	NetCam
8/23/17 16:20	Eve Door 91B3	8/23/17 17:43	Motion
8/23/17 16:21	Eve Door 91B3	8/23/17 17:43	Insight
8/23/17 16:22	Eve Door 91B3	8/23/17 17:45	Switch2
8/23/17 16:31	Eve Door 91B3	8/23/17 17:49	Switch4
8/23/17 16:43	Eve Door 91B3	8/23/17 17:51	Mini
8/23/17 17:44	Eve Motion 31A7	8/23/17 17:55	NetCam
8/23/17 17:46	Eve Motion 31A7	8/23/17 18:01	Switch4
8/23/17 17:46	Eve Energy 556E	8/23/17 18:01	Switch2
8/23/17 17:50	Gunbox	8/23/17 18:01	NetCam
8/23/17 17:54	Eve Weather 943D	8/23/17 18:01	Switch1
8/23/17 17:56	PLAYBULB	8/23/17 18:01	Switch3
8/23/17 17:57	Instant Pot Smart	8/23/17 18:05	Insight
8/23/17 17:58	Eve Weather 943D	8/23/17 18:07	Mini
8/23/17 18:00	Eve Room 4A04		
8/23/17 18:01	PLAYBULB		
8/23/17 18:05	Instant Pot Smart		
8/23/17 18:05	Eve Motion 31A7		
8/23/17 18:06	Gunbox		
8/23/17 18:06	Instant Pot Smart		
8/23/17 18:06	Eve Motion 31A7		
8/23/17 18:07	Eve Energy 556E		
False Positives:			
8/23/17 15:08	Eve Door 91B3	8/23/17 15:21	Motion

		8/23/17 16:09	NetCam
False Negatives:			
8/23/17 17:53	Eve Room 4A04	8/23/17 7:21	Switch4
		8/23/17 17:47	Switch1
		8/23/17 17:48	Switch3

25 Aug 17			
	BLE Results	**Wi-Fi Results**	**Total Results**
True Positives	50	35	85
False Posisitives	5	1	6
False Negatives	0	2	2
Events logged	50	37	87
Events ID'd	58	35	93
EITP	100.000 %	94.595 %	97.701 %
EIFP	8.621 %	2.857 %	6.452 %
EIFN	0.000 %	5.405 %	2.299 %

BLE Events		**Wi-Fi Events**	
True Positives:			
8/25/17 8:16	Eve Energy 556E	8/25/17 8:11	Switch3
8/25/17 8:19	Eve Room 4A04	8/25/17 8:12	Motion
8/25/17 8:20	PLAYBULB	8/25/17 8:14	Switch2
8/25/17 8:22	Instant Pot Smart	8/25/17 8:15	Switch4
8/25/17 8:23	Eve Motion 31A7	8/25/17 8:17	Insight
8/25/17 8:25	Eve Weather 943D	8/25/17 8:18	NetCam
8/25/17 8:25	Eve Motion 31A7	8/25/17 8:21	Mini
8/25/17 8:26	Eve Energy 556E	8/25/17 8:24	Switch1
8/25/17 8:26	Gunbox	8/25/17 8:26	Switch4
8/25/17 8:26	Instant Pot Smart	8/25/17 8:26	Switch2
8/25/17 8:27	Eve Motion 31A7	8/25/17 8:27	Mini
8/25/17 8:27	Eve Room 4A04	8/25/17 8:27	Switch3
8/25/17 8:28	Eve Weather 943D	8/25/17 8:28	Motion
8/25/17 8:29	Eve Motion 31A7	8/25/17 8:28	Switch1
8/25/17 8:29	PLAYBULB	8/25/17 8:28	NetCam
8/25/17 8:45	BLELock	8/25/17 8:29	Insight
8/25/17 8:45	00000b67	8/25/17 9:45	NetCam
8/25/17 8:46	BLELock	8/25/17 9:55	NetCam
8/25/17 8:46	Eve Door 91B3	8/25/17 19:31	NetCam
8/25/17 10:00	Eve Door 91B3	8/25/17 19:36	Mini
8/25/17 10:00	Eve Energy 556E	8/25/17 19:39	Switch2
8/25/17 10:00	Eve Motion 31A7	8/25/17 19:42	NetCam

8/25/17 10:00	Eve Weather 943D	8/25/17 19:43	NetCam
8/25/17 13:23	Eve Door 91B3	8/25/17 19:44	Insight
8/25/17 13:24	Eve Door 91B3	8/25/17 19:45	Switch3
8/25/17 13:26	Eve Door 91B3	8/25/17 19:47	Switch1
8/25/17 13:28	Eve Door 91B3	8/25/17 19:48	Switch4
8/25/17 13:29	Eve Door 91B3	8/25/17 19:49	NetCam
8/25/17 13:30	Eve Door 91B3	8/25/17 19:51	Insight
8/25/17 13:35	Eve Door 91B3	8/25/17 19:51	NetCam
8/25/17 13:36	Eve Door 91B3	8/25/17 19:52	Switch2
8/25/17 15:32	Eve Door 91B3	8/25/17 19:52	NetCam
8/25/17 15:42	BLELock	8/25/17 19:55	Switch3
8/25/17 19:35	Eve Energy 556E	8/25/17 19:55	Switch4
8/25/17 19:37	Eve Motion 31A7	8/25/17 19:56	NetCam
8/25/17 19:38	PLAYBULB		
8/25/17 19:39	Eve Motion 31A7		
8/25/17 19:40	Gunbox		
8/25/17 19:41	Eve Motion 31A7		
8/25/17 19:41	Eve Weather 943D		
8/25/17 19:43	Eve Motion 31A7		
8/25/17 19:43	Instant Pot Smart		
8/25/17 19:46	Eve Room 4A04		
8/25/17 19:51	Gunbox		
8/25/17 19:52	Eve Room 4A04		
8/25/17 19:54	Eve Energy 556E		
8/25/17 19:54	PLAYBULB		
8/25/17 19:55	Instant Pot Smart		
8/25/17 19:57	Eve Motion 31A7		
8/25/17 19:57	Eve Weather 943D		
False Positives:			
8/25/17 8:44	Eve Motion 31A7	8/25/17 15:30	Motion
8/25/17 10:15	Eve Weather 943D		
8/25/17 10:38	Eve Room 4A04		
8/25/17 13:33	Eve Door 91B3		
8/25/17 16:11	Eve Room 4A04		
False Negatives:			
		8/25/17 19:52	Mini
		8/25/17 19:55	Switch1

26 Aug 17			
	BLE Results	**Wi-Fi Results**	**Total Results**
True Positives	37	28	65
False Posisitives	0	1	1
False Negatives	2	5	7
Events logged	39	33	72
Events ID'd	39	29	68
EITP	94.872 %	84.848 %	90.278 %
EIFP	0.000 %	3.448 %	1.471 %
EIFN	5.128 %	15.152 %	9.722 %

BLE Events		**Wi-Fi Events**	
True Positives:			
8/26/17 9:48	Eve Motion 31A7	8/26/17 9:49	Switch3
8/26/17 9:50	Eve Motion 31A7	8/26/17 9:50	Insight
8/26/17 9:52	PLAYBULB	8/26/17 9:53	Switch4
8/26/17 9:54	Eve Weather 943D	8/26/17 9:55	Switch1
8/26/17 9:57	Eve Room 4A04	8/26/17 9:56	NetCam
8/26/17 9:58	Instant Pot Smart	8/26/17 10:00	Motion
8/26/17 9:59	Eve Energy 556E	8/26/17 10:01	NetCam
8/26/17 10:02	Gunbox	8/26/17 10:05	Insight
8/26/17 10:03	Instant Pot Smart	8/26/17 10:05	Switch3
8/26/17 10:04	Eve Room 4A04	8/26/17 10:06	Motion
8/26/17 10:04	Eve Weather 943D	8/26/17 10:06	Switch4
8/26/17 10:05	Eve Energy 556E	8/26/17 10:06	Mini
8/26/17 10:05	Eve Motion 31A7	8/26/17 13:47	NetCam
8/26/17 10:05	PLAYBULB	8/26/17 17:40	Switch3
8/26/17 10:07	Eve Motion 31A7	8/26/17 17:43	Motion
8/26/17 10:24	Eve Door 91B3	8/26/17 17:45	Switch4
8/26/17 10:25	Eve Door 91B3	8/26/17 17:46	Mini
8/26/17 17:36	BLELock	8/26/17 17:50	NetCam
8/26/17 17:36	00000b67	8/26/17 17:51	Switch2
8/26/17 17:37	Eve Door 91B3	8/26/17 17:52	Switch1
8/26/17 17:38	Eve Door 91B3	8/26/17 17:54	NetCam
8/26/17 17:39	Gunbox	8/26/17 17:55	Switch2
8/26/17 17:41	Eve Motion 31A7	8/26/17 17:55	Mini
8/26/17 17:42	Eve Energy 556E	8/26/17 17:55	Insight
8/26/17 17:43	Eve Motion 31A7	8/26/17 17:55	Switch4
8/26/17 17:47	Eve Weather 943D	8/26/17 17:56	Switch1
8/26/17 17:48	Instant Pot Smart	8/26/17 17:56	Switch3
8/26/17 17:49	Eve Room 4A04	8/26/17 17:57	Motion

8/26/17 17:53	Eve Weather 943D		
8/26/17 17:53	PLAYBULB		
8/26/17 17:54	Eve Room 4A04		
8/26/17 17:54	Gunbox		
8/26/17 17:55	Eve Motion 31A7		
8/26/17 17:56	Eve Energy 556E		
8/26/17 17:56	Instant Pot Smart		
8/26/17 17:57	Eve Motion 31A7		
8/26/17 17:59	BLELock		
False Positives:			
		8/26/17 15:31	Motion
False Negatives:			
8/26/17 17:57	00000b67	8/26/17 9:46	Mini
8/26/17 17:59	Eve Door 91B3	8/26/17 9:51	Switch2
		8/26/17 10:04	Switch2
		8/26/17 10:06	Switch1
		8/26/17 17:44	Insight

Appendix P. MAC Track Results

User Logs		
Date	**Arrival**	**Departure**
Without mitigation		
8/16/17	7:03	7:34
	15:43	16:10
	16:20	16:50
8/22/17	5:53	6:27
	14:47	15:03
	15:55	16:44
8/23/17	7:02	7:35
	17:16	17:29
	17:37	18:07
8/25/17	8:09	8:33
	8:39	8:51
	9:45	11:10
	11:23	15:42
	16:36	16:57
	17:06	17:16
	19:23	19:58
8/26/17	9:43	18:00
With mitigation		
12/19/17	9:31	10:40
	16:37	18:25
	19:21	19:40
12/22/17	9:37	10:42
	17:56	19:28
12/23/17	9:51	11:13
	13:41	14:26
	18:31	18:46
12/26/17	10:54	14:23
	16:57	17:23
12/27/17	10:06	13:54
	15:12	17:40

CITIoT Results		
Date	**Arrival**	**Departure**
Without mitigation		
8/16/17	7:03	7:34
	15:44	16:12
	16:20	16:50
8/22/17	5:53	6:27
	14:47	15:03
	15:54	16:44
8/23/17	7:02	7:36
	17:16	17:29
	17:36	18:07
8/25/17	8:09	8:33
	8:39	8:51
	9:46	11:10
	11:23	15:43
	16:37	16:57
	17:06	17:16
	19:23	19:58
8/26/17	9:43	18:00
With mitigation		
12/19/17	9:31	19:40
12/22/17	9:37	19:28
12/23/17	9:51	18:46
12/26/17	10:54	17:23
12/27/17	10:06	14:07
	14:14	17:40

Bibliography

1. United States Government Accountability Office, "Internet of Things: Enhanced Assessments and Guidance Are Needed to Address Security Risks in DOD," 2017, accessed Feb 11, 2018. [Online]. Available: www.gao.gov/assets/690/686203.pdf.

2. Consumer Technology Association, "2018 tech industry revenue to reach record $351 billion, says CTA," 2018, accessed Jan 7, 2018. [Online]. Available: www.cta.tech/News/Press-Releases/2018/January/2018-Tech-Industry-Revenue-to-Reach-Record-$351-Bi.aspx.

3. *Wireless LAN Medium Access Control, MAC, and Physical Layer, PHY, Specification*, IEEE Standard 802.11, 2016, accessed Aug 27, 2017. [Online]. Available: ieeexplore.ieee.org/browse/standards/get-program/page/series?id=68.

4. J. Kurose and K. Ross, *Computer Networking A Top-Down Approach Featuring the Internet*, 7th ed. Boston, MA: Pearson Education, Inc., 2017.

5. E. Skoudis and T. Liston, *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses*, 2nd ed. Upper Saddle River, NJ: Prentice Hall Press, 2006.

6. V. Jacobson, C. Leres, and S. McCanne, "Tcpdump manual page," 2017, accessed Dec 8, 2017. [Online]. Available: www.tcpdump.org/tcpdump_man.html.

7. *Specification of the Bluetooth System*, Core Version 4.0, 2010, accessed Aug 27, 2017. [Online]. Available: www.bluetooth.com/specifications/adopted-specifications/legacy-specifications.

8. R. Heydon, *Bluetooth Low Energy: The Developer's Handbook*. Upper Saddle River, NJ: Prentice Hall, 2013, vol. 1.

9. *Specification of the Bluetooth System*, Core Version 5.0, 2016, accessed Aug 30, 2017. [Online]. Available: www.bluetooth.com/specifications/bluetooth-core-specification.

10. Apple, "iPhone 8 Tech Specs," 2017, accessed Feb 11, 2018. [Online]. Available: www.apple.com/iphone-8/specs/.

11. H. Sharma and S. Sharma, "A review of sensor networks: Technologies and applications," in *2014 Recent Advances in Engineering and Computational Sciences (RAECS)*. IEEE, 2014, pp. 1–4.

12. D. Konings, A. Budel, F. Alam, and F. Noble, "Entity tracking within a Zigbee based smart home," in *2016 23rd International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*. IEEE, 2016, pp. 1–6.

13. C. Badenhop, J. Fuller, J. Hall, B. Ramsey, and M. Rice, "Evaluating ITU-T G.9959 based wireless systems used in critical infrastructure assets," in *International Conference on Critical Infrastructure Protection*. Springer, 2015, pp. 209–227.

14. B. Greenstein, R. Gummadi, J. Pang, M. Y. Chen, T. Kohno, S. Seshan, and D. Wetherall, "Can Ferris Bueller still have his day off? Protecting privacy in the wireless era." in *HotOS*, 2007, pp. 1–6.

15. M. Ossman and D. Spill, "Project Ubertooth: An open source 2.4 GHz wireless development platform suitable for Bluetooth experimentation," 2014, accessed Sep 5, 2017. [Online]. Available: www.ubertooth.sourceforge.net.

16. BlueZ Project, "BlueZ: Official Linux Bluetooth protocol stack," 2016, accessed Aug 27, 2017. [Online]. Available: www.bluez.org.

17. Plugable Technologies, "Plugable USB 2.0 Bluetooth Adapter," 2016, accessed Dec 8, 2017. [Online]. Available: www.plugable.com/products/usb-bt4le/.

18. KimiNewt, "Pyshark," 2017, accessed Aug 30, 2017. [Online]. Available: www.github.com/KimiNewt/pyshark.

19. G. Valadon and P. Lalet, "Scapy: The Python-based interactive packet manipulation program and library," 2016, accessed Dec 8, 2017. [Online]. Available: secdev.org/projects/scapy

20. E. Skoudis, "Internet of things, Voice Control, AI, and Office Automation," 2016, presented at DerbyCon, accessed Dec 8, 2017. [Online]. Available: www.irongeek.com/i.php?page=videos/derbycon6/.

21. *Specification of the Bluetooth System*, Core Version 4.2, 2014, accessed Aug 30, 2017. [Online]. Available: www.bluetooth.com/specifications/bluetooth-core-specification.

22. A. Rose and B. Ramsey, "Picking Bluetooth Low Energy locks from a quarter mile away," 2016, presented at DEF CON 24, accessed Aug 30, 2017. [Online]. Available: www.media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/.

23. S. Jourdois, "BtleJuice: Bluetooth smart (LE) man-in-the-middle framework," 2016, accessed Aug 30, 2017. [Online]. Available: www.github.com/DigitalSecurity/btlejuice.

24. J. Slawomir, "A Node.js package for BLE using man-in-the-middle and other attacks," 2016, accessed Aug 30, 2017. [Online]. Available: www.github.com/securing/gattacker.

25. J. G. del Arroyo, J. Bindewald, and B. Ramsey, "Securing Bluetooth Low Energy enabled industrial monitors," in *Proceedings of the 12th International Conference on Cyber Warfare and Security*. Academic Conferences and publishing limited, 2017, pp. 167–176.

26. A. K. Das, P. H. Pathak, C.-N. Chuah, and P. Mohapatra, "Uncovering privacy leakage in BLE network traffic of wearable fitness trackers," in *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*. ACM, 2016, pp. 99–104.

27. A. Rose, J. G. Del Arroyo, J. Bindewald, and B. Ramsey, "BlueFinder: A range-finding tool for Bluetooth classic and low energy," in *Proceedings of the 12th International Conference on Cyber Warfare and Security*. Academic Conferences and publishing limited, 2017, pp. 303–312.

28. M. Versichele, L. De Groote, M. C. Bouuaert, T. Neutens, I. Moerman, and N. Van de Weghe, "Pattern mining in tourist attraction visits through association rule learning on Bluetooth tracking data: A case study of Ghent, Belgium," *Tourism Management*, vol. 44, pp. 67–81, 2014.

29. M. Zhou, Z. Tian, K. Xu, X. Yu, X. Hong, and H. Wu, "SCaNME: Location tracking system in large-scale campus Wi-Fi environment using unlabeled mobility map," *Expert Systems with Applications*, vol. 41, no. 7, pp. 3429–3443, 2014.

30. B. Bonné, A. Barzan, P. Quax, and W. Lamotte, "WiFiPi: Involuntary tracking of visitors at mass events," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops*. IEEE, 2013, pp. 1–6.

31. J. S. Atkinson, J. E. Mitchell, M. Rio, and G. Matich, "Your WiFi is leaking: What do your mobile apps gossip about you?" *Future Generation Computer Systems*, 2016.

32. C. Madrigal, "Tracking/Monitoring WiFi devices without being connected to any network," 2017, presented at Cyphercon 2.0. Accessed Dec 8, 2017. [Online]. Available: www.irongeek.com/i.php?page=videos/cyphercon2/.

33. M. Gruteser and D. Grunwald, "Enhancing location privacy in wireless LAN through disposable interface identifiers: A quantitative analysis," *Mobile Networks and Applications*, vol. 10, no. 3, pp. 315–325, 2005.

34. R. Rivest, "Chaffing and winnowing: Confidentiality without encryption," *CryptoBytes (RSA laboratories)*, vol. 4, no. 1, pp. 12–17, 1998.

35. K. Fawaz, K.-H. Kim, and K. G. Shin, "Protecting privacy of BLE device users," in *25th USENIX Security Symposium*, 2016, pp. 1205–1221.

36. J. G. del Arroyo, J. Bindewald, S. Graham, and M. Rice, "Enabling Bluetooth Low Energy auditing through synchronized tracking of multiple connections," *International Journal of Critical Infrastructure Protection*, 2017.

37. J. L. Brooks and J. A. Goss, "Security issues and resulting security policies for mobile devices," 2013, accessed Feb 11, 2018. [Online]. Available: www.dtic.mil/dtic/tr/fulltext/u2/a579735.pdf.

38. A. Liptak, "Strava's fitness tracker heat map reveals the location of military bases," 2018, accessed Feb 11, 2018. [Online]. Available: www.theverge.com/2018/1/28/16942626/strava-fitness-tracker-heat-map-military-base-internet-of-things-geolocation.

39. Raspberry Pi Foundation, "Raspberry Pi 3 Model B Specifications," 2016, accessed Aug 30, 2017. [Online]. Available: www.raspberrypi.org/products/raspberry-pi-3-model-b/.

40. nfarina, "Homebridge," 2015, accessed Aug 30, 2017. [Online]. Available: www.github.com/nfarina/homebridge/.

41. devbobo, "homebridge-platform-wemo," 2017, accessed Aug 30, 2017. [Online]. Available: www.npmjs.com/package/homebridge-platform-wemo/.

42. Wireshark, "OUI Lookup Tool," 2015, accessed Dec 28, 2017. [Online]. Available: www.wireshark.org/tools/oui-lookup.html.

43. GimliSonOfGloin, "Time inconsistency in Ubertooth host tools," 2017, accessed Dec 28, 2017. [Online]. Available: www.github.com/greatscottgadgets/ubertooth/issues/251.

44. Nerade, "Ubertooth BTLE freezes on CONNECT_REQ," 2017, accessed Dec 28, 2017. [Online]. Available: www.github.com/greatscottgadgets/ubertooth/issues/270.

45. K. Ognyanova, "Network visualization with R," *PolNet 2017 workshop*, 2017, accessed Dec 28, 2017. [Online]. Available: www.kateto.net/wp-content/uploads/2017/06/Polnet%202017%20R%20Network%20Visualization%20Workshop.pdf.

46. Radiocommunication Sector of ITU, "Isolation between antennas of IMT base stations in the land mobile service," 2011, accessed Dec 27, 2017. [Online].

Available:
www.itu.int/dms_pub/itu-r/opb/rep/R-REP-M.2244-2011-PDF-E.pdf.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
22–03–2018	Master's Thesis	Aug 2016 — Mar 2018

4. TITLE AND SUBTITLE

Pattern-of-Life Modeling Using Data Leakage in Smart Homes

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Beyer, Steven M., Capt, USAF

5d. PROJECT NUMBER

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-18-M-009

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

intentionally left blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This work investigates data leakage in smart homes by providing a Smart Home Automation Architecture (SHAA) and a device classifier and pattern-of-life analysis tool, CITIoT (Classify, Identify, and Track Internet of things). CITIoT was able to capture traffic from SHAA and classify 17 of 18 devices, identify 95% of the events that occurred, and track when users were home or away with near 100% accuracy. Additionally, a mitigation tool, MIoTL (Mitigation of IoT Leakage) is provided to defend against smart home data leakage. With mitigation, CITIoT was unable to identify motion and camera devices and was inundated with an average of 221 false positives per day that made it ineffective at identifying real events. Also, CITIoT was only able to recognize 8 minutes of 24 hours that the user was away from the smart home. This work closes by stressing the vulnerabilities presented through the demonstration of how an adversary can use CITIoT to crack a BLE lock and gain access to the home. Lastly, security recommendations are provided to defend against vulnerabilities presented in this work and create a safer smart home environment.

15. SUBJECT TERMS

IoT, Smart Home, Cybersecurity, Pattern-of-Life Modeling, Wi-Fi, Bluetooth Low Energy

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Barry E. Mullins (ENG)
U	U	U	UU	187	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, x7979; Barry.Mullins@afit.edu