Graceful Degradation in IoT Security

by

Dillon Bordeleau, Jared Grimm, and Roger Wirkala

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in Computer Science

by

December 2019

APPROVED:

Lorenzo De Carli

Craig A. Shue

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1651540.

Table 1: Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BLE	Bluetooth Low Energy
DHCP	Dynamic Host Configuration Protocol
CS/MACA	Carrier Sense Multiple Access Collision Avoidance
GRU	Gated Recurrent Unit
HTTP	Hypertext Transfer Protocol
ISN	Initial Sequence Number
IML	Interactive Machine Learning
IoT	Internet of Things
IP	Internet Protocol
IDS	Intrusion Detection System
LAP	Lower Address Portion
MAC	Message Access Control
NAP	Non-Significant Address Portion
NIST	National Institute of Standards and Technology
OUI	Organization Unique Identifier
RNN	Recurrent Neural Network
TCP	Transmission Control Protocol
UAP	Upper Address Portion
UDP	User Datagram Protocol

Contents

List of Figures

List of Tables

Abstract

As technology advances, personal privacy is constantly being eroded for the sake of convenience. This trend is evident in the consumer grade smart home and IoT devices industry, which has enabled access to potentially compromising information about users and their homes. Current security solutions ignore convenience by requiring the purchase of additional hardware, implementing confusing, out of scope updates for a non-technical user, or quarantining a device, rendering it useless. This paper proposes a solution that simultaneously maintains convenience and privacy, tailored for the Internet of Things. We propose a novel graceful degradation technique which targets individual device functionalities for acceptance or denial at the network level. When combined with current anomaly detection and fingerprinting methods, graceful degradation provides a personalized IoT security solution for the modern user.

We identified three main areas in our approach: fingerprinting devices to understand the composition of a users' network, anomaly detection to detect attacks on devices and network intrusions, and degradation to allow only user specified functions to remain active after an anomaly. First, we layer existing fingerprinting technologies to create a straight-forward device identification process. Then, we introduce an intrusion detection system through expanding the work of Kitsune, a lightweight machine learning intrusion detection system, to monitor the devices for abnormal or malicious behavior. We utilize previous research that highlights identifying network features for discovering device functionalities for an attack and leverage it as a defense mechanism. Next, we create a user-friendly traffic labeling platform to allow home users to label the extracted network features of any IoT device function within three minutes. We demonstrate that our random forest classifier

is able to detect and degrade user-trained functionalities with an average accuracy rate of 94%. The user is provided with information regarding potential privacy risks if a specific function is compromised, and given the option to degrade or allow it in lieu of an anomaly. Finally, we consolidate fingerprinting, anomaly detection, user training and degradation into an intuitive, straightforward User Interface, the GDI Security Suite.

Our suite enables a user to monitor all devices on their network and train normal device and network behaviors. Upon detecting an anomaly, we will follow the specified risk tolerance as to which functions the user wishes to block, and consequently drop traffic related to that functionality as well as other foreign activity, while still allowing other desired functions to remain actionable. Lastly, we notify the user of such events, making them aware of their current state of security, and provide the ability to dynamically update their risk profile.

Chapter 1

Introduction

As technology becomes more ubiquitous, and people are in search of convenience aids, the Internet of Things (IoT) industry continues to automate more aspects of everyday life. IoT devices are electronic devices that can be remotely controlled, monitored, or that perform remote commands. IoT devices span from consumer grade devices controlling home appliances, to financial applications, energy and water sensors, health care devices, wearable technology and transportation [35]. With 26.6 billion IoT devices connected in 2019 and an estimated 30.7 billion to be connected in 2020 [52], IoT is one of the largest growing sectors of technology. Each of these devices is vulnerable to attack or exploit, which could result in an invasion of privacy, loss of data, or even physical harm. In 2020 it is predicted that more than 25% of cyber attacks will involve IoT devices [14]. Ironically, 72% of consumers would own IoT devices to feel safe [7], yet 48% of them were unaware that these devices can be used to conduct a cyber attack [12]. Consumers lack an understanding of what data these devices hold, how devices use their data, and what could happen if their data is exploited by attackers.

Companies who do not specialize in electronics do not take the necessary pre-

cautions when adding connectivity to their products. They outsource the "smart" development to IT experts without understanding if a selected third party is specifically an IoT expert. When security and privacy issues arise in these cases, the manufacturer is of no help to the user, since they lack knowledge of how their product was developed [13].

Common security steps in consumer-based devices have been composed of fingerprinting devices, anomaly detection, and mitigation. Currently, mitigation only consists of quarantining the device off of the network, or asking the home user to implement confusing firmware updates [36]. Ultimately, most security solutions simply notify the user that there is something wrong with a device, without explaining its implications, or suggesting a clear resolution.

Aside from technical solutions, government legislation has attempted to solve the lack of coherence among the IoT community when it comes to security standards. Most recently introduced, is the IoT Cybersecurity Improvement Act of 2019 which mainly requires National Institute of Standards and Technology (NIST) to issue recommendations, and enforces government and vendors to follow those recommendations, as well as disseminate discovered vulnerabilities sooner and more widespread. However, there will be an inherent gap in time between the release of new devices, and when NIST has completed ample penetration testing to make recommendations, or discover new vulnerabilities to warn companies about. This legislation only calls for guidelines, and it will remain up to the vendor to adopt best practices.

A large problem that stems from the sheer size of the IoT industry is the lack of comprehensive, real world test traffic for researchers to access and analyze. Most devices and security solutions are tested, or trained on a small sample of data in an isolated environment, for a miniscule number of devices. In a survey by

Boutaba et al. the authors "observed that numerous works relied on synthetic data, particularly in network security. Synthetic datasets are usually simplistic and do not truly reflect the complexity of real-world settings" [10]. Since there is a lack of common knowledge regarding the features and normal behavior of the majority of IoT devices, it is hard to develop a solution that is scalable to all devices.

Despite recent advances in technology and policies regarding IoT security, we have identified lack of user education and lack of mitigation strategies as consistent flaws in modern IoT security approaches.

1.1 Contributions

We offer a solution to the problems presented by developing a holistic security suite that a) identifies devices on Wi-Fi networks, b) monitors network traffic for anomalous behavior, and c) gracefully degrades device functionalities. We develop a scalable platform that labels traffic associated with particular device functionalities by leveraging machine learning. From these labels, we train classifiers to gracefully degrade user specified functionalities in real-time with minimal user input. Further, we present risk profiles associated with categorical bins of device functionalities to educate end users.

Education:

As previously mentioned, nearly 50% of consumers are unaware that the devices they use everyday in their homes can be turned against them by cyber criminals. Our security platform informs the home user about their potential vulnerabilities and perceived risk posture. To accomplish this we show the user what devices are on their network at any time, what functionalities and features these devices possess, how different functionalities could be used maliciously or intrude on the

user's privacy, and when and which device is under attack. This information will be portrayed through our Graphical User Interface (GUI) using best Human Computer Interaction (HCI) practices.

By providing this information, we are arming users with the tools they need to consider what is an acceptable level of risk for their security and privacy and how introducing IoT devices will affect their risk posture. Ultimately, a more alert and aware community will make for a safer and more secure one.

Degradation:

Our main innovation is a mitigation technique customized by the user, to allow for graceful degradation. Graceful degradation is the practice of maintaining core functionality, while isolating or disrupting nonessential functions to avoid complete failure, or in our case, further privacy breaches. Our degradation strategy will also block anomalous traffic that may be new malicious activity. For example, a Wi-Fi camera sending email is not a feature that a user would train or expect to occur, resulting in it being identified as anomalous traffic and dropped in degradation.

Devices have a wide array of applications and in some cases, they may not pose as a security threat to the home user. For example, a user installs a security camera placed outside the house to look for intruders. The owner would prefer to continue to record video and audio, yet still be notified of the devices compromise and prescribed possible fixes. However, if a camera is used to monitor a child, owners may prefer to immediately stop video and audio recording if the device is compromised. This approach allows a user to own their privacy and security while maintaining the convenience that persuaded them to purchase the device in the first place. Giving consumers the peace of mind that their security and privacy will be safeguarded, while still being able to use their devices to their full potentials, will begin to build a reputation of trust in IoT devices. We believe graceful degradation can provide

an enormous shift in the IoT community.

Interactive Supervised Machine Learning for Traffic Analysis:
With the ever growing number of IoT devices, no current lab-trained model will be scalable enough to continue to work for all new devices. We propose an extensible interactive machine learning model, which prompts a user to easily train and label exact functionalities of any IoT Wi-Fi device. This technology has not been applied to device functionality traffic classification before. With this method, any device can be trained and its individual functions can be isolated for network traffic analysis, or in our case, degradation. The user's role is to label what type of functionality, such as streaming, audio or video, or simple on/off commands is associated with the traffic, as well as denote exact time frames when that functionality is occurring within a one hundred and eighty second training interval. This labeled functionality can then be stored, with the user's approval, and utilized by the scientific community or other consumers.

Chapter 2

Background

2.1 IoT Protocols

Smart homes are comprised of a multitude of devices working alongside each other in a collaborative mesh of devices, all of which is facilitated by protocols. Protocols tie all devices on a network together, facilitating the flow of information, and providing ways for devices to communicate with one another, the internet, and the user. In order to perform fingerprinting, anomaly detection and degradation, understanding how these devices communicate is important. IoT devices have a large array of functions, sensors, data streams and varying ways to communicate with other devices. Due to this wide range of data transfer rates and power sources for IoT devices, several data-link protocols have been created to best facilitate communication to meet device requirements. The most widespread of these protocols are Wi-Fi, Bluetooth Low Energy (BLE) and Zigbee.

2.2 Current Device Fingerprinting Techniques

Device fingerprinting techniques are used to determine the types of devices present on a network. IoT fingerprinting is of particular interest in the cybersecurity field because adversaries routinely exploit Internet-connected devices while considering their type. Similarly, different types of devices have varying sets of behavioral capabilities and risk profiles. For instance, a compromised, internet-enabled camera would likely intrude on user's privacy more so than a compromised, Internet-enabled smart plug would. It is easier to gain information about a place and its inhabitants through a camera live feed rather than knowing if an outlet was drawing power or not. Device fingerprinting techniques are used to determine the types of devices present on a network. Current research titles such as AuDI, ProfilIoT, and IFRAT, use a handful of specific network traffic features to ascertain a device's manufacturer, operating system, or categorical classification of capability. Some open source APIs (application programming interfaces), such as `Fingerbank` [18], passively query particular network traffic signatures across databases of historical device identifications.

Here, we discuss several existing approaches to fingerprinting that operate without reference to historical databases of known network traffic signatures. AuDI is a system for quickly and effectively identifying the types of devices on IoT networks by analyzing network communications [32]. AuDI models the periodic communication traffic of IoT devices using an unsupervised learning method to perform identification [32]. ProfilIoT, a machine learning approach for IoT device identification based on network traffic analysis uses HyperText Transfer Protocol (HTTP) packet properties to differentiate between IoT device traffic and non-IoT device traffic [38]. ProfilIoT then enriches source and destination IP addresses and ports with publicly available data from Alexa Rank and GeoIP to categorize device type [38]. IFRAT,

an IoT field recognition algorithm based on time-series data, observes sampling rates of various network features, performs clustering with the self-organizing map machine learning algorithm, then determines IoT field and specific device type [25].

There are observed disadvantages to machine-learning-based fingerprinting approaches such as AuDI, ProfilIoT, and IFRAT. First, such techniques require time for training. With no historical signature data to rely on, machine-learning-based techniques must extract, organize, and analyze network features as observed in real time. AuDI, for instance, will rarely perform device identification in less than thirty minutes [32]. Second, such techniques require that an adversary does not disrupt the learning stages. Introduction of spoofed network data during learning stages could easily disrupt a machine-learning-based fingerprinting technique and severely limit an overall confidence level.

`Fingerbank` [18] is an existing repository for data that can be used to fingerprint Internet-enabled devices. `Fingerbank` [18] takes a Media Access Control (MAC) address and Dynamic Host Configuration Protocol (DHCP) fingerprint as parameters in attempting to identify device type. A device exposes its MAC address, a unique identifier for its network interface, when it connects to a network; individual manufacturers use particular ranges of MAC addresses for their products. Similarly, when requesting an Internet Protocol (IP) address assignment with DHCP, a device asks for certain DHCP options such as Domain Name Server (DNS) server, or default gateway,and the sequential ordering of these option requests can be indicative of device type. `Fingerbank` [18] uses patterns in MAC addresses and DHCP option requests to make a best guess at specific device type and manufacturer. Finberbank does not directly interact with the device it is attempting to fingerprint, rather it relies on a user interacting with a device to supply a MAC address and DHCP fingerprint. When `Fingerbank` [18] receives these values, it queries them across historical

records in order to ascertain device type in a quickly evolving IoT landscape.

`Nmap` [30], or the Network Mapper, attempts remote operating system detection using Transmission Control Protocol (TCP)/IP and User Datagram Protocol (UDP) stack fingerprinting. `Nmap` [30] sends a series of TCP and UDP packets to the remote host and examines the data that is returned [30]. After performing a number of tests, for example, TCP Initial Sequence Number (ISN) sampling, TCP options support and ordering, IP ID sampling, and initial window size checking, `Nmap` [30] compares results to a database of more than two-thousand known operating system fingerprints [30]. Each fingerprint includes a textual description of the operating system and a classification that provides vendor name and device type [30]. `Nmap` [30] differs from `Fingerbank` [18] in that it directly interacts with a device that it is attempting to fingerprint.

`Nmap` [30] directly interacts with devices through port scanning, which generates large quantities of traffic and can slow or disable weaker devices on less robust networks. Further, target devices can be configured to disable port scanning and limit the accuracy of `Nmap` [30] operating system detection. Lastly, `Nmap` [30] operating system detection requires sudo privileges on the host device. With this, `Nmap` [30] differs from machine-learning-based techniques, such as AuDI, ProfilIoT, and IFRAT, and signature-repository-based techniques, such as `Fingerbank` [18], in the way that it gathers information about a device. The former approaches can perform passively on a network, while `Nmap` [30] requires active engagement with target hosts.

`p0f` [53], a passive fingerprinting tool distributed in some versions of Kali Linux, offers a passive alternative to `Nmap` [30]. `p0f` [53] utilizes an array of purely passive traffic fingerprinting mechanisms to identify the players behind any incidental TCP/IP communications without interfering in any way [53]. Of note, `p0f` [53] is

highly scalable and offers extremely fast identification of the operating system and software on both endpoints of a vanilla TCP connection, especially in settings where `Nmap` [30] probes are blocked, too slow, unreliable, or would simply set off alarms [53].

2.3 Current Anomaly Detection

Anomaly detection refers to the capability of identifying rare events that differ from the expected or observed status quo and subsequently raise suspicion. In the cybersecurity space, anomaly detection is a common technique utilized by intrusion detection systems (IDS). Recently, machine learning and Artificial Intelligence (AI) techniques have made anomaly detection much more sophisticated and reliable when properly trained and deployed. The majority of anomaly detection techniques in IDSs operate on network traffic, examining different statistics and extracting metadata from network packets. IoT, however, broadens the scope of anomaly detection. It can still be applied in the traditional sense to network traffic, but it can also be applied on user actions as inferred through IoT device states.

Within network based anomaly detection, there are several strategies of varying complexity and effectiveness. Approaches that aim for extremely low overhead use simplistic algorithms that are not adequate to address the rapidly developing threat landscape they face [51, 49]. Signature based approaches [44, 15, 42, 39] rely on having advance knowledge or a database of known attacks that is constantly kept up to date. Attacks may be device specific, and with the constant influx of new devices, these models must evolve. Signature-based models fail at detecting zero-day attacks as those vectors are not yet known. As attackers deploy new, more sophisticated attacks, these systems have no mechanism to detect the initial infections.

IDSs can either be network-based or host-based. In the case of IoT, however, host-based IDSs present several issues. First, deploying a host-based IDS requires access to device software and the ability to inject third party code into these devices, which presents a significant vulnerability. [46, 45]. This change would require more development work on behalf of nearly all device manufacturers and the creation of an industry standard to which they would need to comply with for this approach to be successful. Given the sheer size of the IoT market and its competitive nature, this would be a monumental feat and is unlikely to happen. Second, the multitude of IoT devices presents a serious scalability issue. Every device added to the network would have to be flashed with the IDS software [11]. A main benefit of IoT comes in the ease of deployment and ubiquity. To maintain usability, approaches that are less intrusive would be preferred.

Some approaches leverage blockchain technology in attempts to become more resilient to adversarial attacks during the training phase in which attackers can inject traffic to have the IDS learn malicious traffic as benign traffic. This requires additional blockchains per IoT model/firmware [23]. The publishing of additional chains limits the scalability of this approach, especially in the consumer space where several models, manufactures, and devices interact.

Another approach is Artificial Neural Networks (ANNs). One class of ANNs is a recurrent neural network (RNN). The main advantage of RNNs over ANNs in terms of anomaly detection is the ability to have stored states to examine temporal data more deeply. A Gated Recurrent Unit (GRU) is a less computationally expensive flavor of an RNN, requiring less training data. Nguyen et al. [38] leverage GRUs in a self-learning, distributed IDS for IoT networks. While this system achieves a high level of success, recent research has suggested improvements.

One issue plaguing modern IoT security and IDS is the low overhead available

in the system contrasted with the high performance demands of sophisticated ANN IDS methods. To address the gap between performance demands and available computaional power, Kitsune [34] uses an ensemble of autoencoders. Autoencoders are ANNs that are able to learn efficient data encodings by reconstructing their inputs, essentially learning the identity function of the original data. In order to maintain low overhead costs, the depth of the ANN is limited to three layers. Deep, or layered, ANNs can learn much more complex concepts than shallower networks but become computationally expensive to train and execute. Furthermore, autoencoders can be trained in an unsupervised manner, meaning that labeled training data is not required. Thus, the costly process of labeling a data set, which can be massive when examining network traces, can be avoided.

Kitsune provides a relatively lightweight ANN anomaly detection approach through an ensemble of autoencoders. The low cost of autoencoders, accompanied by benefits such as online, unsupervised learning, differentiates this solution. The open source data and code base for Kitsune streamlines the deployment and customization of the system for nearly any configuration. Implementing Kitsune and taking advantage of their open source code-base and data sets will satisfy the IDS requirements for our IoT security platform.

2.4 Current Machine Learning Techniques for Network Traffic Classification

Machine learning is a growing area of Computer Science with seemingly endless applications. Relative to the context of our research, machine learning provides the capability of network traffic classification. Network traffic is made up of packets, each containing several features related to their origin, destination, and content.

In analyzing numerous packets, there is no apparent correlation between a devices functionality and its network packet. Machine learning allows for features to be isolated, weighted based on uniqueness, and compared against each other, to form patterns and "clusters" of similar traffic. Machine learning algorithms can vary in two main ways. The first is how data is fed into a model to train the classifier. There are three different methods of labeling the data to build a foundation off which to match new unseen traffic to a particular cluster. The second is, which features of the network packets should be extracted in order to form the desired clusters. We explore the current research field in relation to these two areas in order to understand which data labeling and feature extraction approaches best fit our need for device function clustering.

The three main machine learning training paradigms that can be utilized for classification are supervised, unsupervised, and semi-supervised. Supervised learning is a paradigm where an input set of data is completely labeled with the correct classification or value. Supervised learning is useful for problems concerning classification and regression. Due its requirement of complete correctness of input data, supervised learning is highly accurate, but the overhead necessary to label all of the data is often large and usually requires an expert to label the data accurately. Unsupervised learning absorbs all unlabeled data and learns to infer structures from the baseline training data. Unsupervised learning also is applicable for clustering and association, being able to differentiate whether a data point closely follows a group of data previously ingested. Unsupervised models have less initial overhead than supervised, due to the lack of necessity of labeling all of the input. However unsupervised models are only as good as the quality of input data that is used to train them. To combine the strengths and weaknesses of both, semi-supervised learning is an approach that labels small amounts of data to train algorithms to accurately

and precisely extrapolate associations and structures in unlabeled data. Numerous methods across all three paradigms have been tested to address the problem of network traffic classification [10]. However, these experiments seek to classify protocols and types of network traffic rather than classify which traffic corresponds to specific device functionality.

Specific to network traffic classification, the four main areas for feature extraction in past research include port number, packet payload, host behavior or flow features. Port number was an effective early traffic classifier as different protocols often were restricted to certain port number. However, relying solely on port numbers has become ineffective due to the use of dynamic port negotiation. Meanwhile, adding port number alongside other features has shown to boost classification scores [10]. Next, the payload of network packets contains application signatures which are indicative of the purpose of the traffic. With modern security concerns, however, a majority of these payloads are encrypted and unavailable for inspection. For those that are available, searching the payload incurs a high overhead, and the number of unique signatures needed for identification may grow rapidly, rendering this method difficult to scale and maintain. Host-based classification relies on a monitoring system being placed in a network to examine criteria, such as the number of hosts on the network, to predict the classes of interest. It avoids using direct ports or payloads and instead monitors how a network is interacting with a group of hosts, such as mail servers, or peer to peer connections. By analyzing how many hosts on the network are involved with the communication, how many ports are active and how often different hosts communicate, it is possible to derive types of traffic such as mail services and video downloads. The final feature space is known as flow-based. A network flow is a selected sequence of consecutive packets during a connection. Within this sequence, countless features can be quickly isolated from

packets in transit. Example features include direction of flow, source and destination IP addresses and ports, inter-arrival time, average packet length, and number of packets per flow. Flow-based classification "has the potential to overcome numerous limitations of other techniques, such as unregistered port numbers, encrypted packet payload, routing asymmetries, high storage and computational overhead. However, it remains to be evaluated if flow feature-based classifiers can achieve the accuracy of payload-based classifiers" [10]. In this work, we took a combined approach of testing several feature spaces in order to rapidly identify device functionality without disrupting normal operation speed or performance.

2.5 Interactive Machine Learning for Network Classification

Interactive Machine Learning (IML) was first introduced in 2003 as a way to "allow users to train, classify, view, and correct classifications" [17]. This work focused on having a user analyze images and create classifications based on the images alone. Then a classifier would be created from the selected images and display feedback. The user could either refine the classifier by adding more manual classification or export the classifier. This demanded that the user knew exactly what the classifier was supposed to output. IML had the benefits of offloading extensive training time from technical experts, as well as reducing computation time of model training, as models could be trained by millions of non-technical users simply by clicking images or typing words. This methodology has been expanded upon and used in practice across different areas of study. It is most famously known from Google's Recaptcha, an authentication method that prompts users to interpret scrambled text or images containing certain items. Through the interpretation as to whether

a picture contains an object like a bus, the millions of data points that have been entered have helped advance systems such as driver-less cars. Classifying static images and text has expanded to streaming data within the past few years.

In 2006, active learning was introduced for network protocol analysis [5]. This patented model extracts features of the traffic and generates examples of it, then re-trains its learning model to better label traffic in the future, but once again relies on the user's accuracy in correct labeling. The combination of IML applied to streaming data and IoT devices is on the cutting edge of research. In 2019 the technology was used for activity recognition (standing, walking sitting or lying down, as well as room occupation) of a user based on IoT device streams [47]. In this case, a prompt for a user to label activity is time sensitive, as a user may forget what past actions they were doing at a given time. This study investigated active learning by asking a user to label streams for one activity from several different devices at once. Not all of these device streams may be directly related to the activity of the user. Our approach directly labels specific and individual functions of a single device, at the time the user chooses, allowing them to know exactly which function they are entering.

Our approach leverages past methods of allowing a user to label data, in order to train a model which will then handle all unlabeled data. We apply this to device functionality traffic, where other dynamic noise will be present. Contrary to the previously discussed methods, our learning model does not rely on the end-user having knowledge of the correct output of the classifier. Rather the user labels provides a friendly name to the traffic of a functionality. This label allows the user to understand which function they are choosing to degrade or allow.

2.6 Current Degradation Attempts

When a system is compromised through a cyber-attack, we would like critical infrastructure and functionality to remain operable. The majority of people introduce IoT devices into their smart homes for the purpose of convenience and usability, automating everyday tasks and generally making life easier. Therefore, consumers would prefer to maintain as much functionality as possible when an IDS detects an attack. Incorporating graceful degradation will allow our network to maintain partial functionality while compromised or under attack.

Removing all device functionality nullifies the benefits of adding these devices to the home. For example, Minim is a company providing an AI based IoT security approach to monitor devices and traffic on your network. When a device is infected or an attack detected, Minim's response is to quarantine the source device from the network [33]. This renders the device useless until the attack is resolved. No functionality of that device may be accessed.

Another comprehensive home security solution, the BitDefender Box, acts as a network gateway and IDS. How Bitdefender handles security breaches, however, is not clear. The company simply states that the box "shuts down traffic" [36]. This language seems to imply quarantining devices or creating an IP blacklist. Nowhere in their platform does the notion of degradation get mentioned.

Graceful degradation can be approached at multiple levels of complexity. The simplest method would be to access publicly-available application programming interfaces (APIs) or settings menus programmatically [26, 4]. If this functionality can be accessed through menus and APIs it is reasonable to assume that an adversary could do the same, effectively overturning the degraded functionality.

A more complex and robust approach to IoT degradation identifies and targets

traffic associated with specific device functionality. Accar et al. [3] present a novel attack that utilized machine learning techniques to profile traffic of smart home device communications even when the traffic was encrypted. The paper proposes a passive adversary that is listening to network traffic as part of a multi-layer privacy attack. By identifying traffic from devices such as light bulbs and locks, an adversary can determine when the user is home and even what room they could be in. Although this is framed as an attack, the techniques proposed can be leveraged as a defense mechanism. By identifying and dropping traffic associated with device functionality, graceful degradation in IoT-enabled smart homes can be achieved.

The above is an example of a privacy attack [3], but attackers are also interested in harnessing the power of combining IoT devices into botnets [15]. Such an attack could have no tangible effects on the end user. The user might not ever know they were ever compromised.

Chapter 3

Approach and Methodology

In this chapter, we discuss our fingerprinting, intrusion detection, graceful degradation, and integration capabilities. Particularly, we present a novel, scalable approach for the graceful degradation of IoT device functionalities.

Approach 1: In order to enable graceful degradation, our team proposed a system consisting of of three main modules with a manager handling integration communication between modules. Our IDS is anomaly detection-based, leveraging machine learning to differentiate benign and malicious network traffic. Once an anomaly is triggered it sends a message to the manager with the MAC addresses that raised the flag. The MAC addresses are then sent to the fingerprinting module. This module is responsible for identifying and tracking all of the IoT devices present on the network. Given a MAC address, the fingerprinting module will be able to identify what device triggered the anomaly. Fingerprinting devices also allows us to understand what capabilities are present in the smart home and translate these capabilities to security and privacy vulnerabilities. These attack vectors are presented to the user, who can decide what functionality to degrade in order to custom tailor a risk profile that is acceptable to them. The degradation of features

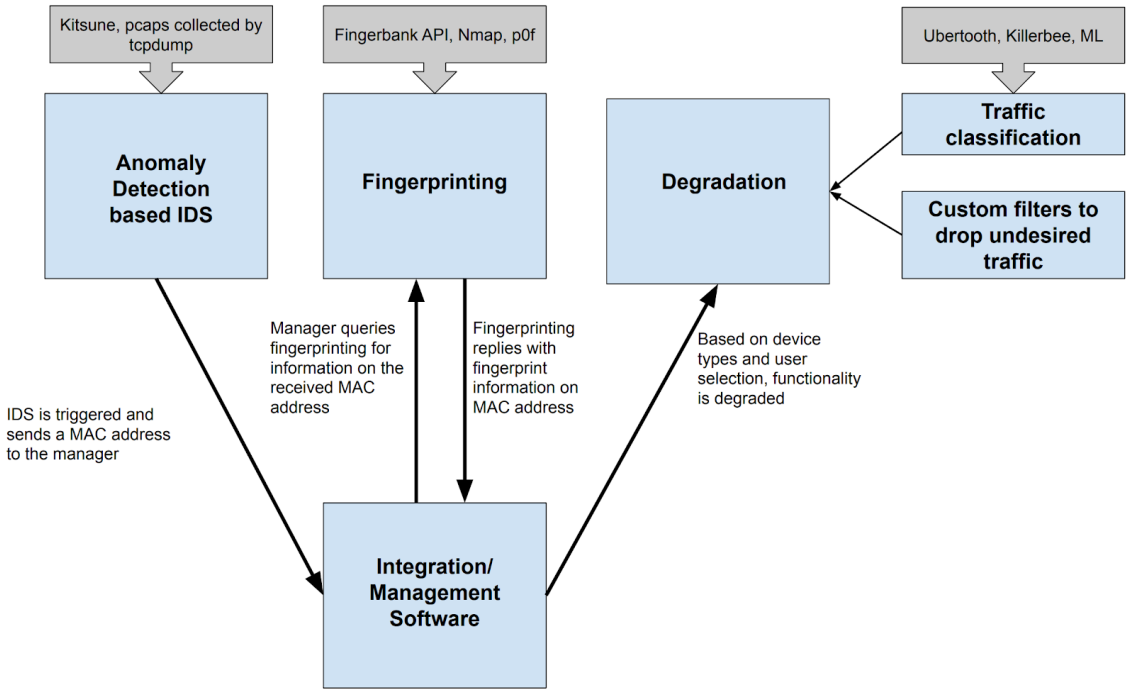

Figure 3.1: High level system design and service call relationship layout

will be accomplished through traffic classification across multiple protocols such as Wi-Fi, Zigbee, and Bluetooth.

3.1 Enabling Sniffing of Alternative Protocols

While the testbed infrastructure makes it trivial to sniff for IoT Wi-Fi traffic associated with the `mqpinet`, the other two protocols of BLE and Zigbee require further hardware. Sniffing these two protocols is necessary for capturing the traffic that occurs between these devices and determining which packet attributes match device functionality.

In order to perform such BLE sniffing, our team chose Ubertooth One Hardware and `Ubertooth` linux software, due to its low cost, complete documentation and portability with pcap files and Wireshark integration. Ubertooth One hardware

was purchased and the software and was installed following the github instructions [21]. Once the software was installed, the Ubertooth One device was attached to the computer and bridged into the virtual environment. Ensuring that the device was found using *ubertooth-util -V*. Once the device was properly recognized, we followed the guide [22] for flashing the firmware onto the device. Our device was now capable of sniffing BLE traffic.

In order to sniff Zigbee traffic, we chose the ApiMote v4 beta developed by Riverloop security, as it comes pre-flashed with Killerbee firmware, however an extra antenna was gathered from a router to improve range and performance of the APiMote. We also installed `Killerbee`, following the instructions [50]. Once Killerbee was installed, and hardware was detected, we were able to sniff Zigbee traffic.

3.2 Change in Approach

We originally considered Bluetooth Low Energy and Zigbee enabled devices to cover the full scope of common protocols in the IoT space for fingerprinting and degradation. However through interacting with the devices, we discovered a large portion of device functionality eventually communicates over Wi-Fi. We decided we would no longer investigate the Bluetooth and Zigbee protocols in the scope of this project for the following reasons:

- Nearly all IoT devices have a transport layer of either TCP or UDP and communicate over Wi-Fi in some capacity, regardless of use of alternate protocols [43]

- The majority of Zigbee and BLE devices have hubs that connect to Wi-Fi and all function traffic is transmitted from a users phone to said hub. In fact Samsung SmartThings, who is a producer of Zigbee devices, clearly states

Figure 3.2: Ethernet traffic from Hue Bulb Zigbee device reveals device functionality

> "An active Internet connection to the Hub is always required to manually control your devices and SmartApps via the SmartThings mobile app." So any controls sent from your smart-phone to the hub to be relayed to the devices occurs over Wi-Fi connection. So by sniffing all Wi-Fi traffic, we are able to identify functionality in Zigbee devices such as Phillips Hue Bulbs.

Next, BLE end devices can only be connected to one master device like a cell-phone, so the attacker would have to be present to intercept the first Bluetooth packet in order to connect to it. In this case the home user would be suspicious if they cannot connect and disconnect it. To exploit BLE or Zigbee directly, the attacker would have to be within 10 to 100 meters of the devices, while 30 m is usually the cutoff for a reliable connection. Those connection ranges are also recommended for line of sight with the device [43]. So an attacker would have to be relatively close to a users' house and specifically target their devices, rather than find them on the Internet from a remote location.

Our original approach to investigating the degradation of a few functions of Zig-

bee and BLE devices involved capturing the traffic, analyzing the flow rate, and setting a threshold. This approach was modeled after the Peekaboo attack vector [3]. However we discovered with Wi-Fi devices, we are able to track specific connections with enough constant features that it is possible to create ML classifiers to degrade functionality without looking at the traffic in static snapshots to compare to a threshold. Rather, we can train a model to map certain device functionality to a large number of features. Exploring a solution to automate functionality degradation, rather than statically proving degradation on small features over these protocols, will be more fruitful in both the engineering and IoT security space.

Moving forward, our contributions of degradation will not be diminished by focusing only on Wi-Fi connections, and introducing the ability for any home user to label, train and degrade any functionality of a device as they choose, making it truly graceful.

3.3 New Approach

The proposed updated system consists of six main modules with a manager handling integration and communication between tools and the user. An addition to our new approach was the addition of a novel user-labeled ML model for device functionality. The testbed, management system and anomaly detection methods remained the same. The major difference is that only Wi-Fi devices will be fingerprinted, with the same tools as before.

The user will be able to access the GUI through a web server hosted on the Pi and discover all Wi-Fi devices on their network. The fingerprinting will display guesses and allow the user to name the devices IP address a “Friendly Name” based on the fingerprinting results. They will be able to identify and train new machine learning

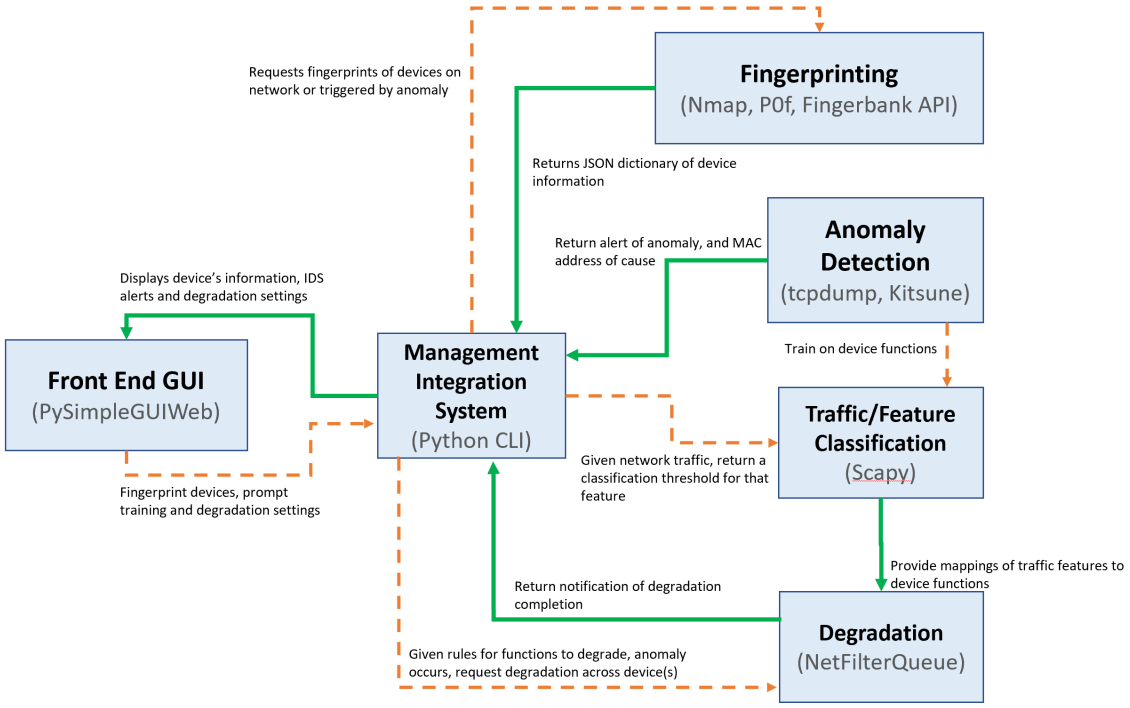

Figure 3.3: Updated high level system design and service call relationship layout

models for degradation based on each feature of the device. As the user is prompted to test a new function, the network packets are sampled through netfilterqueue, and unique identifying features are extracted to form a Random Forest Model. The user will be able to select devices and trained functions of those devices to pinpoint what functionality they would like to degrade.

When an anomaly is detected, the corresponding Degradation IP table rules to which functions the user has selected to degrade will be turned on and that traffic will be blocked.

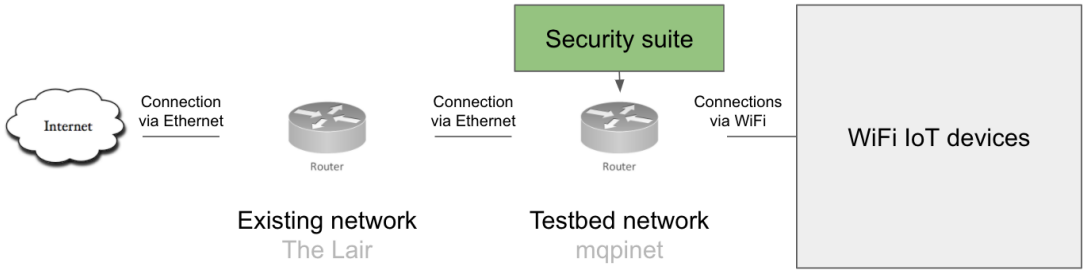

Figure 3.4: An overview of the testbed network architecture.

3.4 Configuring a Standalone, Wireless, Test-bed Network with Raspberry Pi

In order to perform our experiments without sniffing private traffic from the university, or degrading other users' devices, we created our own private test network. This network will act like any other Wi-Fi Network; however, only our IoT devices will be connected to it and the only traffic passing through will be the traffic of interest to us from these devices. This testbed will serve as the base for all four aspects of the project mentioned above. The team configured a Raspberry Pi as an access point in a standalone network using NAT (Network Address Translation) [19]. The inbuilt wireless features of the Raspberry Pi 3 can be used to host a standalone network with integrated DHCP software.

First, the team used Ethernet to connect the Pi to the existing network present at the apartment where testing was being conducted. With this, the Pi was connected to the existing network using `eth0` as opposed to `wlan0`. Second, the team installed `dnsmasq` and `hostapd` software and assigned the static IP address `192.168.4.1` to the Pi. Third, the team configured `dnsmasq` such that the Pi would provide IP addresses between `192.168.4.2` and `192.168.4.20` with a lease time of 24 hours. Fourth, the team configured `hostapd` such that the Pi would host a 2.4 GHz wireless

network called `mqpinet` using `wlan0`. Finally, the team added a masquerade for outbound traffic on `eth0` and saved corresponding rules.

After rebooting the Pi and starting `dnsmasq` and `hostapd` services, the team began connecting IoT devices to the `mqpinet` wireless network. As devices connected to the `mqpinet` network, `dnsmasq` would assign IP addresses on the existing network in the range `192.168.4.2` to `192.168.4.20` as previously configured. In this way, all wireless traffic originating from the IoT devices was 1) received by the `mqpinet` on the Pi via `wlan0` and 2) forwarded to the existing network via `eth0`. With this, the team could swiftly differentiate between traffic from the IoT devices involved in the experiment and traffic from devices on the existing network not involved in the experiment. From the Pi, the team sniffed `wlan0` to only intercept network traffic interacting with devices on the `mqpinet` testbed network. Even still, the team could sniff `eth0` to intercept all network traffic from both `mqpinet` and the existing network.

3.5 Device Gathering

The team gathered a range of IoT devices for the project. Some devices offered simple on or off functionalities, such as smart plugs, while some devices offered complicated audio or video streaming functionalities, such as the Amazon Blink Wi-Fi cameras and Amazon Alexa Echo Dot.

- Amazon Alexa Echo Dot
- TP-Link Wi-Fi smart plug
- Raspberry Pi 3
- Amazon Blink Wi-Fi camera

- Amazon Blink Wi-Fi camera
- Etekcity Wi-Fi smart plug
- Particle Photon
- Google Chromecast
- Philips Hue Bridge

3.6 Fingerprinting Configuration

The selected devices were then connected to the Pi testbed network which was acting as a router for `mqpinet` at `192.168.4.1`. Devices joined this network just like any other Wi-Fi network. With active devices on our network, the following necessary step was to identify the devices on the network through the Raspberry Pi itself. Fingerprinting devices is an important step in consumer security as it allows the home user to be aware of what devices are on their network, incase an attacker introduces a rogue device. Fingerprinting also allows us as the security system to analyze and store information on network traffic to human readable devices, and to gain insight into what types of traffic there may be, depending upon which device is generating that traffic. Fingerprinting is a well studied and explored research area, so we focused on layering pre existing tools, rather than developing our own stand alone solution. The team centered device fingerprinting capabilities around functionalities offered by the `Fingerbank` [18] API and `Nmap` [30]. By sniffing `wlan0` on the Pi testbed, the team could filter IoT traffic routed through the Pi. Fingerprinting capabilities were housed and executed on the Pi itself, eliminating the need for additional hardware.

First, the team authored `fingerbank_handler.py` to extract relevant packet features and interact with the `Fingerbank` [18] API. With Scapy sniffing, the team used filter `udp and (port 67 or port 68)` on the interface `wlan0` to intercept packets on `mqpinet` with a DHCP layer. When a packet with a DHCP layer was intercepted, the `fingerbank_handler.py` extracted 1) the DHCPv4 fingerprint and 2) the MAC address of the device from which the packet originated. To determine a DHCPv4 fingerprint, `fingerbank_handler.py` examined DHCP options from the DHCP layer and extracted the value of the `param_req_list` key. To determine a MAC address, `fingerbank_handler.py` examined the Ether layer and extracted the `src` MAC address. Once `fingerbank_handler.py` had extracted a DHCPv4 fingerprint and MAC address, it queried the `Fingerbank` [18] API to obtain a JSON package denoting Fingerbank's best guesses at manufacturer, operating system, and type for the device from which the examined traffic originated. It is important to note that `fingerbank_handler.py` only queried the `Fingerbank` [18] API if an extracted MAC address had not been seen before. `fingerbank_handler.py` kept record of API queries that were previously conducted by uniquely identifying JSON packages by the MAC addresses of devices whose extracted features were being queried.

Second, the team authored `nmap_handler.py` to 1) map IP addresses to MAC addresses for hosts on the `mqpinet` network and 2) perform `Nmap` [30] operating system detection on discovered hosts. To map IP addresses to MAC addresses, `nmap_handler.py` executed `nmap -sP 192.168.4.0/24` with `sudo` privileges and used regular expression matching to sequentially extract and pair IP addresses and MAC addresses from the command's output. To perform `Nmap` [30] operating system detection on a discovered host with IP `host_ip`, `nmap_handler.py` first executed

```
$ nmap -O -Pn --host-timeout 3m host_ip
```

with `sudo` privileges. Second, `nmap_handler.py` used regular expression match-

ing to extract a MAC address, manufacturer, device type, and operating system version and details from the command's output to forge a custom JSON package. `nmap_handler.py` also kept record of OS detection scans that were already conducted on particular hosts, uniquely identifying JSON packages by the MAC addresses of each historically targeted host.

Third, the team authored `p0f_handler.sh` to 1) use `tcpdump` to collect packets originating from a specified MAC address and 2) assess captured packets for fingerprint signatures with `p0f` [53]. To capture packets originating from a specified MAC address, the team used

```
$ sudo tcpdump -c [number of packets to be captured] -i
wlan0 -w [.pcap to write to] ether src host [target MAC
address]
```

.

To assess captured packets for fingerprint signatures, the team used

```
$ sudo ./p0f -r [.pcap written to] -o [.log to write to]
> /dev/null
```

.

To review `p0f` [53] logs for operating system guesses, the team authored a Python script to run regular expression matching over the log. In some cases, `p0f` [53] offered varying operating system guesses, such as several guesses for Linux and several guesses for Android, based on traffic originating from the supplied MAC address. The Python script considered the amount of times a guess for a particular operating system appeared in the `p0f` [53] log while formulating the final `p0f` [53] JSON.

Fourth, the team authored `fingerprint.sh` to incorporate `Fingerbank` [18], `Nmap` [30], and `p0f` [53] capabilities into a single script. While

`Fingerbank` [18] capabilities were to always be sniffing in the background, `Nmap` [30] and `p0f` [53] capabilities were to be run in iteration every n minutes. Further, JSON synthesization needed to be conducted every n minutes as well, following the completion of `Nmap` [30] and `p0f` [53] scans. `fingerprint.sh` ensured that all three capabilities could be run in harmony and that JSON records of device fingerprints would be updated as frequently as possible. To enable automated start and stop of the script, the team created a sudo-level service called `mqp_finger.service`. To start all fingerprinting capabilities, the team called
`sudo systemctl start mqp_finger.service`. Similarly, to stop all fingerprinting capabilities, the team called
`sudo systemctl stop mqp_finger.service`. This interface allowed seamless interaction between the fingerprinting module and all other modules in the project. Additionally, a Python script was authored to retrieve the most current fingerprinting data on file for any MAC address.

3.7 Anomaly Detection Configuration

In order to protect and secure the smart home we must have an intrusion detection system in place. This system constantly monitors network traffic to detect and catch any abnormal behaviors. The specific framework our team selected was the Kitsune IDS for network based anomaly detection leveraging machine learning techniques.

At the heart of our anomaly detection IDS, constantly inspecting network traffic, is Kitsune. Kitsune's use of an ensemble of autoencoders enables unsupervised learning to be easily deployed on any network. Unsupervised learning eliminates the need for labeling large training datasets and enables real time, online learning which is beneficial in the current rapidly evolving threat landscape. Due to the utilization

of autoencoders and the restriction on neural network depth, this framework is relatively light weight for a machine learning based IDS. Using as little resources as possible for a robust solution is imperative in the IoT space where there the majority of devices have limited computational power.

There were several small tweaks made to the Kitsune codebase to better suit our application and testbed infrastructure. For the scope of this project, we are not concerned with low latency anomaly detection times. Therefore, we implemented a pseudo-live IDS with Kitsune. This implementation uses tcpdump, with the `-W` and `-G` flags to specify time and rotation, to collect the network traffic flowing through the Raspberry Pi gateway over a user specified time interval and stores these traces in pcap files. These pcaps are then read by Kitsune for processing and inspection. Initially Kitsune was built to take in a single pcap file to learn the features, train and execute the anomaly detector. After some reverse engineering, it can now dynamically parse a directory of pcaps, inspecting the pcaps in chronological order.

By adding this functionality we only have to train the feature extractor and anomaly detector once, and then Kitsune can be run to detect anomalies over the user specified time interval. All of this functionality was then condensed into a single python script that can handle training Kitsune, capturing packets, and executing Kitsune to detect anomalies. A a repository including this script and our modified version of Kitsune is publicly available in the appendix.

3.8 Anomaly Detection Integration

After Kitsune was properly configured the next step in becoming operational is determining an anomaly threshold. Picking an appropriate threshold for an IDS is of paramount importance. The value should result in high confidence that there

is an anomaly while reducing the number of false positives. A common approach to determine a threshold is to use a receiver operating characteristic (ROC) curve, which plots the aptitude of any given threshold. In order to generate a ROC curve we need experimental data of our IDS's performance against a variety of attacks.

The autoencoders that drive Kitsune are trained to reconstruct their inputs. In the case of an network based IDS, the encoders will learn to reconstruct network traffic based on a sample of benign traffic. Once the IDS is finished training, if the traffic is malicious, not benign, then it will have a high reconstruction error. Kitsune measures this error with the RMSE, which will be the metric used to trigger an anomaly.

Tracing high root mean squared error (RMSE) values to their corresponding packets was added to begin tracing back anomalies to potentially infected devices or attack vectors. Once the packet was found for the anomalous error score the MAC address is extracted. Our fingerprinting framework takes this MAC and matches it with the corresponding device. Integrating anomaly detection and fingerprinting is the first step in graceful degradation pipeline because we now have a vague understanding of how the vulnerability was introduced into the network. This is also gives us a strong starting point to begin degrading device specific functionality.

Initially we were hoping to compile experimental results of running our IDS against benign and malicious traffic over the various different attack pcaps provided in the Kitsune paper. While just beginning to experiment with the provided pcaps, we noticed inconsistencies in the RMSE values when executing over the same file. We were able to trace this difference to the preprocess feature built into Kitsune. When reading a file, Kitsune checks to see if `tshark`, the command line version of wireshark [20], is installed on the system. If `tshark` is present, it will parse the pcap with the tool and dump the output to a tsv file. The reason for this being

that `tshark` is much faster than `scapy`, which is used when `tshark` is not present. Reducing overhead is beneficial, however, pcaps parsed by `tshark` were returning different values than those parsed by `scapy`. Given as `scapy` was used to generate the example plots on Kitsune's github, we assumed that `tshark` parsing was a feature added later in development and subsequently went through less testing. Due to the accelerating time frame of this project we could not spend any time debugging and fixing the preprocessing feature.

After the preprocessing bug, we continued examining pcaps and found there were significant differences between the Mirai experiments and other common network attacks such as SSDP floods, DoS attacks, wiretaps, and fuzzing. We believe this discrepancy arises from the footnote in the Mirai pcap description that states it was recorded on a different IoT network. This would explain the drastic difference in RMSE value ranges for malicious traffic. The discrepancy in the experimental data sets left us with two options: generate a ROC curve with all the data excluding Mirai, or generate data on our own network which could include Mirai. Given the sheer scale and devastation Mirai wreaked, particularly in IoT devices, we decided to carry out experimental attacks on the IoT infrastructure we had setup.

The first two attacks deployed were network disruption/denial of service (DoS) attacks. The attacks were launched from a machine running Kali Linux 2019.3, and the primary tools used were airmon-ng, airodump-ng [2], and mdk3 [27]. The attacker also has access to a network card that has both a packet monitoring and injection mode. In this experiment the Alfa AWUS036NEH network card with Ralink RT3070 chipset was used. Both attacks started with a brief reconnaissance phase by utilizing monitor mode to discover and identify devices on the network. Once the target(s) were identified, airodump-ng was launched to watch the target traffic. The first attack launched was a targeted DoS attack directed at a Wi-Fi

camera. Using airmon-ng, the attacker sent a flood of deauthentication packets between the IoT access point and the camera, effectively cutting off communication to and from the camera. The success of this attack was verified through airodump-ng logs and the inability of the home user to launch a live stream from the targeted camera. The second attack was not targeted and aimed to disrupt the entire network. This time airodump-ng was used to monitor the IoT access point and mdk3 sent a flood of deauthentication packets pointed at the access point. This attack is much louder and more noticeable than the targeted attack since it disables communication between all devices connected to the victim access point. The success of the attack was verified by trying and failing to reach multiple different devices connected to the victim access point.

The third attack we carried out was a meddler-in-the-middle (MITM) attack. These types of attacks can be conducting with many different techniques, but we choose an address resolution protocol (ARP) spoofing attack. In this attack we assume that the attacker has gained access to the network. Once in the network, the attacker begins to spoof ARP packets in attempts to associate their MAC address with the target IP address, thus intercepting all traffic meant for the target. Our meddler successfully spoofed ARP messages and intercepted traffic between the IoT access point and an Amazon Alexa. This attack was detected by Kitsune as two major anomalies. The first is when the attacker joined the network and the second is when the attacker began routing traffic through their machine.

The fourth and final attack conducted was infecting a device with the Mirai botnet malware. Mirai targets poorly secured IoT devices to infect as bots. At its height this botnet contained hundreds of thousands of infected devices that were able to launch a record breaking 620 Gigabit per second denial of service (DoS) attack in 2016 [28]. The source code for the malware has since been made public,

and we retrieved a copy from Github (https://github.com/jgamblin/Mirai-Source-Code). To ensure that we could deploy the malware safely in our testbed we disabled several features of the original malware. First we disabled the ability of Mirai to kill other communication services such as telnet and ssh on the bots so that we could reconnect and disinfect the devices. We also removed the ability for the botnet to propogate itself by deleting the loader. However, we kept the scanner for the purposes of watching the bot attempt to talk to and infect other devices but not having the means to without the loader. With our custom build of Mirai, we infected our IoT testbed. A remote command and control (CNC) server was also deployed to monitor the bots and send attack instructions. Once infected, we instructed the infected bots to carry out a DoS attack on a target IP. From examining the traffic and the output from Kitsune, we saw clear spikes when communication with the CNC server was initiated and when the attack commands were received and carried out.

The graphs in Figure 3.5 are a compilation of the RMSE values of all four attack vectors. Notice the scales on each of the graphs and greatly they differ from each other. Obviously just comparing these value to determine a threshold would not be effective. We need a data processing technique to compare Kitsune's output across its wide dynamic range.

A number of different techniques were considered to address this problem including moving averages and whitening transforms. We discovered that by taking the slope of the standard deviation we were able to compare these datasets with the same threshold. Standard deviation is a statistical tool used to measure the amount of variation in a set of values. Therefore, it acts as bridge to compare the variance between sets that have varying ranges. To look for sudden changes or spikes in this variance, which would indicate an anomaly, we take the slope of a set of standard

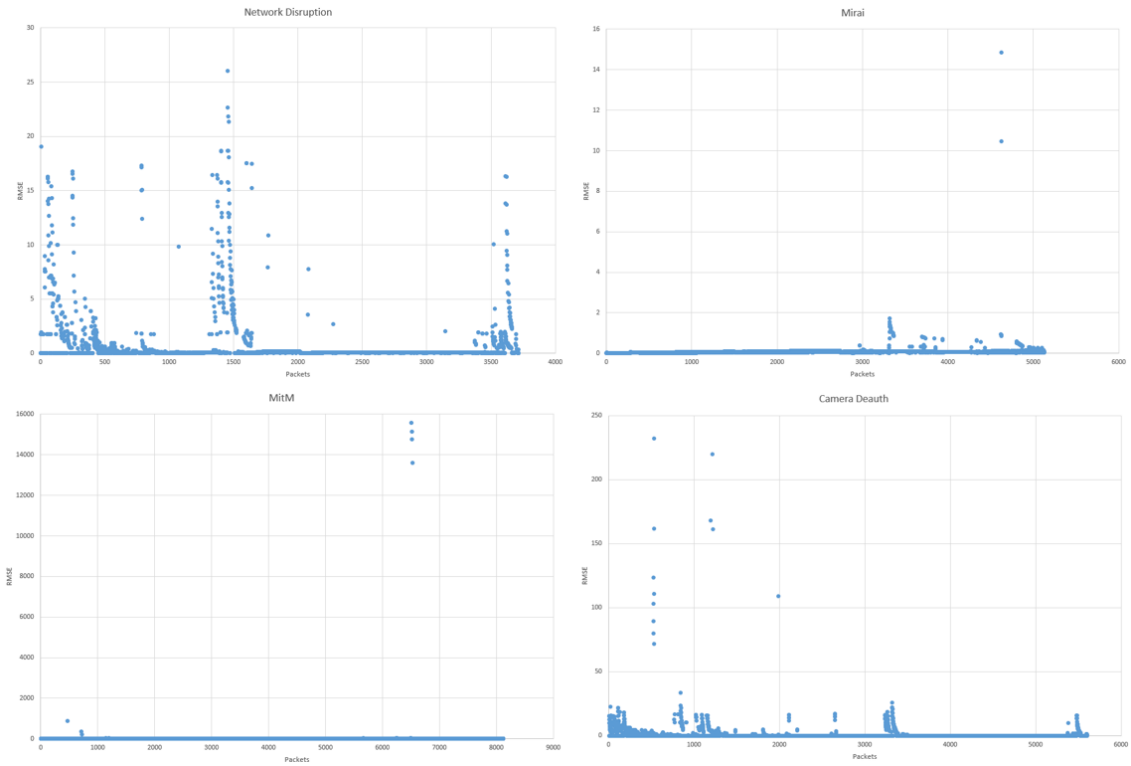

Figure 3.5: Kitsune's RMSE output for all four attack vectors.

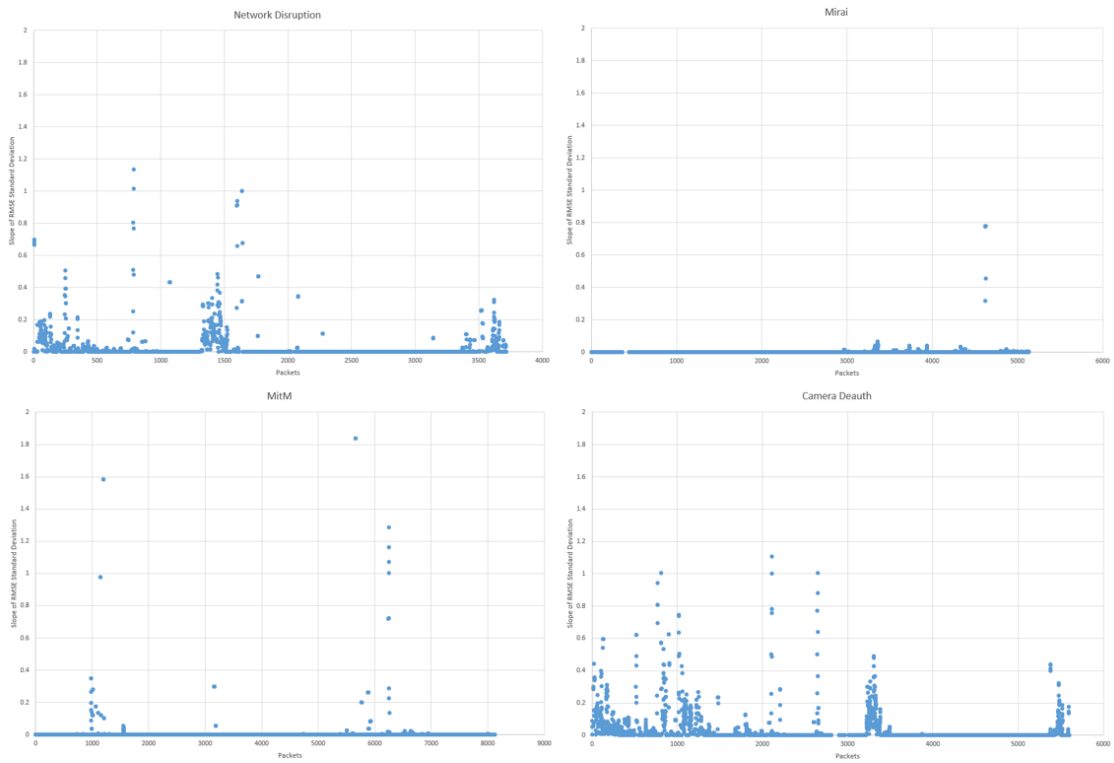

Figure 3.6: The affect of our data processing technique.

deviations. Experimentation yielded the best results when taking the sample standard deviation of sets of 20 values and then calculating the slope of every 5 standard deviation measurement. The results of our data processing technique are shown in Figure 3.6. Note how all graphs now have the same scale.

During this experimentation we discovered another bug that we had introduced with an earlier revision to the codebase. When making Kitsune pseudo real-time, we split the large prerecorded pcap files into smaller files that we later stitched back together to simulate a real operating environment. We noticed that when stitching pcaps back together we were losing dynamic range for the RMSE values. We eventually discovered that splitting the first 50,000 packets that Kitsune uses to train the feature extractor and learn benign traffic patterns causes this loss in range. During the training phase Kitsune must receive a single pcap for optimal performance. A change was made in the driver script and now the first capture waits until the capture reaches 50,000 packets before segmenting pcaps in smaller time frames.

The attack data was compiled into the excel file "ROC.xlsx" which can be found in our Git repository. The generated ROC curve will be discussed in anomaly detection results, section 4.2.

3.9 Determining Device Functionality Bins for use in Risk Identification

The team categorized device functionalities across testbed IoT devices into "bins" so as to provide simple, generalized risks. Across all possible IoT device functionalities, the team designated the following categorical bins in decreasing order of riskiness: a) video streaming, such as live video streaming from a smart camera, b) audio

streaming, such as live audio streaming from an Amazon Echo, and c) device state change, such as switching a smart plug or lightbulb on or off [3]. Then, the team considered scenarios in which devices that offered functionalities in these bins were compromised; here, the team considered risks associated with each type of device functionality bin.

First, an allowed video streaming functionality on a compromised device would allow an adversary to access the onboard camera without restriction. The risks associated with allowing a video streaming functionality on a compromised device are apparent: an adversary could covertly monitor live-stream camera feeds at any time [37]. Footage from indoor smart cameras, perhaps positioned in bedrooms or in living spaces, could be streamed, or even broadcasted publically, at will. Degrading a video streaming functionality on a compromised device would prevent this from occurring.

Second, an allowed audio streaming functionality on a compromised device would allow an adversary to access the onboard microphone and likely onboard speakers without restriction. The risks associated with allowing an audio streaming functionality on a compromised device are therein straightforward: an adversary could covertly listen-in at any time or likely make announcements through speakers [37]. This risk is particularly notable when discussing Amazon Echo Dots; having the ability to listen and to "speak", an adversary could literally carry a two-way conversation with a homeowner. Degrading an audio streaming functionality on a compromised device would prevent this from occurring.

Third, an allowed state change functionality on a compromised device would allow an adversary to not only change device state, but to also make powerful inferences from an end user's state change patterns. Prior research indicates that state changes in devices as simple as smart plugs and smart light bulbs could reveal

user sleep patterns, home and away patterns, and more [3]. With little additional work, adversaries could leverage this information to coordinate burglaries and home invasions. Degrading a state change functionality on a compromised device would prevent this from occurring.

3.10 Graceful Degradation of IoT Device Functionalities

The team defined "graceful degradation" as the ability to autonomously and selectively allow or disallow particular IoT device functionalities to a configuration specified by an end user. The team developed an extensible platform that encouraged an end user to introduce many types of functionalities across many types of devices to a maintained degradation knowledge base. The team's application allowed an end user to specify which functionalities should be allowed or disallowed if certain devices had been compromised. First, the team investigated network traffic associated with device functionalities to ascertain a packet feature vector for use in machine learning integration with the `sklearn` Python library. Second, the team developed a functionality labeling workflow, supervised by an end user, that allowed the introduction of new functionalities for devices on the network. Third, the team developed a functionality modeling workflow, completely unsupervised by way of machine learning integration, that yielded high-confidence classifications for traffic associated or disassociated with device functionalities as introduced initially by an end user. Lastly, the team developed a packet filtering workflow that integrated device functionality machine learning classifiers with analysis of live traffic so as to determine if packets were associated or disassociated with particular device functionalities in real-time.

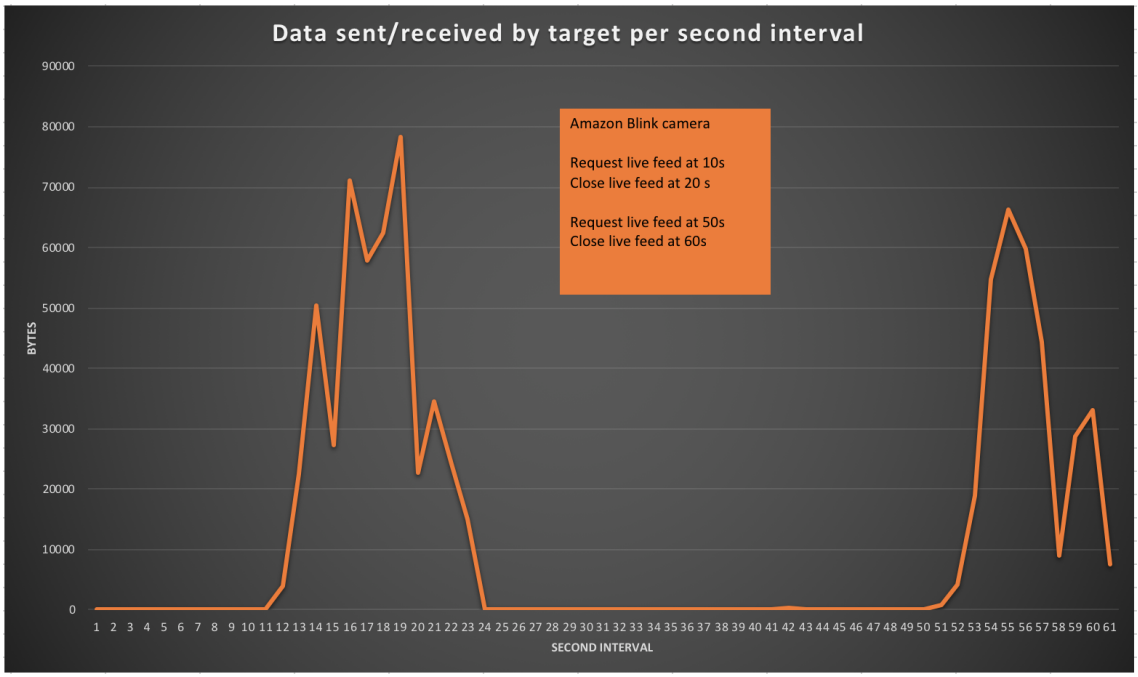

Figure 3.7: Blink camera functionality as derived from throughput.

3.11 Investigating Network Traffic Associated with Device Functionalities

First, the team investigated network traffic associated with device functionalities to ascertain a packet feature vector for use in machine learning integration. Initially, in modeling the work presented in Peek-a-boo [3], the team examined traffic volume, or total data throughput per second, of traffic bound for and/or originating from particular devices in pursuit of recognizing network traffic associated with particular functionalities. Line plots of data throughput against time illustrated that packet size is a useful indicator in mapping traffic to particular device functionalities.

There were many reasons as to why the team could not rely solely on throughput data in classifying traffic, though. Most importantly, with research into NetfilterQueue [1], the team learned that packets cannot gracefully be "delayed" at a router until throughput data over a certain interval could be observed. In other

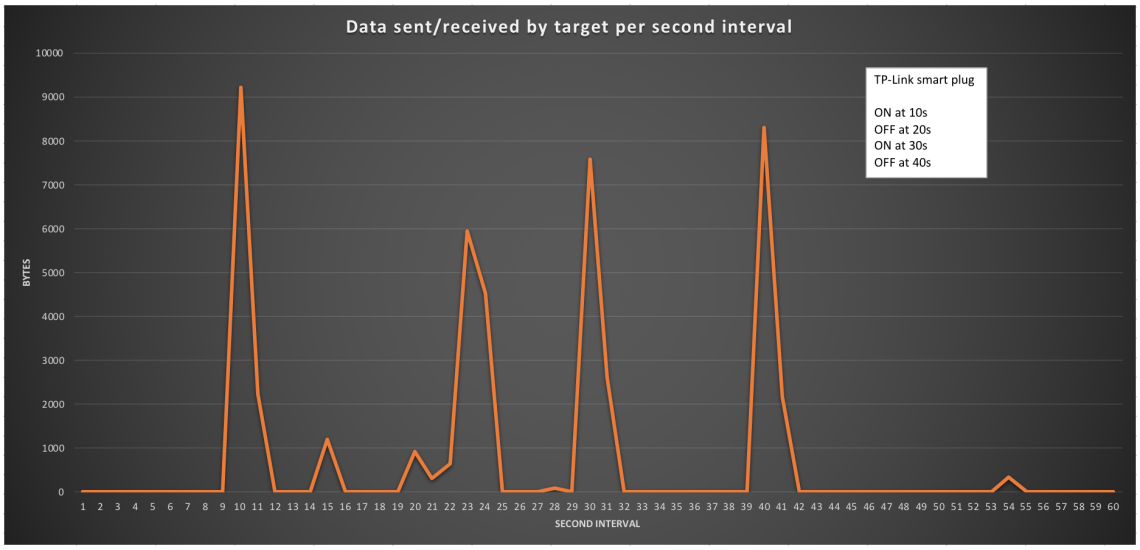

Figure 3.8: TP-Link plug functionality as derived from throughput.

words, in order to determine throughput per second, the team would need to "delay" packets at a router for a period of time as to assess the throughput of the preceding period of time. Herein, using solely packet size in live-time classification would prove difficult in development and cumbersome in implementation. In further modeling the work of Peek-a-boo [3] and Kitsune [34], the team expanded the packet feature vector to include a) packet size, b) TCP source port, c) TCP destination port, d) an integer hash of TCP flags, e) UDP source port, and f) UDP destination port.

In investigating network traffic, the team considered the possible existence of certain device functionalities that supported the concurrent maintenance of multiple TCP streams. TCP differs from UDP in the sense that it is not loss tolerant. The team identified a theoretical edge case where multiple, concurrent streams belonging to device functionalities existed, but where only a subset of those streams was to be degraded. In this case, TCP would attempt to reorganize and resend packets that had been lost to degradation, making it difficult to differentiate one concurrent TCP stream with another. However, with extensive testing, the team concluded that every IoT device to be tested did not support concurrent TCP streaming. The team

observed that even the most complicated IoT devices, such as the Amazon Echo Dot, could only maintain a single TCP stream at once. Further, the team noted that every IoT device to be tested had a "timeout" feature that ceased attempts to access functionalities when they were actively being degraded. UDP streams, being loss tolerant, did not concern the team in this context.

3.12 Labeling Device Functionalities with User Supervision

Second, the team developed a functionality labeling workflow, supervised by an end user, that allowed the introduction of new functionalities for devices on the network. To the team's research, the concept and implementation of an end-user-supervised functionality introduction phase is a novel contribution to the field. The functionality labeling workflow invites an end user to supply a) a device functionality label, such as camera live stream, Echo Dot listen-in, and b) the MAC address of the device to which this functionality will be added in the degradation knowledge base. In turn, this phase maintains a base of various functionalities as associated with various devices as introduced by an end user. Similarly, the team also developed an unsupervised workflow to collect "baseline" traffic from devices. An example of baseline traffic includes device communications when an end user was not interacting with a device so as to make use of a functionality. In booting, the workflow uses Scapy [8] to filter packets sourced from the target MAC address. While interacting with this workflow and adding a new device functionality, an end user is asked to mark start and end timeline bounds for traffic associated with that functionality. For instance, in adding a live-stream functionality for a smart camera, an end user would, in real-time, a) mark a start boundary just before starting to live-stream from that

camera, and b) mark an end boundary just after ending the live-stream from that camera. In marking time intervals in which device functionalities were being used, an end user allows the functionality labeling workflow to then autonomously label individual, captured packets as either a) associated with or b) disassociated with a particular device functionality. For each functionality across each device, captured packets are labeled as functionality-associated or functionality-disassociated, added to a set of (packet, label) tuples, and saved with Python for future reference in machine learning phases of the team's application.

3.13 Modeling Device Functionalities without User Supervision

Third, the team developed a functionality modeling workflow, completely unsupervised by way of machine learning integration, that yielded high-confidence classifications for traffic associated or disassociated with device functionalities as introduced initially by an end user. This workflow was performed autonomously immediately following a user-supervised stage as described previously. As discussed, each introduced functionality for each device maintained a saved set of `(packet, label)` tuples. In modeling the work of Peek-a-boo [3], the team used each picked set, and the saved sets of baseline traffic and of other existing device functionalities, to create a random forest classifier for each device functionality. For instance, in creating a classifier for a new device functionality "A", the team considered existing packet captures previously associated with baseline traffic, device functionality "B", device functionality "C", etc. With this approach, the team utilized existing, saved packet sets to provide "noise" from other device functionalities when modeling a new functionality so as to improve sample size. The modeling workflow extracted

a) packet size, b) TCP source port, c) TCP destination port, d) an integer hash of TCP flags, e) UDP source port, and f) UDP destination port from each saved packet, labeling each as functionality-associated or functionality-disassociated using the labels provided by the user-supervised workflow. In turn, for each device functionality, a matrix of as many rows as there were saved packets in the training set was created with a) packet size, b) TCP source port, c) TCP destination port, d) an integer hash of TCP flags, e) UDP source port, f) UDP destination port, and g) functionality associated by indicating whether a particular packet was associated with a device functionality as column headers. Across all device functionalities, the team achieved an average model confidence of 94% using the random forest algorithm. Tuples of model objects and model accuracies were created and saved for each device functionality for reference in live packet filtering.

3.14 Integrating Functionality Models into Live Packet Filtering

Lastly, the team developed a packet filtering workflow that integrated machine learning models to identify device functionality so as to determine if packets were associated or disassociated with particular device functionalities in real-time. The degradation system maintained a directory of saved random forest models as associated with particular device functionalities. For each device with modeled functionality, the packet filtering workflow paired a NetfilterQueue queue [1] and iptables rule,

```
$ sudo iptables -t raw -I PREROUTING -m mac --mac-source
{device MAC address} -j NFQUEUE --queue-num {queue number}
```

so as to direct traffic sourced from such devices to a uniform Python workflow.

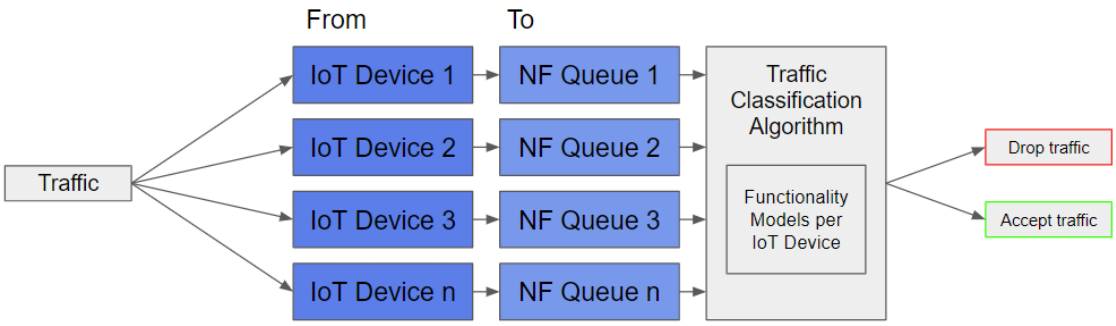

Figure 3.9: From one stream of traffic, a Netfilterqueue is created for each device on the network, and all feed into one classifier.

In utilizing the PREROUTING table, the team could drop packets even before they had been routed, significantly increasing overall performance [31]. Initially, the team attempted to use a singular NetfilterQueue queue to handle all traffic on the network. With further investigation, the team made the realization that, for best performance, packets need to be "in and out" of a NetfilterQueue queue in under 500 milliseconds. To achieve this peak performance, the team decided to create one queue per device to be secured, but maintain a single target Python workflow so as to maintain scalability.

For each intercepted and queued packet, the Python workflow extracted a) packet size, b) TCP source port, c) TCP destination port, d) an integer hash of TCP flags, e) UDP source port, and f) UDP destination port. The workflow would extract source MAC address, as well, so as to determine the appropriate random forest models to reference. All random forest models across all device functionalities were loaded into memory at the start of live-filtering to improve live-time responsiveness. After identifying the appropriate models for a device, the Python workflow used the packet feature vector as extracted from each packet to make a prediction against each functionality model so as to associate or disassociate a queued packet with a functionality. In testing, the team determined that a single prediction could be

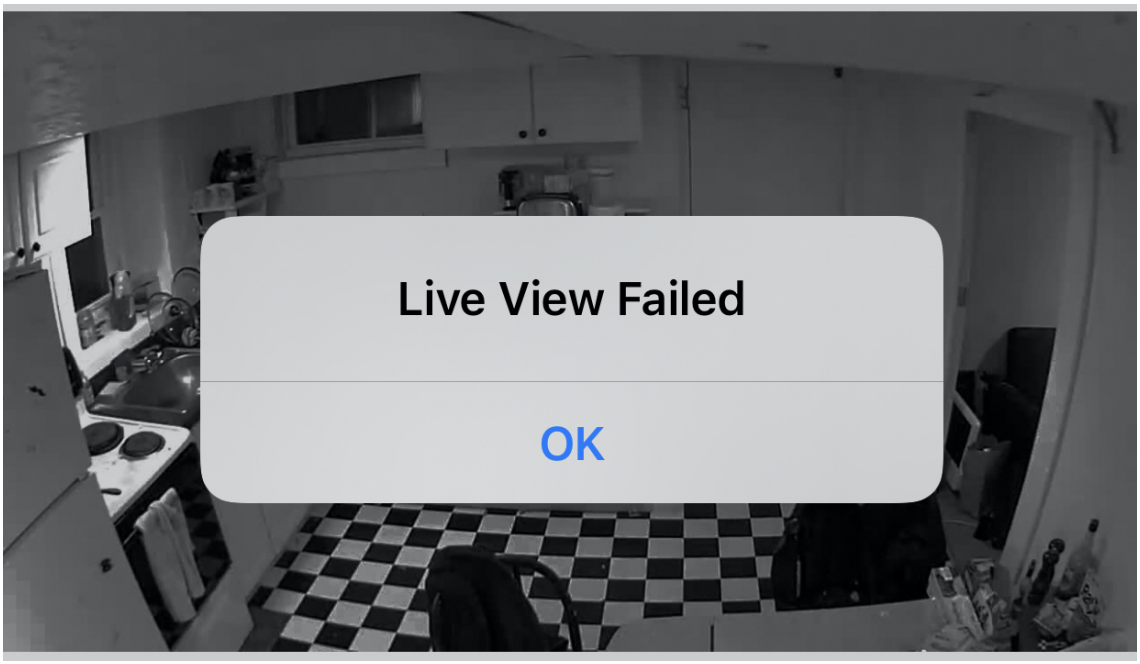

Figure 3.10: Blink camera live stream fails when degraded in real time.

made in approximately 40 milliseconds, meaning that around twelve functionalities could be added for each device while maintaining peak performance. If a packet was predicted to be associated with a particular functionality, and if the particular functionality was to be degraded as specified by an end-user, that packet would be dropped. Similarly, if a packet could not be positively classified as a particular device functionality by any of random forest models on record, that packet was assumed to be anomalous and was dropped.

3.15 Constructing and Integrating the Graphic User Interface

This integration platform would take the form of a user-friendly Graphic User Interface for a nontechnical home user to interact with.

Since we were hosting the GUI on the Raspberry Pi, which was situated behind a

router, we needed to set up port forwarding such that all HTTP traffic to a specified port of the router, would connect into the Pi. We mapped Port 8081 on the Pi to port 2023 on the router. On the Pi, the GUI was hosted at `192.168.0.149`, the Pi's private IP address, on port 8081. Once running on the Pi, the GUI can be opened from any web browser by entering the routers IP 68.116.160.168, a colon, followed by the forwarding port, 2023, for example http://68.116.160.168:2203/.

In selecting a development platform for a front end, we had the following requirements:

1. We need the ability to run headless, since the suite will be ran from the Pi without a connected display

2. We need abundant documentation to support the rapid development needed for the scope of this project

3. We need a lightweight solution with few dependencies, as the resources available on the Pi are limited

4. We need the ability to run commands and to dynamically load and change data to display

We considered Django, Pyramid, PyQT, and `PySimpleGUI` as possible solutions. We selected `PySimpleGUIWeb` [40] as it met all of our criteria. First, web hosting functionality available and streamlined through Remi python package. Second, full "cookbook" of documentation for various programs as well as tutorials. Third, Remi is the only other package installed with `PySimpleGUI`. Fourth, while elements cannot be created or deleted during runtime, they can be hidden or shown, and can use subprocess through python to execute commands as normal to a shell. `PySimpleGUI` is a python wrapper for tkinter, Qt, wxPython and Remi as a means

for rapid development through less lines of code. It is deployable on almost all OS's as well as deployable on hardware form Android devices, Raspberry Pi Monitors, computers, and web browsers.

3.16 Viewing the Results of Fingerprinting

The management system displays the fingerprint data to the GUI by analyzing the output folder from our Fingerprint module. If a new folder is discovered it is added to the data structure and the device's MAC address is displayed under found devices. Then, the detailed information and fingerprint result json is parsed.The several nested dictionaries within each synthesized json had to be flattened. Extraneous information such as boolean values, numbers, or empty lists, were also removed. The final display output is the most relevant non-empty fields that were obtained from fingerprinting, and useful to a home user in matching the description to a device they own.

3.17 Loading Saved Trained Functions

When the "Load Functions" button is pressed, any saved device function models will be identified and loaded into the GUI as a list. This will search through each device folder that has been generated from fingerprinting. If a sub-directory is discovered that is not "baseline" training data, then the name of that function, the device MAC address and default rule of "allow" is added as a single line in a text file named `degradeRules.txt`.

3.18 Anomaly Detection Integration and Notifications

Upon start-up of the GDI Suite, a threaded call to the `runIDS()` function from `gui_kitsune.py` begins training for anomaly detection. Arguments can be passed for anomaly trigger threshold, the number of packets to collect for training, the .pcap time interval, and max auto encoder size. Currently we pass `(10, 30000, 10, 15)`. Once trained, if an anomaly occurs, the source and destination MAC addresses will be appended to a log file. The GUI must actively look for modifications to this file without blocking other processes. Through the python library `watchdog` [6], we create a pattern matching observer that monitors the log file for modifications. When modification is detected we pull the last added line of the log file for the source and destination MAC addresses of the anomaly. We notify the user, activate the degradation filter and display the friendly names of the anomalous devices if available.

3.19 Viewing and Changing a Function's Degradation Status

Upon selecting a function from the displayed list, a look-up function is called to check the current rule by MAC address and function name pair. If educational flags are found, the relevant educational information is displayed. Depending on the current rule in the text file for degrade or allow, the opposing choices button will be enabled and highlighted. For example, if a rule is currently allowed, Only the option to block the function will be highlighted. Upon clicking block in this example, the look-up function is called again, to update the value within `degradeRules.txt` from

`allow` to `block`. The allow button will now be highlighted to provide the option to accept this function once again.

3.20 Training a New Function through the User Interface

Once a user populates the device and function information. When ready the user can click the "Train Function", button. This will pass MAC address, and function name to a blocking subprocess in the form of

```
$ sudo python3 /home/pi/MQP/degradation/create_training_set,py --mac
MAC --feature functionName
```

.

When successfully trained, the MAC, function name, allow, and educational flags stored as numbers are added to "degradeRules.txt". Then another blocking process calls `update_models.py`, which will update the random forest classifier models of all functions.

3.21 Activating the Degradation Filter

The degredation filter is triggered by anomalies and can also be manually triggered through clicking the "Activate Degrade Filter" button. This allows a user to have the option to always enact their preferred level of privacy. When the degradation filtering rules are activated, we parse the MAC,function,rule lines from `degradeRules.txt` into pairs of a device MAC, followed by all of its functions and corresponding rules. Example of such pairs can be seen as

"MAC1,function1,rule1,function2,rule2 MAC2,function1,rule1". These pairs are passed as arguments to a subprocess of `secure_filter.py`. From there `secure_filter.py` builds the IP table rules to degrade the functions accordingly.

3.22 Disabling the Degradation Filter

When the user either wishes to update their device functionality preferences, or deactivate the filter, they press the red button which states it will remove the filter. This calls

```
$ pkill -TERM -P pid
```

where pid is the process ID (PID) of the sudo command which activated the filter. This will kill the process and all child processes of the PID which spawn from issuing with sudo privileges. Next a subprocess is issued to ensure the IP Table rules are flushed via

```
$ sudo iptables -t raw -F
```

. All degradation is stopped, and can be started again if the filter is triggered by anomaly or button selection.

Chapter 4

Results, Discussion, and Limitations

Here, we discuss the fortes and limitations of the approach aforementioned. We speak to fingerprinting, intrusion detection, graceful degradation, and integration capabilities as developed.

4.1 Fingerprinting

To validate the fingerprinting approach aforementioned, the team conducted research to determine actual manufacturers, operating systems, and device types of the testbed IoT devices. Then, the team referenced fingerprinting data obtained from `Nmap` [30], `Fingerbank` [18], and `p0f` [53] and cross-checked these results with open source information known to be correct [29] [48] [9] [16] [41]
[24]. A summary of fingerprinting accuracies can be observed in the table below. Overall, the team's fingerprinting approach correctly determined device manufacturer in 89% of cases, operating system in 67% of cases, and device type in 67%

Device	Manufacturer guess	Operating system guess	Device type guess
Amazon Echo Dot	Google	Android	Media device
TP-Link smart plug	TP-Link	IBM OS/2	Media device
Raspberry Pi 3	Raspberry Pi	Linux 3.x | 4.x	Raspberry Pi
Blink smart camera 1	Texas Instruments	Lantronix OS	IoT sensor
Blink smart camera 2	Texas Instruments	Lantronix OS	IoT sensor
Etekcity smart plug	Espressif	Espressif firmware	IoT plug
Particle Photon	Univ. Global Scientific	Linux	NEST thermostat
Google Chromecast	Google	Linux 2.6 - 3.10	Google Home device
Philips HUE bridge	Philips	No guess	Lighting bridge

Figure 4.1: Correctness of manufacturer, operating system, and device type guesses made by fingerprinting capabilities (correct guesses highlighted in green).

of cases. However, the probability that at least two of the three categories were determined correctly for a device was 89%. Below, correct guesses are highlighted in green. Full fingerprinting results can be found in the Appendix.

The team's layered approach to device fingerprinting yields generally reliable data. The results above suggest that device manufacturers can be determined accurately, but that device operating systems and broadened device types are more difficult to ascertain. In some cases, fingerprinting capabilities would yield the manufacturer of a device's network card. Though accurate, these guesses could confuse an end user attempting to identify devices on his or her network. The team acknowledges that previous works such as AuDI [32], ProfilIoT [38], and IFRAT [25] provide comprehensive suites that more accurately fingerprint devices. However, for the purposes of this project, which focuses primarily on intrusion detection and graceful degradation, the team found its fingerprinting capabilities to be satisfactory. In future work, the team would integrate a more comprehensive fingerprinting

solution so as to provide increased ease of use for an end user.

Although our fingerprinting approach accurately determined two of three fingerprinting indicators (manufacturer, operating system, broader type) for 89% percent of tested devices, we consider possible limitations of the capability. First, `Nmap` [30] is "active", meaning that it interacts directly with a targeted device via network communication. Fingerprinting traffic between a router and a targeted device can, in some cases, be flagged as anomalous by an intrusion detection system. Although we did not encounter this scenario in testing, we recognize that this problem could arise in future work. Second, the team uses well-known and well-documented fingerprinting tools that could harbor known vulnerabilities. A well-read attacker could tailor tool-specific traffic spoofers so as to misdirect `Nmap` [30], `Fingerbank` [18], or `p0f` [53] with the ultimate goal of undermining fingerprinting confidence.

4.2 Anomaly Detection

After exploring a multitude of attack vectors and collecting and labeling network captures, we synthesized the data into a ROC curve. This visualization allowed us to determine an appropriate threshold based on the trade off of detecting an attack versus triggering a false positive. By utilizing this tool, we are able to select a threshold that limits false positives and maximizes meaningful alerts. Minimizing false positives is necessary to build trust between the system and the user, as the user does not want to be bombarded by false alerts. If the false positive rate is too low, however, then the IDS begins to loose detection sensitivity. For example, by selecting the threshold that has less than a 10% false positive rate we achieve an attack sensitivity rate of nearly 50%.

While Kitsune excelled at providing an efficient and effective IDS, it is not infal-

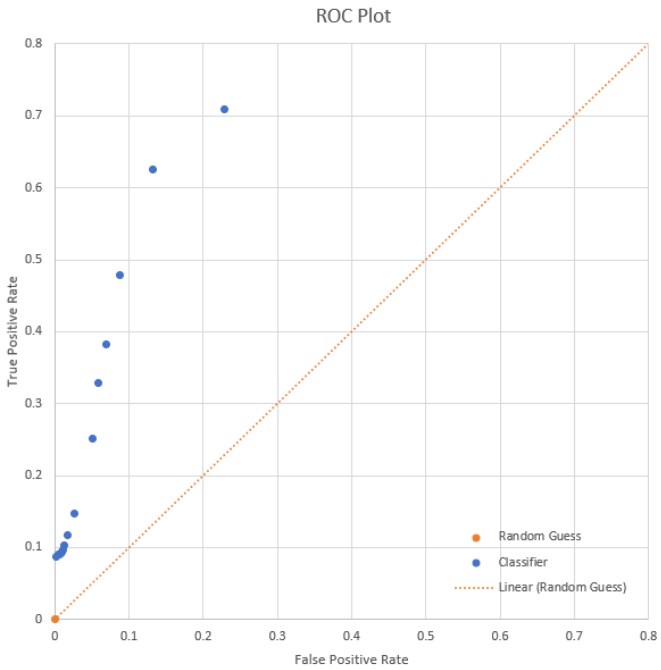

Figure 4.2: ROC Plot for Kitsune IDS.

lible. Unsupervised learning, for example, is susceptible to poisoning attacks. This can occur when malicious behavior is present in network traffic during the training or learning phases of the feature extractor and anomaly detector. The consequences of a successful poisoning attack would mean that the IDS would not detect malicious traffic as it was trained to consider that traffic as normal.

Additionally, when carrying out a MITM on our testbed network, the introduction of a foreign device to the network triggered an anomaly. The behavior is beneficial in the case that the foreign device is actually an attacker with malicious intent. In this situation, the user would like to be notified when such an event occurs. However, if the foreign device is benevolent device, such as a new device the user is adding to thier home, they would not like to trigger an anomaly. Kitsune would have to be retrained to include benign traffic from this additional device.

Average classifier confidences per device		
(Across all trained device functions)		
Device	**Confidence**	**Trained functions**
Amazon Blink camera	0.91	4
Amazon Echo Dot	1.00	3
TP-Link smart plug	0.93	2
Etekcity smart plug	0.95	2
Raspberry Pi 3	0.93	1
Average	**0.9440**	**2.4**

Figure 4.3: Average classifier confidences for trained functionalities across devices.

4.3 Degradation

To validate the degradation approach aforementioned, the team determined an average classifier confidence and attempted to positively identify device functionalities in real time. First, the team's approach formulated a random forest classifier for each device functionality. Each random forest classifier provided a confidence level, that is, a self-determined, but theoretical, measure of accuracy. Across trained device functionalities, an average classification confidence level of 0.9440 was achieved.

Second, the team used ten live tests to record a confusion matrix for each device functionality as trained. For the purposes of experimentation, the team started degradation capabilities in a passive context so that device functionalities would be recognized but not degraded. In five of the ten tests, the team demonstrated a target functionality and then recorded the functionality recognized by the degradation capabilities. For example, if the team was testing for recognition accuracy of smart camera live streaming, the team would initiate live streaming from a mobile app and then record the functionality as recognized by the degradation suite. Similarly, in the

Average classifier accuracies per device		
(Across all trained device functions)		
Device	**Accuracy**	**Trained functions**
Amazon Blink camera	0.90	4
Amazon Echo Dot	1.00	3
TP-Link smart plug	0.85	2
Etekcity smart plug	0.95	2
Raspberry Pi 3	0.93	1
Average	**0.9260**	**2.4**

Figure 4.4: Average classifier accuracies for trained functionalities across testbed devices, determined with real-time experimentation.

other five tests, the team would demonstrate other, trained device functionalities, but not the target functionality. For example, if the team was testing for recognition accuracy of smart camera live streaming, the team would initiate another feature, such as motion detection arming, and then record the functionality as recognized by the suite. Results per functionalities were represented as follows in the matrices as presented herein: a) true positive: a functionality was demonstrated and classified, b) false positive: a functionality was not demonstrated but was classified, c) true negative: a functionality was not demonstrated nor classified, and d) false negative: a functionality was demonstrated but not classified. Across all confusion matrices, the team achieved an average accuracy, or ACC, of 0.9260.

The team's degradation results demonstrate that three minute, user-supervised, functionality training phases can be leveraged to accurately identify packets associated with particular IoT device functionalities in real time. By maintaining functionality-labeled packets as retrieved from previous training phases, the team creates accurate classifiers infused with noise from extraneous device functionalities

and requires a very brief training phase for new functionalities. The results demonstrate that noise improves classifier confidence. The team expects this approach would scale across all IoT devices that utilize Wi-Fi or Ethernet communication as random forest classifiers are created individually for each and every device functionality. Further, this approach shows that even functionalities with encrypted traffic can be classified with high confidence.

Many advanced traffic classification schemes, such as Peek-a-boo [3] and Kitsune [34], leverage machine learning approaches to draw meaningful associations and dissociations between feature vectors as derived from packets of various device functionalities. Random forest, as used in this project, was evaluated by the authors of Peek-a-boo [3] and reported to have an accuracy of 91% when mapping network traffic to device functions; expectedly, our team achieved a comparable accuracy of 94% using random forest classifiers. This is understandable because our team's leverage of machine learning to degrade functionalities was very similar to Peek-a-boo's [3] leverage of machine learning to ascertain device activity from an adversarial standpoint. Of note, Peek-a-boo [3] also evaluated the k-nearest neighbors algorithm, which was reported to have an accuracy of 89%. Future work could explore various machine learning algorithms in the context of degradation as supported by machine learning.

Although our approach yielded high confidences and accuracies, we consider possible limitations of the system. First, this particular approach could face lower accuracies while attempting to classify traffic associated with automated firmware updates. Automated updates perform at random time intervals and are inherently difficult to classify without existing knowledge of their behavioral characteristics, which oftentimes vary across iterations. Therefore, automated updates would likely be labeled as anomalous in a contingency scenario where a compromised device was

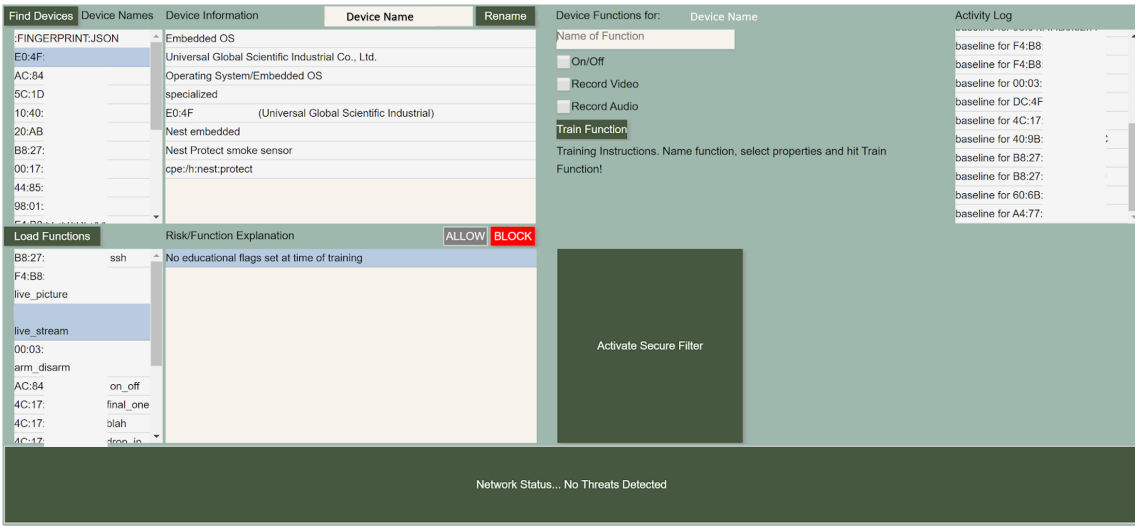

Figure 4.5: A full look at the completed GDI Security Suite.

attempting to update automatically. However, if a user were to train an "update functionality" manually, this problem would likely not arise as traffic associated with the update could be modeled and identified as a functionality. Second, while brief, the degradation stages of this approach still require user supervision. Assuming that a home user will cooperate in training is critical to the success of this system, and while training periods are short, if a user does not train functions, there is no way to detect or degrade them.

4.4 User Interface

We successfully unified all three platforms into one straightforward, appealing user interface we named the GDI Security Suite, see Figure 4.5. Our user interface implements the HCI practices of repetition, alignment, contrast, and spacing to create a familiar and easier to navigate interface for users.

From the GDI Suite, a user can fingerprint devices, view device information, apply friendly names to the MAC addresses of devices, load trained device functions,

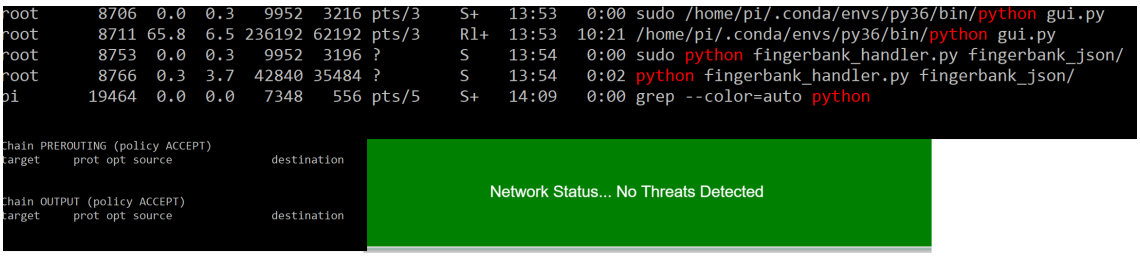

Figure 4.6: On top is all running python processes, followed by the normal UI display shown in green and the empty iptable rules.

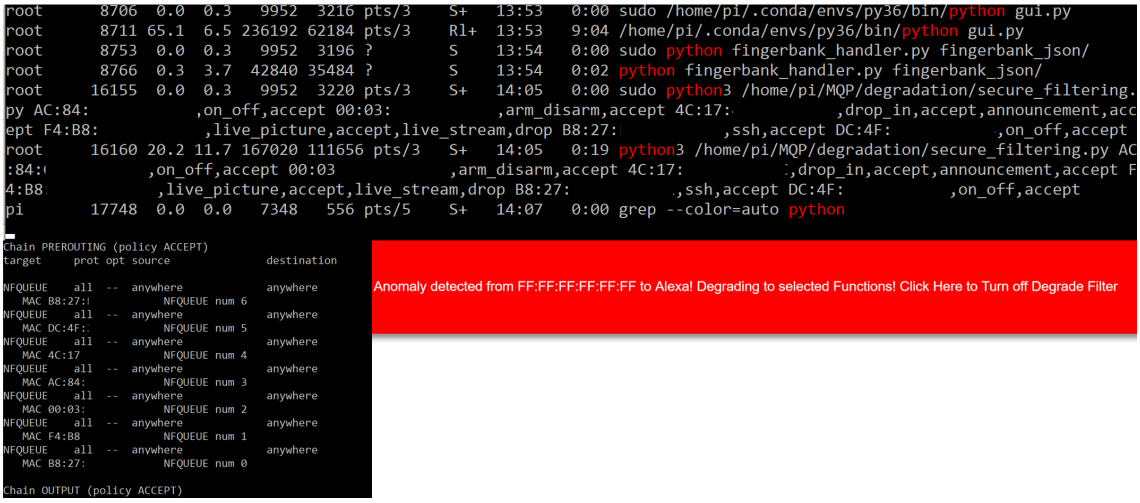

Figure 4.7: With the degrade filter on, the iptable rules are bound and notification displayed.

train new device functions, toggle a device function between allowed or blocked, and activate the security filter.

In Figure 4.6 we see the background information related to the GDI Suite running in an uninfected state. In Figure 4.7, we see the updates to both the UI and the back-end when an anomaly is detected, and the degrade filter is enabled.

The GDI Suite can provide the following optional educational text:

- *This function operates basic device states like on or off. Hackers can turn these devices on and off at their choosing. Also functions that control the powering on or off of devices such as lights, or thermostats, may allow hackers to infer*

when you are home or awake, versus away or asleep.

- *This function records audio. Hackers can view camera feeds, take pictures, and activate or deactivate cameras. Again they can see if you are home or awake, versus away or asleep.*

- *This function records video. Hackers can listen to and record audio files from your device. They may also be able to broadcast their own audio announcements over the speakers whenever they choose.*

Our UI faced latency issues which stem from two main issues. The first is that `PySimpleGUI` is designed for rapid development of simple form-like UIs. It is meant for the simple getting of text and values, not for a complex dynamic web application. The size and complexity of the UI causes occasional glitches where not all of the information is fully displayed and requires another click, which should be eliminated for a final product. The second, and largest attributor to lag in the UI is the hardware restrictions of the Raspberry Pi. We are using a Raspberry Pi 3 with 1 GB of RAM and limited processing power, running Raspian Buster operating system. From this device we host a web server, train machine learning algorithms, host an intrusion detection system, and run several subprocesses. The random forest models are loaded directly into RAM to ensure quick matching, yet consume almost the entirety of the available 1 GB in some cases. The learning of new functions and intrusion detection is carried out by the CPU. The RAM and CPU are central limiting factors to this projects current scalability and performance speed.

Another limitation of the UI is the lack of user testing for the educational value of our system. While our system implements HCI design principles and provides simplicity and ease of use, there is no ground truth for its educational value to a non-technical home user.

Chapter 5

Future Work

In this chapter, we summarize limitations of our current work and highlight areas for future improvement. To summarize, we believe our current limitations include:

- Latency from hardware restrictions
- Fingerprinting inaccuracies and/or lack of specificity
- Degradation requires user interaction
- Susceptibility to adversarial machine learning attacks
- Creating a more sensitive IDS while still minimizing the false positive rate

In order to fully automate the system, a team could develop a traffic classification approach that could identify clusters as indicative of particular functionalities without supervision. The fingerprinting and anomaly degradation stages as discussed previously are completely unsupervised; an unsupervised degradation stage would make the team's approach completely autonomous.

While already automated, anomaly detection could be made more robust by creating a device functionality-based anomaly detection system. Currently, our model

intercepts network-based attacks. However, being able to view types and combinations of device functionalities as well as their frequncies, offers a more holistic anomaly detection system. For example, if a Hue light bulb starts turning off and on rapidly in the middle of the night, an attacker could be causing such issue that should trigger an anomaly. Function-based anomaly detection would require a much more in depth training phase as a users "normal" device usage must be learned over several days or even weeks. During this training phase, the device may be susceptible to start up attacks, since the large period of time allows for attackers poison the benign training data with unnatural device functionalities.

We would also like to continue testing our implementation of Kitsune to both validate our data processing technique and test the IDS against a wider array of attack vectors. The more thoroughly we test the IDS the better we can adapt it to the dynamic threat landscape that it will undoubtedly see. Additional implementation would also include a retrain functionality that will activate after events such as firmware updates and the introduction of a new, benign device.

Further integration between anomaly detection and the UI could include a slider to tune the sensitivity of the IDS. Based on our generated ROC curve, different thresholds can be used to achieve a less sensitive IDS with smaller probabilities of raising false positives and vice versa. Incorporating a slider to allow a user to tune the IDS enables to user to customize the security suite to their desired acceptable risk tolerance.

A future user testing experiment would provide feedback on the interface design, labeling queries, and risk education of our system. Updates could then be added to restructure the information in a more logical order to users, or streamline areas users found confusing.

As mentioned, our current system could be expanded to save the function-traffic

feature mappings to a database, and used in the future by the scientific community. The findings from a 2018 survey of machine learning practices and techniques for ML in network applications, highlights a need for such live device traffic: "Therefore, a combined effort from both academia and industry is needed, to create public repositories of data traces annotated with ground truth from various real networks" [10]. We already classify and store rules for each functionality with a friendly name. With the consent of a home user, a friendly name and training sample pair can be saved to a public database and used to boost performance of classifying the same feature, as well as used by researchers for similar work.

If future devices depend heavily on Bluetooth or alternative protocols, we believe our random forest classification, and training method would allow for alternate protocol functions to be degraded similar to Wi-Fi. This would require the proper hardware to sniff these protocols. New features would have to be identified unique to the transmission packets of those protocols. Those features would then be extracted and computed with the random forest classifier. Alternate protocols do not have IP Tables, so enacting rules to allow or drop packets will be more complex, possibly requiring additional hardware.

Chapter 6

Conclusion

In conclusion, the team engineered a complete, integrated IoT security suite that accurately and confidently fingerprinted WiFi devices, detected network-based intrusions, and gracefully degraded individual device functionalities. First, across 89% of tested devices, the team accurately fingerprinted manufacturer and operating system, manufacturer and type, or operating system and type. Second, our near real-time IDS detects network attacks and intrusions with greater than 90% accuracy by leveraging a lightweight ensemble of autoencoders. Third, the team unified all of these successful tools into one cohesive, user-friendly, graphical user interface. The interface allowed non-technical end users to identify and train device functionalities while learning about the risks associated with different categorical classifications of device functionalities. Lastly, across all device functionalities as trained, the team achieved a) an average traffic classifier confidence of 94% and b) an average traffic classifier accuracy of 93%. The performance of our degradation capabilities were greatly enhanced by training the functionality classifiers with network traffic already recorded from previous training phases.

If widely adopted, the approach presented would offer an elevated level of net-

work security and user privacy for WiFi networks harboring IoT devices. We present a highly scalable, accurate, and confident framework that remains comprehensible and customizable to the average end user. GDI security differs from current solutions in presenting a graceful degradation strategy that allows compromised devices to maintain a minimum level of functionality in lieu of complete isolation or quarantine. The IoT device industry only continues to grow, introducing new security and privacy vulnerabilities. GDI security provides and adaptable, comprehensive home security suite for the IoT era.

Chapter 7

Appendix

In this section, we present confusion matrices and console logs as obtained through degradation testing and additional screenshots of the user interface.

Amazon Blink Camera	
Functionality	**Classifier confidence**
Baseline behavior	0.85
Live video feed	0.98
Live photo snapshot	0.99
Motion detection arm	0.87
Motion detection disarm	0.87
Average	**0.91**

Figure 7.1: Classifier confidences for Blink camera.

Amazon Blink Camera		Actual class	
Baseline behavior, 180s training		Classified	Not classified
Predicted class	Classified	4	1
	Not classified	0	5
ACC:	**0.9**	4	6

Figure 7.2: Confusion matrix for Blink baseline behavior experimental testing.

Amazon Blink Camera		Actual class	
Motion arm/disarm, 180s training		Classified	Not classified
Predicted class	Classified	4	1
	Not classified	1	4
ACC:	**0.8**	5	5

Figure 7.3: Confusion matrix for Blink motion detection experimental testing.

Amazon Blink Camera		Actual class	
Live photo, 180s training		Classified	Not classified
Predicted class	Classified	4	1
	Not classified	0	5
ACC:	**0.9**	4	6

Figure 7.4: Confusion matrix for Blink live photo experimental testing.

Amazon Blink Camera		Actual class	
Live video feed, 180s training		Classified	Not classified
Predicted class	Classified	5	0
	Not classified	0	5
ACC:	**1**	5	5

Figure 7.5: Confusion matrix for Blink live video experimental testing.

Amazon Echo Dot	
Functionality	**Classifier confidence**
Baseline behavior	1.00
Live listen-in	0.99
Announcement	1.00
Average	**1.00**

Figure 7.6: Classifier confidences for Echo Dot.

Amazon Echo Dot		Actual class	
Baseline behavior, 180s training		Classified	Not classified
Predicted class	Classified	4	1
	Not classified	0	5
ACC:	**0.9**	4	6

Figure 7.7: Confusion matrix for Echo Dot baseline behavior experimental testing.

Amazon Echo Dot		Actual class	
Announcement, 180s training		Classified	Not classified
Predicted class	Classified	5	0
	Not classified	0	5
ACC:	**1**	5	5

Figure 7.8: Confusion matrix for Echo Dot audio announcement experimental testing.

Amazon Echo Dot		Actual class	
Drop in, 180s training		Classified	Not classified
Predicted class	Classified	5	0
	Not classified	0	5
ACC:	**1**	5	5

Figure 7.9: Confusion matrix for Echo Dot live, audio drop-in experimental testing.

Etekcity Smart Plug	
Functionality	**Classifier confidence**
Baseline behavior	0.89
On/off	1.00
Average	**0.95**

Figure 7.10: Classifier confidences for Etekcity smart plug.

Etekcity Smart Plug		Actual class	
Baseline behavior, 180s training		Classified	Not classified
Predicted class	Classified	4	1
	Not classified	0	5
ACC:	**0.9**	4	6

Figure 7.11: Confusion matrix for Etekcity smart plug baseline behavior experimental testing.

Etekcity Smart Plug		Actual class	
On/off, 180s training		Classified	Not classified
Predicted class	Classified	5	0
	Not classified	0	5
ACC:	**1**	5	5

Figure 7.12: Confusion matrix for Etekcity smart plug on and off behavior experimental testing.

Raspberry Pi 3	
Functionality	**Classifier confidence**
Create SSH connection	0.91
Average	**0.91**

Figure 7.13: Classifier confidences for Raspberry Pi 3.

Raspberry Pi 3		Actual class	
Create SSH conn., 180s training		Classified	Not classified
Predicted class	Classified	4	1
	Not classified	0	5
ACC:	**0.9**	4	6

Figure 7.14: Confusion matrix for Raspberry Pi 3 SSH connection initiation behavior experimental testing.

TP-Link Smart Plug	
Functionality	**Classifier confidence**
Baseline behavior	0.96
On/off	0.90
Average	**0.93**

Figure 7.15: Classifier confidences for TP-Link smart plug.

TP-Link Smart Plug		Actual class	
Baseline behavior, 180s training		Classified	Not classified
Predicted class	Classified	4	1
	Not classified	0	5
ACC:	**0.9**	4	6

Figure 7.16: Confusion matrix for TP-Link smart plug baseline behavior experimental testing.

TP-Link Smart Plug		Actual class	
On/off, 180s training		Classified	Not classified
Predicted class	Classified	4	1
	Not classified	1	4
ACC:	**0.8**	5	5

Figure 7.17: Confusion matrix for TP-Link smart plug on and off behavior experimental testing.

Figure 7.18: Classifying Alexa drop-in behavior

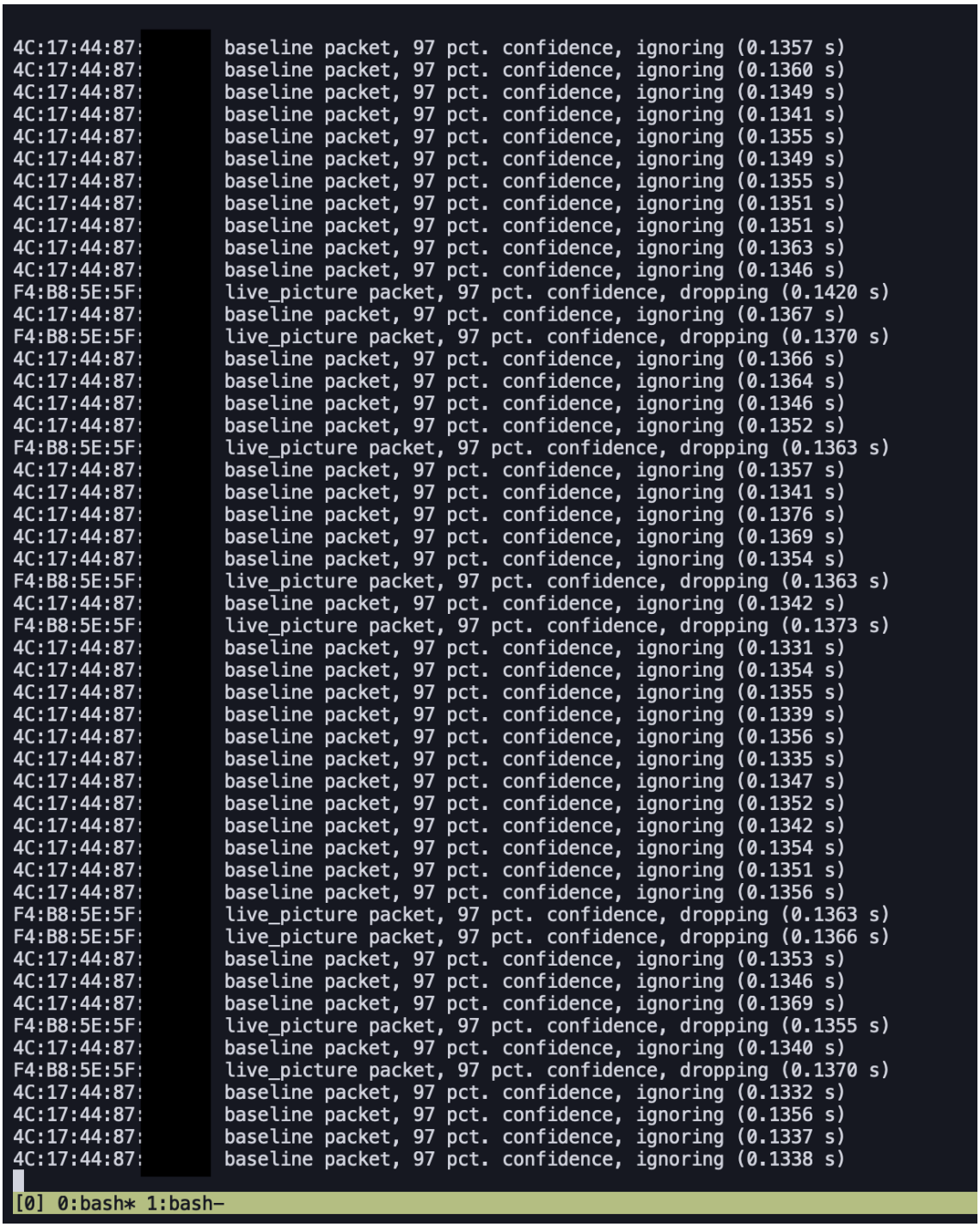

Figure 7.19: Classifying Alexa baseline behavior

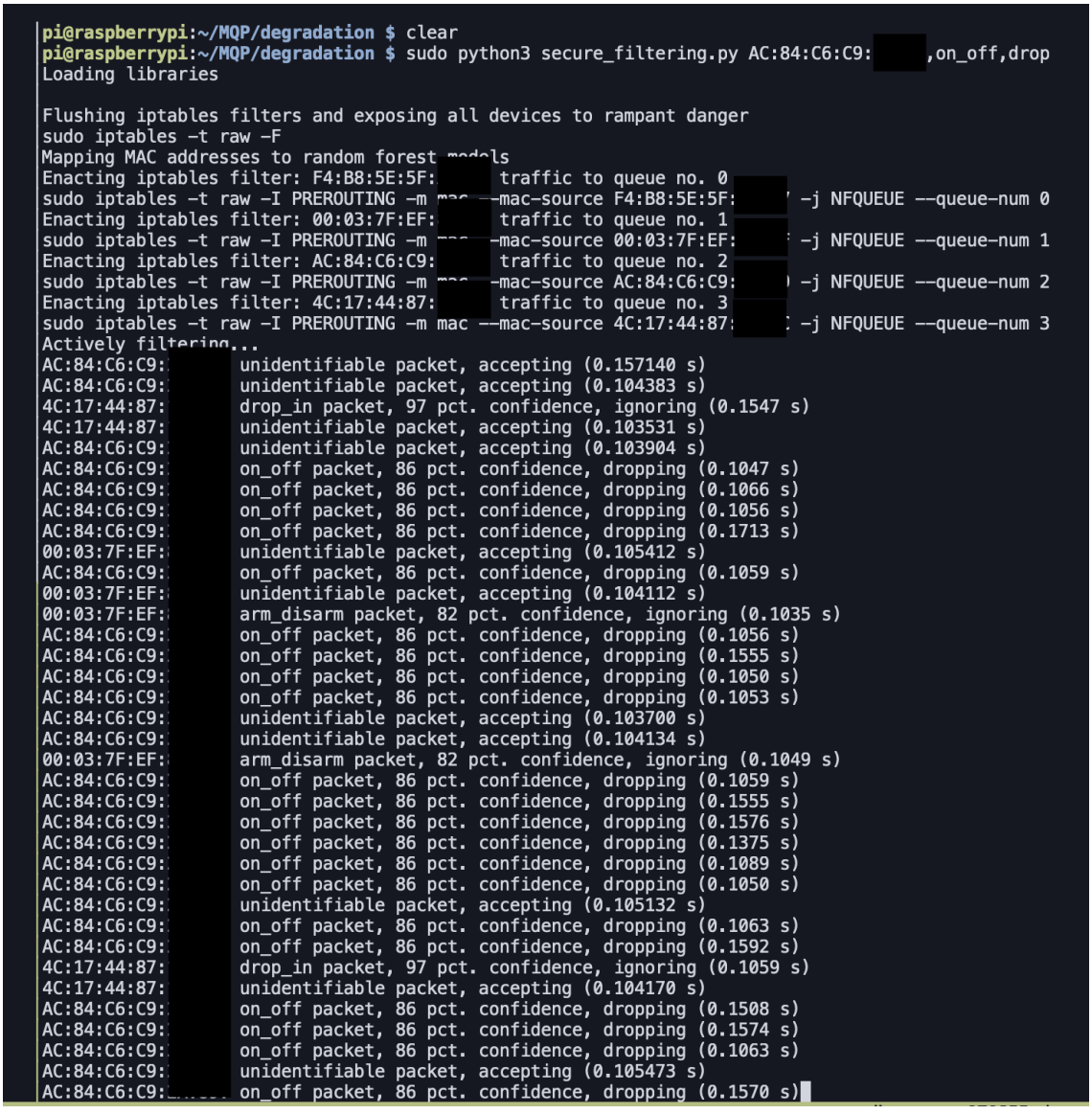

Figure 7.20: Classifying TP-Link plug on/off behavior

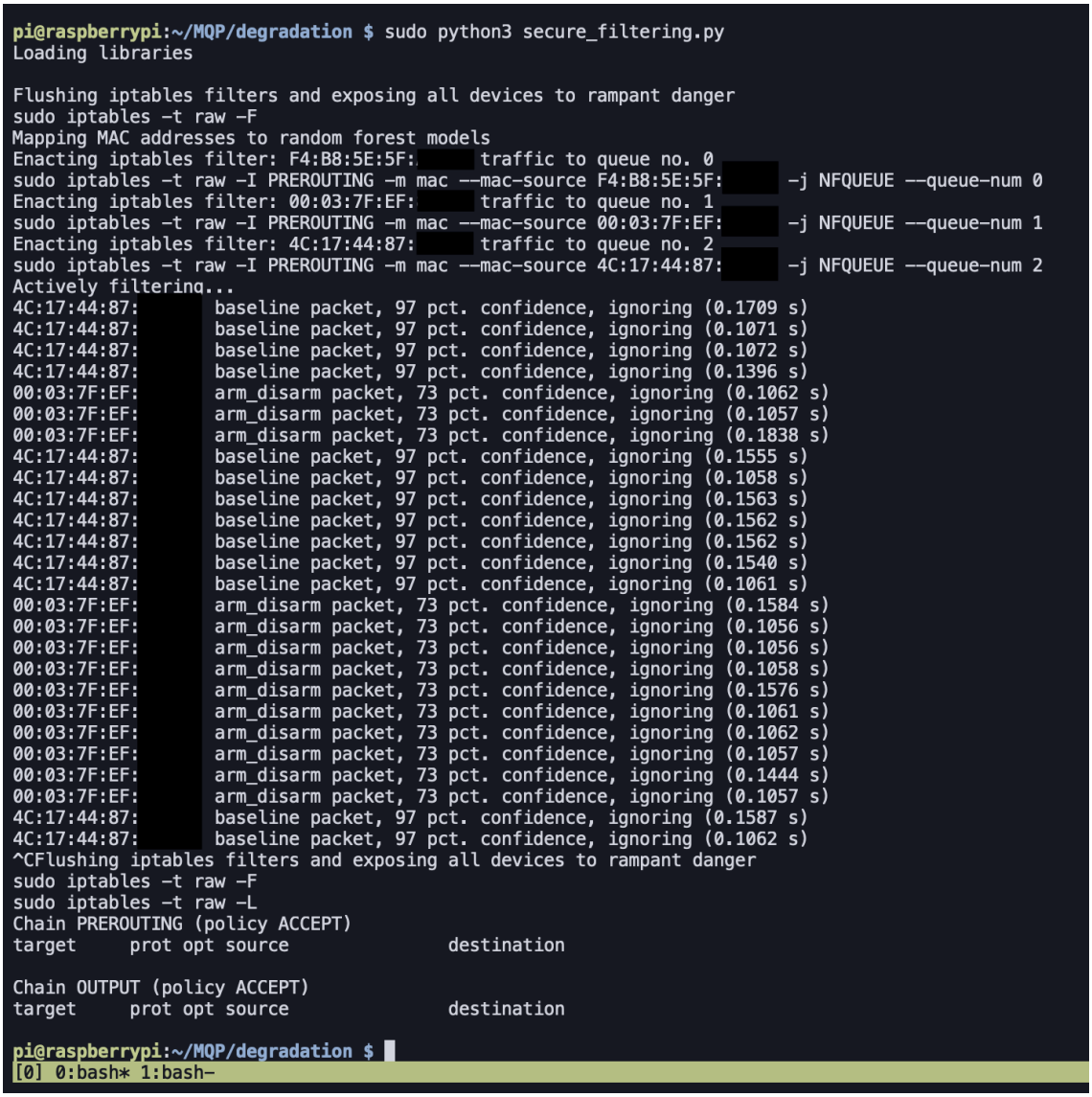

Figure 7.21: Classifying Blink camera arm/disarm behavior

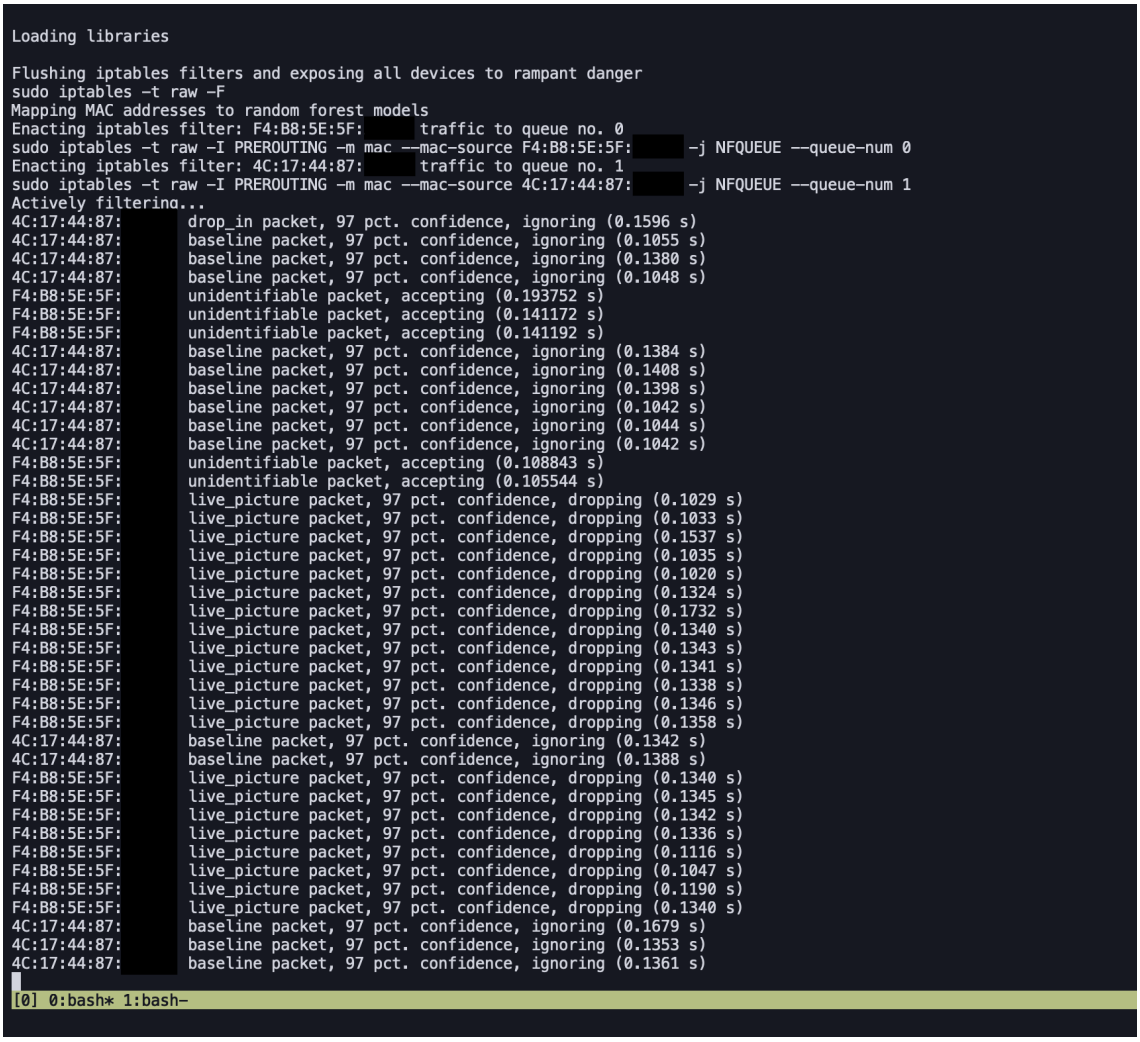

Figure 7.22: Classifying Blink camera live picture behavior

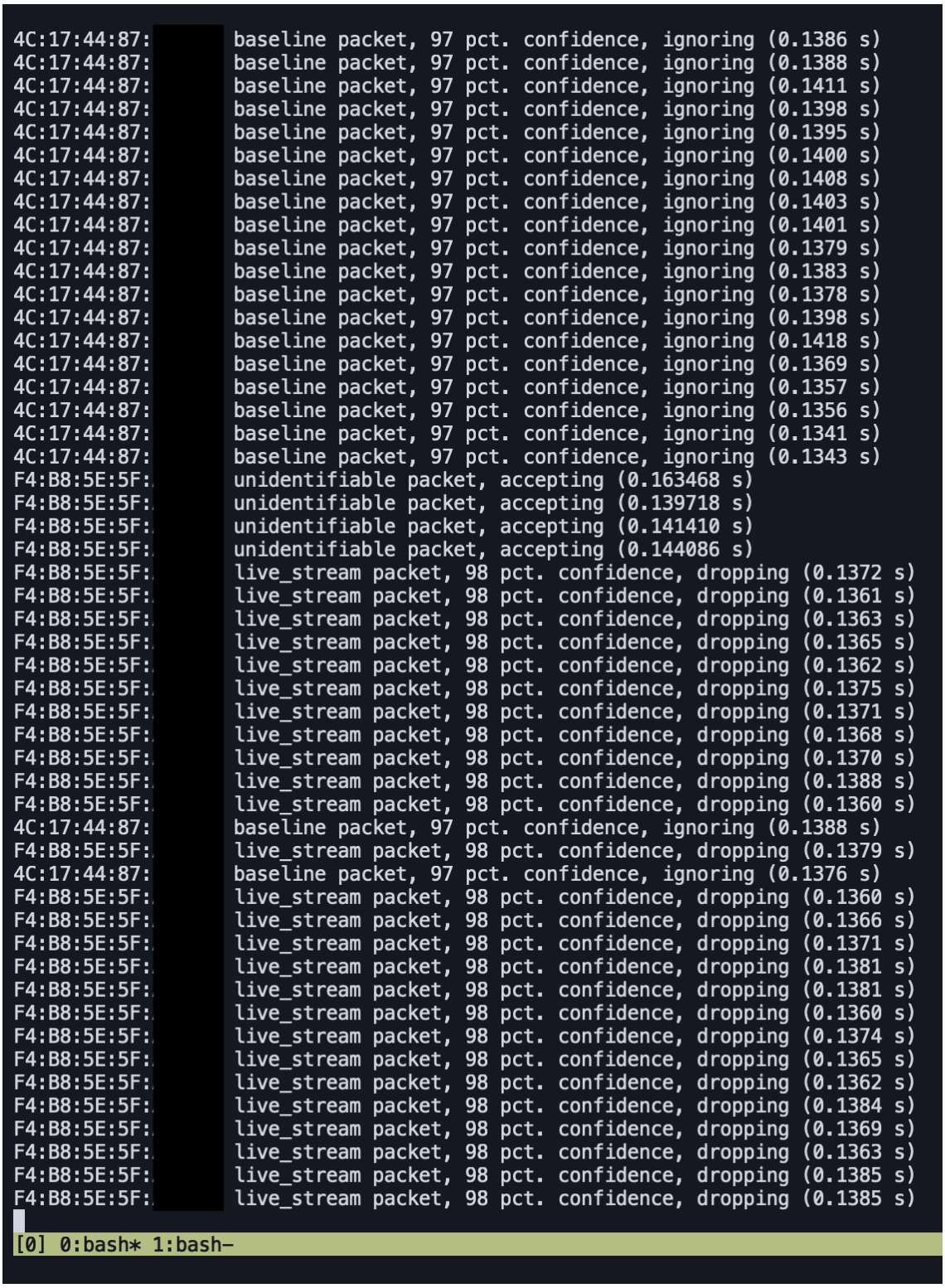

Figure 7.23: Classifying Blink camera live video stream behavior

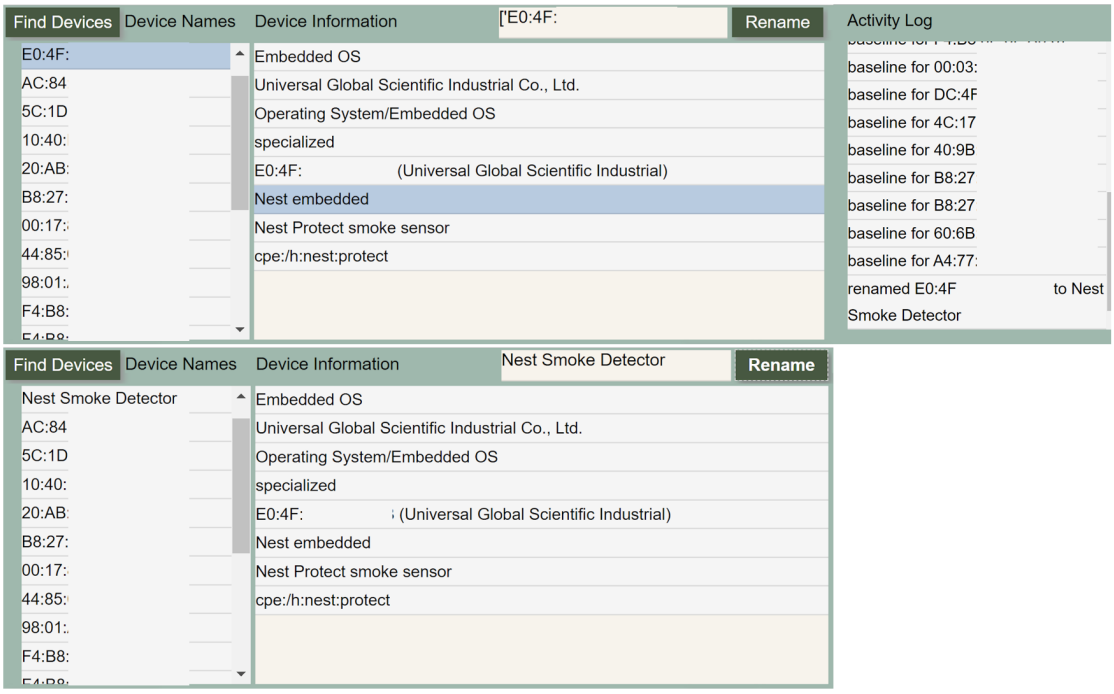

Figure 7.24: Renaming a device in the UI

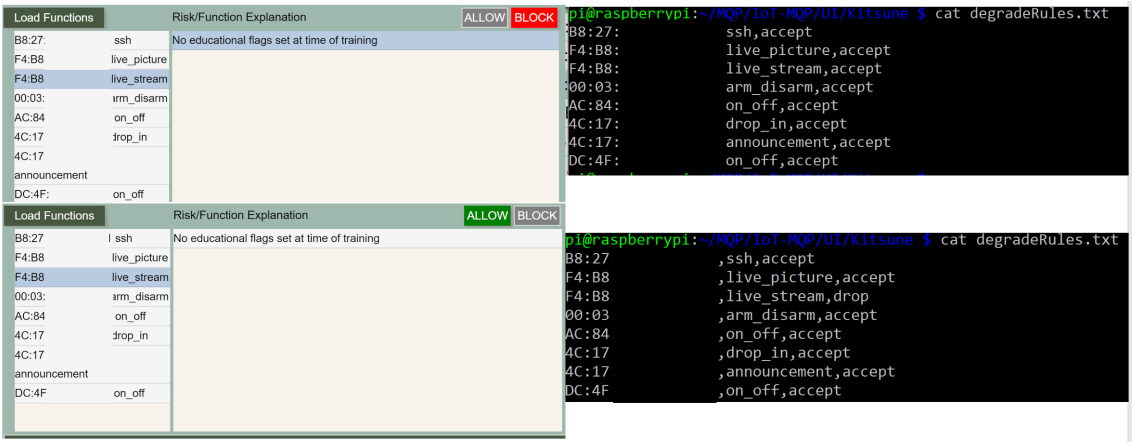

Figure 7.25: Choosing to degrade a function through the UI

Bibliography

[1] Netfilterqueue PyPi documentation. PyPi, 2011.

[2] Aircrack-ng suite. volume 1.5.2, 2019.

[3] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and A. Selcuk Uluagac. Peek-a-boo: I see your smart home activities, even encrypted. Aug 8, 2018.

[4] Amazon. Amazon Alexa capabilities API. 2019.

[5] Sumit Basu, Karthik Gopalratnam, John David Dunagan, and Jiahe Helen Wang. Active learning framework for automatic field extraction from network traffic. volume 11/567,328, Jan. 19, 2010.

[6] Danilo Bellini. watchdog. volume 0.9. PyPi, 2011.

[7] Ana Bera. 80 IoT Statistics. Feburary, 2019.

[8] Philippe Biondi. Scapy: packet crafting. volume 2019, 2019.

[9] Blink. Blink smart camera user guide. 2018.

[10] Raouf Boutaba, Mohammad Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar Caicedo. A comprehensive sur-

vey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):1–99, Dec 2018.

[11] Dominik Breitenbacher, Ivan Homoliak, Yan Lin Aung, Nils Ole Tippenhauer, and Yuval Elovici. Hades-IoT: A practical host-based anomaly detection system for iot devices (extended version). May 2, 2019.

[12] Canonical. Taking charge of the IoT's security vulnerabilities. January 2017.

[13] Alice Crelier. Trend analysis the challenges of scaling the Internet of Things. Technical report, Center for Security Studies (CSS), August 2019.

[14] CSPi. How to overcome today's industrial IoT security challenges. July 2019.

[15] Rohan Doshi, Noah Apthorpe, and Nick Feamster. Machine learning ddos detection for consumer Internet of Things devices. page 29, Piscataway, Jan 1, 2018. The Institute of Electrical and Electronics Engineers, Inc. (IEEE).

[16] Etekcity. Voltson Smart Wifi Outlet manual. volume 2019, 2018.

[17] Jerry Alan Fails and Dan R. Jr Olsen. Interactive machine learning. Technical report, May 2003.

[18] Fingerbank. Fingerbank API. volume 2019, 2019.

[19] Raspberry Pi Foundation. Setting up a Raspberry Pi as an access point in a standalone network (NAT). volume 2019, 2019.

[20] Wireshark Foundation. Wireshark. volume 3.0.7, Dec 4, 2019.

[21] Great Scott Gadgets. Ubertooth build guide. March 5th 2019.

[22] Great Scott Gadgets. Ubertooth firmware guide. May 2nd 2019.

[23] Tomer Golomb, Yisroel Mirsky, and Yuval Elovici. CIoTa: Collaborative IoT anomaly detection via blockchain. Mar 10, 2018.

[24] Google. Chromecast help. volume 2019, 2019.

[25] Shuai Guo, Zhongwen Guo, Zhijin Qiu, Yingjian Liu, and Yu Wang. Ifrat: An IoT field recognition algorithm based on time-series data. pages 251–255. IEEE, Aug 2017.

[26] Wilcox James. How to turn off smart TV snooping features. Number 9/22/19, October 24 2018.

[27] ASPj k2wrlz. mdk3. volume 3.0.6, 2015.

[28] Brian Krebs. Mirai botnet. Number 12/4/19, November 4, 2016.

[29] Brad Linder. Fun fact: Amazon's Echo devices run Android-based OS. volume 2019, May 12, 2017.

[30] Gordon Lyon. Nmap webpage. volume 2109, 2019.

[31] Marek Majkowski. How to drop 10 million packets per second. Cloudflare, July, 2018.

[32] Samuel Marchal, Markus Miettinen, Thien Duc Nguyen, Ahmad-Reza Sadeghi, and N. Asokan. Audi: Toward autonomous IoT device-type identification using periodic communication. *IEEE Journal on Selected Areas in Communications*, 37(6):1402–1412, Jun 2019.

[33] Minim. Minim products. Minim, Inc., 2019.

[34] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. Feb 25, 2018.

[35] Abdul Moiz. Implications of the Internet of Things. 2019.

[36] Brian Nadel. Bitdefender box review. Oct 11, 2018.

[37] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. Demystifying IoT security: An exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale IoT exploitations. *IEEE Communications Surveys and Tutorials*, 21(3):2702–2733, 2019.

[38] Thien Duc Nguyen, Samuel Marchal, Markus Miettinen, Hossein Fereidooni, N. Asokan, and Ahmad-Reza Sadeghi. DIoT: A federated self-learning anomaly detection system for iot. Apr 20, 2018.

[39] Doohwan Oh, Deokho Kim, and Won Woo Ro. A malicious pattern detection engine for embedded security systems in the Internet of Things. *Sensors (Basel, Switzerland)*, 14(12):24188–24211, Dec 16, 2014.

[40] The PySimpleGUI Organization. Pysimplegui. volume 4.11.00, 2018.

[41] Particle. Particle Photon user guide. volume 2019, 2019.

[42] Huuck Raf. IoT: Internet of threat and static program analysis defense. 2015.

[43] Azamuddin Bin Ab Rahman and Raj Jain. Comparison of Internet of Things (IoT) data link protocols. Technical report, Washington University in St Louis, November 30, 2015.

[44] Shahid Raza, Linus Wallgren, and Thiemo Voigt. Svelte: Real-time intrusion detection in the Internet of Things. *Ad Hoc Networks*, 11(8):2661–2674, Nov 2013.

[45] Hichem Sedjelmaci, Sidi Mohammed Senouci, and Mohamad Al-Bahri. A lightweight anomaly detection technique for low-resource IoT devices: A game-theoretic methodology. pages 1–6. IEEE, May 2016.

[46] Douglas H. Summerville, Kenneth M. Zach, and Yu Chen. Ultra-lightweight deep packet anomaly detection for Internet of Things devices. pages 1–8. IEEE, Dec 2015.

[47] Agnes Tegen, Paul Davidsson, and Jan A. Persson. Interactive machine learning for the Internet of Things: A case study on activity detection. In *International Conference on the Internet of Things*, New York, New York, October 22 - 25 2019 Oct 2019. ACM.

[48] TP-Link. TP-Link HS100 user manual. 2018.

[49] Ke Wang and Salvatore Stolfo. Anomalous payload-based network intrusion detection. 2004.

[50] Joshua Wright, Ryan Speers, and Ricky Melgares. Killerbee installation. volume 2.7.1, 2010.

[51] Miao Xie, Jiankun Hu, Song Han, and Hsiao-Hwa Chen. Scalable hypergrid k-nn-based online anomaly detection in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1661–1670, Aug 2013.

[52] Er Pooja Yadav, Er Ankur Mittal, and Dr Hemant Yadav. IoT: Challenges and issues in indian perspective. pages 1–5. IEEE, Feb 2018.

[53] Micheal Zalewski. p0f v3: passive fingerprinter. volume 2019, 2012.