University of Nevada, Reno

# AI Enabled IoT Network Traffic Fingerprinting with Locality Sensitive Hashing

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Engineering

by

Jay Thom

Dr. Shamik Sengupta/Dissertation Advisor

May 2024

THE GRADUATE SCHOOL

We recommend that the dissertation
prepared under our supervision by

**Jay Thom**

entitled

**AI Enabled IoT Network Traffic Fingerprinting with Locality Sensitive Hashing**

be accepted in partial fulfillment of the
requirements for the degree of

**Doctor of Philosophy**

*Advisor*
Shamik Sengupta, Ph.D.

*Committee Member*
Frederick Harris, Ph.D.

*Committee Member*
Batyr Charyyev, Ph.D.

*Committee Member*
Emily Hand, Ph.D.

*Committee Member*
Hanif Livani, Ph.D.

*Graduate School Representative*

Markus Kemmelmeier, Ph.D., Dean
*Graduate School*

May 2024

# *Abstract*

The ubiquity of IoT devices in both public and private networks has increased dramatically in the last few years with billions of network-connected devices appearing in every sector. In many cases these devices provide low-power and low-cost solutions to multiple problems, but this convenience comes with a price. Low-powered devices often lack the computational capacity to support encryption or other means of protection. In addition, devices are often optimized for easy connection out-of-the-box, potentially leaving them vulnerable due to unchanged default configurations. It has been shown that device IP and MAC addresses can be easily spoofed, making accounting for IoT devices within a network problematic. These vulnerabilities have led to devices being compromised by malware such as the Mirai botnet, allowing for their unintentional use as access points to protected networks, as well as participants in large scale distributed denial of service (DDoS) attacks. Much work has been done in recent times to address the problem of identification by fingerprinting network traffic using various techniques, allowing network administrators to track device membership and detect anomalous behavior. While a high degree of accuracy has been achieved, effective feature extraction and acceptable computational overhead continue to be an issue. In addition, machine learning models often require frequent modification and retraining to remain effective. We apply a combination of locality sensitive hashing and machine learning techniques to identify specific devices based on their network traffic, eliminating the need for complex feature engineering and model retraining. This approach achieves an accuracy identifying known devices as high as 98% using only a single packet sniffed from the network allowing for real-time device identification, providing a significant improvement over previous approaches. We leverage this method to assist in the real-time identification of IoT devices based on their network traffic fingerprint, providing improved security and network device accounting.

# Acknowledgement

During my journey as a computer science student and information security engineer at UNR, I have met and worked with many great people. To name a few, I would like to thank several fellow students that have since graduated and moved on; M. Abdullah Canbaz, Suman Bhunia, Khalid Bakshaliyev, Amar Patra, Raj Shukla, Prasun Dey, Paulo Regis, Tapadhir Das, and many others. In particular I'd like to thank Batyr Charyyev, we spent much time working together...time well spent. Every day is a good day!

Most of my time at UNR has been a combination of work and study, and I've very much enjoyed it. Much thanks to all of the faculty here for their friendship and assistance. I won't forget your kindness.

I would like to thank my dissertation committee Dr. Fred Harris, Dr. Batyr Charyyev, Dr. Emily Hand, and Dr. Hanif Livani for their time, insightful comments, and encouragement. Their constructive questions, suggestions, and guidance made this dissertation stronger.

I was very fortunate to be advised and to work under the supervision of Dr. Shamik Sengupta and I would like to thank him for everything he has done for me. I really have a lot to say about his ethics, his guidance, his example, and his professionalism.

Finally and most importantly, I would like to thank my family. My wife Shendry, my sons Ben, Max, Nick, and Nate, along with their wives have all been students here at UNR along side me. We've had a great time. In particular I'm grateful for the time I was able to spend working and studying with Nate, it was a huge help to me. I want to thank all of you for your patience, understanding, and encouragement. You've all been a blessing to me and I count myself fortunate every day.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In recent times the concept of an Internet of Things (IoT) has been emerging with numerous devices such as cameras, low-powered sensors, wearable technologies, and a multitude of household and industrial devices inter-connected to one another and to the Internet. We are essentially surrounded by IoT devices which play increasingly significant roles in our lives, with the average American household utilizing 22 connected devices, a number that increased significantly as a result of the Covid-19 pandemic [5]. At the enterprise level, the average network consists of 48% IoT devices, much of which is related to premise security (cameras, sensors, etc.). These trends are likely to continue, with reports indicating there are as many as 30 billion IoT devices connected worldwide [6] with continued rapid growth expected. While offering convenience and utility, there are also increased security risks with many devices under-equipped to prevent compromise.

One example of such security risk was seen in 2016 with the emergence of the Mirai botnet and its variants [7]. At its peak, Mirai was able to cripple a number of key services including OVH (the worlds largest cloud provider), Dyn (a key DNS resolution service), and the Krebs on Security website by generating distributed denial of service attacks (DDoS) of unprecedented scale. OVH reported attack traffic exceeding 1 terabyte per second, making it the largest such attack to date [8]. These attacks were perpetrated primarily through the use of low-powered and otherwise innocent Internet-of-Things (IoT) devices such as air-quality monitors, home routers, baby monitors, and web cameras. It is estimated that at its height, Mirai infected over 600,000 vulnerable IoT devices [9]. While not especially sophisticated, the Mirai botnet was very effective, and demonstrated the vulnerability present in many such devices, as well as the need for effective methods for monitoring their behavior.

Much of the vulnerability inherent in these devices rests in their simplicity. The typical IoT devices offers a limited range of functions, i.e. a smart socket simply turns power on and off, or in the case of a smart light bulb a user can control power and change color schemes. To make the devices easy to use, manufacturers often forego adequate security practice by setting the device up for 'plug-and-play' so that users can easily activate the device out of the box for a favorable user experience. While convenient, issues such as default login credentials may go unchanged, and users with limited technical experience may not even be aware of potential problems. It is estimated that the Mirai botnet was able to compromise nearly 600,000 devices with 60 username/password combinations [10]. This along with a telnet service left enabled by default by several manufacturers made IoT devices easy prey for malicious activity.

On a positive note, the simplicity of these devices also lends itself to effective device fingerprinting. Along with their limited functionality, IoT devices typically produce a limited number of network transactions and tend to perform the same duties in a repetitive manner. We find that using locality sensitive hashing (LSH), it is possible to capitalize on this repetition to identify devices by producing a recognizable fingerprint. By extension, our hybrid LSH approach in combination with machine learning extends this capability from identifying a known device in a heterogeneous group to identifying a specific individual in a homogeneous group of identical devices, and to identifying the genre (i.e. webcam, smart bulb, smart socket, etc.) of an unknown device found on a network, as in the case of a rogue IoT device. These fingerprints can also be used to detect changes in known device behavior. For instance, in the case of Mirai, while devices may be engaged in scanning or attacking activities, they will continue to perform their expected duties. Owners or administrators may be completely unaware their devices or networks are engaged in nefarious activity. Changes in the patterns of a device fingerprint can be an indicator of compromise.

As an initial research we begin by constructing a model of a Smart City, including infrastructure such as transportation, street lighting, traffic signals railroad crossings, and a mock power plant, all controlled by a fully networked programmable logic controller (PLC) which utilizes the MODBUS communications protocol. Known for their vulnerabilities, we demonstrate how the PLC can be easily hacked from outside the network and then work to secure the network perimeter to prevent malicious access. This work helped us along the path towards researching IoT vulnerabilities.

Next, we build an IoT test bed for research and education based on a software defined network model, and explore performance and vulnerabilities in connected devices. The test bed consisted of a number of virtual edge devices and a network of routers controlled by an OpenDaylight SDN controller. Experiments were conducted to explore performance and mitigation options for IoT devices in a software defined network environment. From here we begin to focus specifically on IoT devices and how they are compromised by malicious botnets.

Shortly after the initial appearance of the Mirai botnet its authors released their code as open-source (likely to evade authorities) leading to the proliferation of numerous variants, many of which improved performance and added functionality to the original code [11]. To gain a better understanding of botnet behavior, we employed a global network of honeypots to collect and analyze network traffic activity data by infected devices and the downloading of botnet software scripts to sand-boxed devices as detailed in Chapter 6 of this paper. Data collected between May 2019 and December 2021 show high degrees of repetitive behaviors associated with Mirai and its variants, as noted by [12]. It is likely that new approaches to this attack will continue to emerge in the foreseeable future requiring new methods of defense. One aspect of this defense will rely on greater security measures built into IoT devices, but there will always be a need for methods to monitor device behavior and identify their presence by security personnel and network administrators.

The first step in providing security for IoT devices and the networks on which they reside is proper accounting and management, but keeping track of IoT devices in a network can prove to be a challenging task. Identifiers such as IP and MAC addresses are unreliable as they can be easily spoofed, making it difficult to know when rogue devices join a network. Known devices can continue to perform their simple duties, with malicious background activities such as eavesdropping, scanning, or sending unsolicited traffic to other targets. Some means of detecting changes in their expected behavior is required. The topic of IoT device identification by analyzing generated network traffic has been a popular area of research in the last few years with several proposed solutions in the literature. Various approaches using machine learning techniques have been explored, wherein network traffic is collected and analyzed either in real time or offline, and models are developed for machine learning classifiers which have been able to identify specific devices and device types as well as differentiating *normal* traffic from *anomalous* traffic. This has also been helpful for highlighting when new or unknown devices connect, or when familiar devices begin behaving in unexpected ways. Many of these methods are not without their weaknesses though, requiring specific traffic intervals be collected, that adequate quantities are collected, or that classifiers are able to operate within available computational constraints. Many require feature extraction and engineering which can introduce a high degree of overhead, or high degrees of required domain knowledge to choose the appropriate features for the given classifiers. In addition, many classifiers require models be updated and retrained frequently to continue providing good results [13] [14].

Non-machine learning approaches have been explored as well, seeking to identify device similarity based on packet header information, temporal features such as traffic flow, traffic volume, and by hashing packet data. Of particular interest to us is a study by [15], wherein device fingerprints are generated using locality sensitive hashes. Locality Sensitive Hashing (LSH) is useful in performing similarity searches, and has proven to be of fundamental importance in numerous fields, such as spam email and malware detection, data mining, and content based retrieval [16]. In a similarity search, "given a collection of objects $D$ with some similarity measure $s$ defined between them and a query object $q$, retrieve all objects from $D$ that are most similar to $q$ according to the similarity measure $s$" [17]. Put more simply, a hash can be generated that is very similar for similar inputs, with minor changes causing only slight differences in the output. Details of LSH are explored in more detail in Chapter 7. When employing LSH for IoT identification, a model can be built from several samples of traffic generated by the same device, then compared to unknown traffic to find the highest degree of similarity to approximate a match. In the same sense, some threshold can be set to determine devices of similar type. This approach provides a means for identifying devices without the need for feature extraction or model retraining. Unfortunately, it has been shown that as the sample size decreases, say from a 10-minute sample to a 5-minute sample, the accuracy degrades considerably, requiring that a large enough sample be collected for reliable identification. In addition, device samples need to be stored and used for comparison when attempting to identify new devices limiting system scalability. While an effective method for device identification on a limited scale, we find the utility of locality sensitive hashing can be greatly increased when combined with machine learning methods.

To enhance LSH for network traffic fingerprinting we develop a two-stage process wherein device traffic hashes generated with a modified LSH technique are combined with machine learning to creating a unique device fingerprint. To accomplish this traffic flows collected as *.pcap* files are captured and converted to locality sensitive hashes. These hashes are then broken down into their constituent bytes and are combined into a feature vector with each byte represented as a base-10 integer. In this way we combine the power of locality sensitive hashes with machine learning for device classification. Precision, accuracy, and recall are found to be high with this method, avoiding the complexities of feature extraction, and maintaining good performance for very small traffic flow samples (single packets); a significant improvement over previous works.

## 1.1 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 will present a background on this work including a brief history of botnets and their relationship to IoT devices. This discussion will lead us to the problem we intend to solve with this work. Chapter 3 will present a review of the related works found in the literature as they relate to each chapter of this dissertation. Here we will also discuss gaps in the previous research that we intend to fill, and we also provide a taxonomy of previous research in the area of IoT device identification with network traffic fingerprinting. This research begins with the development of a Smart City testbed and an analysis of its vulnerabilities in Chapter 4. Next we develop a software defined networking testbed in Chapter 5 to analyze vulnerabilities in IoT networks. Chapter 6 will discuss the collection and analysis of malicious traffic using a global network of honeypots. Chapter 7 will cover our initial research in network traffic fingerprinting using Nilsimsa, an n-gram based locality sensitive hashing tool, in combination with convolutional neural networks to identify known heterogeneous devices. Chapter 8 will discuss hybrid locality sensitive hashing with machine learning. Here we explore fingerprinting individuals in groups of identical devices, including by device genre, and devices in the presence of of background noise. Finally, Chapter 9 will present a conclusion to this dissertation and future research directions.

# Chapter 2

# Background

## 2.1 Bot Origins

The history of botnets in many ways parallels the evolution of cybercrime itself [18]. It is marked by notable innovation, adaptation, and increasingly sophisticated tactics, driven in many cases by intelligent but simple (and often young) players. While a relatively recent phenomenon, the botnet landscape is one that will continue to influence network interactions for the foreseeable future, making IoT network traffic fingerprinting and device identification an important and relevant topic.

Prior to the *Internet* as we now know it, legitimate bots were developed to assist with simple tasks and user engagement on local machines and intranets. In the mid 1960's a program named *ELIZA* emerged as one of the first chatbots, capable of simulating conversation through the use of early natural language processing techniques [19]. ELIZA could engage users by parsing text, then by applying pattern matching it could then transform user input into questions or prompts based on pre-defined templates. While not particularly *intelligent*, it could engage users with a reasonable simulation of human conversation. In the late 1970's *USENET*, a network of discussion groups present on the early Internet was developed to automate various administrative tasks such as managing newsgroups, identifying spam, or filtering out unwanted content [20]. While not strictly speaking a bot itself, USENET embodied the automated nature that would shape the bots of the future.

With the development of the modern Internet in the late 1980's more sophisticated bots for automating tasks began to emerge such as Internet Relay Chat (IRC) bots, automated tools developed to manage functions within IRC channels such as curating user lists, conducting searches, and providing news and update services. In the 1990's more advanced service bots such as *WebCrawler* [21] and *ALICE* [22] were developed to index and search for the rapidly increasing number of web pages and resources that were appearing, and to provide more advanced user engagement with new artificial intelligence powered natural language engines. Up until this time the function of bots was primarily legitimate, but even in the early days of the Internet malicious software such as worms and viruses began to appear; precursors of the darker side of network bots we see today.

The beginning of the new millennium saw the development of the first known malicious bots and bot-nets. In 2000 a 15 year old hacker used a network of compromised college and university network computers to launch a Distributed Denial of Service (DDoS) attack against several large targets such as CNN, Dell, E-Trade, eBay, and Yahoo! resulting in $1.1 billion in damages [18]. A few years later another botnet appeared titled *Storm* which also delivered attacks with a network of as many as 2 million compromised computers, and accounted for an estimated 20% of all spam found on the Internet at that time.

In 2012 the *Carna Botnet* performed an illegal but useful scan of the Internet by leveraging unprotected or poorly protected routers to perform a mapping of the entire IPv4 address space. By using infected devices to scan for other vulnerable devices, botnet software was propagated across network routers which would then perform scans and continue propagation, as well as conducting Internet topology measurements, deleting itself on reboot. Carna discovered that hundreds of thousands of routers lacked even basic security measures, and also noted that of the 4.3 billion available addresses in existence in the IPv4 space, only about 1.3 billion were actually in use [23]. Data amassed from the Carna botnet was published to the scientific community anonymously as the process, while beneficial, was highly illegal.

## 2.2 Proliferation and Commercialization

With the continued growth of the Internet and a rapid increase in the number of targets of interest for cybercriminals, the proliferation of botnets also increased. Simple connected devices such as IoT with meager or non-existent security controls provided readily available victims that could be added to bot armies. Underground forums and black markets facilitated the commodification of botnet services, enabling individuals to rent or purchase botnets for launching cyber attacks, conducting espionage, or engaging in financial fraud.

By the mid-2000s botnets had become a pervasive threat, capable of executing a wide range of cybercrimes at scale. Notable botnets during this period included the *Storm Worm*, *Conficker*, and *Zeus* botnets, each known for their sophistication and global reach. These botnets were responsible for massive spam campaigns, data theft, credential harvesting, and other malicious activities, attacking individuals, businesses, and governments worldwide.

## 2.3 Era of Advanced Botnet Techniques

With tangible financial gain in the balance, the arms race between cybercriminals and defenders escalated as botnet operators adopted increasingly sophisticated tactics to evade detection and maintain control over their networks. One significant development was the use of fast-flux DNS techniques to constantly change the IP addresses associated with command and control (CC) servers [24]. This dynamic infrastructure made it difficult for security researchers and law enforcement agencies to trace and disrupt botnet operations effectively.

Moreover, botnets began leveraging encryption and peer-to-peer (P2P) communication protocols to enhance resilience and decentralize command infrastructure. By distributing command functions across multiple nodes within the botnet, operators reduced the risk of a single point of failure and made it harder for authorities to dismantle their networks.

## 2.4 Botnets in the Age of Cyber Warfare

The increasing prevalence of botnets did not go unnoticed by nation-states seeking to advance their strategic interests in cyberspace. State-sponsored actors began incorporating botnets into their arsenal of cyber warfare tactics, using them for espionage, sabotage, and strategic attacks against rival nations. One of the most well known examples is the *Stuxnet worm* [25], discovered in 2010, which targeted Iran's nuclear program with unprecedented precision and sophistication.

Stuxnet exploited multiple zero-day vulnerabilities and advanced propagation mechanisms to infiltrate an air-gapped system and manipulate a specifically targeted set of industrial controls. The worm, believed to be developed by the United States and Israel, demonstrated the potential of botnets as tools of geopolitical influence and coercion, leading to a new era of cyber warfare.

## 2.5 Rise of IoT Botnets

The continued rapid growth of the Internet of Things has provided a rich new landscape for botnet proliferation and exploitation. IoT devices, ranging from smart cameras to home routers, often lacked robust security measures making them vulnerable to compromise. 2016 saw the appearance and success of the Mirai Botnet, underscoring the potency of IoT botnets and highlighting the urgent need for improved security measures in connected devices, as well as tools for monitoring and detection. Subsequent IoT botnets, many of which were variants of Mirai, continued to exploit IoT vulnerabilities, posing significant challenges to network defenders and cybersecurity personnel worldwide.

One of the most infamous attacks orchestrated by Mirai targeted Dyn, a major Domain Name System (DNS) provider, in October 2016. The attack disrupted access to popular websites and services, including Twitter, Netflix, and Spotify, highlighting the disruptive potential of IoT botnets on a global scale.

## 2.6 Mirai Landscape: Evolution and Diversification

In an attempt to evade authorities, the developers of the Mirai botnet released their code as open source making it publicly available to anyone interested in producing their own botnets. The landscape of IoT botnets underwent rapid evolution and diversification as cyber-criminals capitalized on this source code to develop new variants and derivatives, each with its own unique features and capabilities.

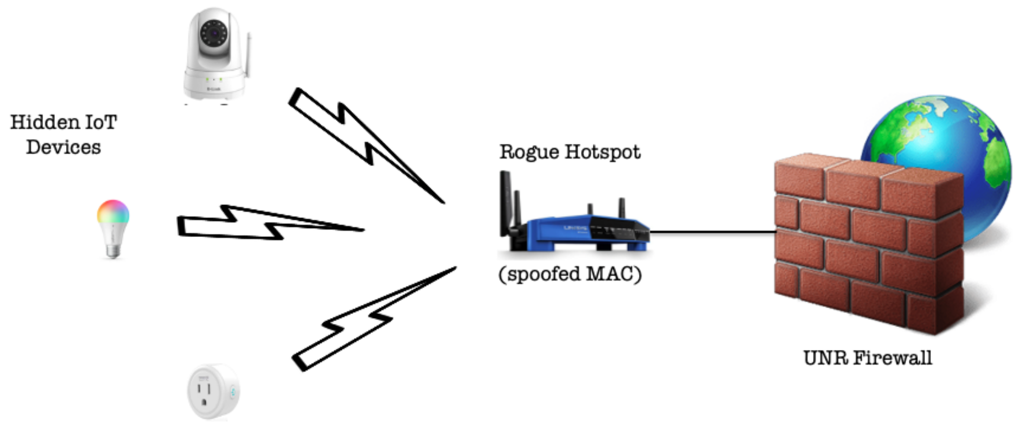


FIGURE 2.1: IoT devices can be hidden behind a rogue hotspot, effectively concealing their network membership. Only a trace of their network traffic can reveal their presence or identity.

As mentioned, many of the botnets present today are variants of the original Mirai code. One notable successor was the *Reaper* botnet, discovered in 2017. Unlike Mirai, which relied on exploiting known vulnerabilities, Reaper employed a more sophisticated approach, actively scanning for and exploiting undisclosed vulnerabilities in IoT devices. This proactive strategy enabled Reaper to infect a wide range of devices and evade traditional security measures.

## 2.7 Current Problems

One of the major difficulties for network administrators today is effectively monitoring known IoT devices on a network, and discovering unauthorized or rogue devices that may appear. For instance, in Figure 2.1, we see a scenario wherein a group of IoT devices reside behind a rogue hotspot which is spoofing its MAC address to avoid detection by administrative software. As the devices behind the hotspot are masked, the only artifact to work with is the network traffic they are producing. Through network traffic fingerprinting, packets can be randomly captured, and hidden devices can be discovered based on characteristics of their behavior and activity, and their presence and *genre* can be determined. In the same way, known and trusted devices can be monitored by traffic sampling as well. When device traffic deviates from normal parameters, it can be assumed there is some anomalous behavior possibly indicating compromise.

# Chapter 3

# Related Works

The proliferation of IoT devices connected to the network today and the threat of attack against them necessitates that research in the area of device protection be conducted. This includes an investigation of industrial controls such as programmable logic controllers, traditional and software defined networks, recent attacks, the the collection of malicious traffic data to understand attack patterns, and improved techniques for the identification and monitoring of connected devices for the purpose of accounting and anomaly detection. In this section we categorize the main approaches to these problems, and provide a review of recent works from the literature. Section 3.1 will explore works related to smart cities and smart city test beds, Section 3.2 will review works on IoT testbeds, Section 3.3 will review works related to honeypots and data collection, and finally Section 3.4 will review studies on device fingerprinting categorized by the most common approaches.

## 3.1 Security Vulnerabilities in a Smart City

Previous research on programmable logic controllers and smart city control technologies has revealed that the most common problems are related to human error and negligence. Primarily, many of these technologies are installed out-of-the-box, meaning the equipment is set up with very basic and easy-to-access levels of security [26]. With only default settings for protection, even a low-level hacker could exploit vulnerabilities and cause major disruptions. As prior research explains, these disruptions can range from stealing citizens' personal information to shutting down entire city services. Compromise can lead to massive financial and physical damages to the city itself. Successful exploits can damage the reputation of the city, harming confidence in systems and reflecting poorly on their ability to protect their citizens' information and livelihoods. An example of this was demonstrated in a recent incident wherein a hacker was able to shut down Ukraine's entire electrical grid for several hours, leaving a quarter of a million customers without service [27].

One of the most significant vulnerabilities researchers have found in smart cities is in the area of transport management systems [28]. This would include activities such as disrupting the flow of traffic or a ransomware attack on ticketing services that can completely shut the ticketing system down. As an example, it is reported that "the University of Michigan managed to hack and manipulate more than a thousand wire-less-accessible traffic signals in one city using a laptop, custom-software, and a directional radio transmitter" [29]. The research revealed that any system that relies on supervisory control and data acquisition (SCADA) software had vulnerabilities that could be taken advantage of by hackers because almost all commands sent to and from SCADAs are transmitted in plain text. Sending sensitive instructions in plain text is dangerous as data is easily intercepted and manipulated.

## 3.2 IoT Testbeds

A number of testbeds have been offered recently to facilitate IoT research and development. They are presented here by category:

### 3.2.1 Home IoT Testbeds

In [30], Yamin et. al. introduce *build it, break it, fix it* philosophy, where students are expected to design and construct an automated home IoT environment with an emphasis on secure design principles. An exercise is held during a two-day boot camp where students are separated into two groups, each constructing its own secure environment. The groups are tasked with attacking the opposing teams' environment to identify security weaknesses. Several constraints for the proposed environment are given, such as mandatory door locks, a simple user interface, cloud data storage, security alerts, etc. Pre- and post-event surveys are given to reinforce the principles learned. This type of approach is beneficial in that students play dual roles of *maker* and *breaker*, which is not normally provided by cybersecurity exercises.

Another approach to teaching home IoT systems is presented in [31], where a *smart home* environment consisting of a security system, air conditioning system, entertainment system, lighting system, and smart appliances is provided. The environment is constructed using a miniature doll house fitted with devices and sensors connected through a home controller which in turn communicates with a central management system via web sockets. The smart home server communicates in the same way with other clients such as a smart phone, a smart home assistant, etc. simulating a realistic environment for students to interact with.

The development of a *multi-dimensional* IoT testbed and associated challenges are described in [32]. The construction of a realistic and controlled environment is detailed, which supports multiple communication protocols, data processing, gateway operations, cloud integration, node deployment, and security concerns. This platform allows for interactive teaching of IoT concepts and practice-based education using Commercial Off-The-Shelf (COTS) components.

A free and remotely-accessible platform *AssIUT* is described in [33]. Unlike other testbeds that involve hundreds to thousands of IoT devices and servers to conduct advanced research experiments, AssIUT is built to accommodate students and novice researchers. A system of *layers* is defined, separating IoT concepts into groups for more effective learning. Layers are defined as *Things layer (Layer 0)* which contains IoT devices to be connected; *Sensor layer (Layer 1)* which involves sensors, actuators, etc.; *Nodes layer (Layer 2)* that includes processing nodes; *Communication layer(Layer 3)* that contains communication modules; and *Cloud layer(Layer 4)* that encapsulates large cloud platforms. Users can remotely log into a control portal and write/compile their own software solutions and upload binaries to an Arduino micro controller on reserved IoT nodes in a Software Defined network. A guide including experimental examples is included to facilitate the process. The AssIUT framework depends on custom hardware devices connected via wireless services (WiFi, LoRa, Zigbee, and 3G/4G Cellular networks) to form a software defined network, which is in turn dependent on cloud services. While very useful, it does not incorporate physical (wired) or virtual Ethernet networks, and lacks the tangible element provided by the testbed proposed in this paper. In addition, our testbed also emphasizes IoT and network security, a feature not present in AssIUT.

### 3.2.2 Security-Specific Testbeds

The design and implementation of an automated IoT security testbed is developed in [34]. The system automatically tests devices for vulnerabilities based on device type. Two basic device-types are described; those that host a website for interaction as in a web camera, and those that advertise their state to a remote server and receive instructions back to make changes, as in a smart light bulb. This categorization is then used to implement an automated device testing and vulnerability detection framework by analyzing communication patterns of devices. The system exposes devices to the Internet, and automatically disable network access when an anomalous communication behavior is detected.

ISAAC [35] is a realistic cyber-physical testbed system designed to facilitate learning and research specifically for IIoT. Recognizing that both complexity and real time interactions in many cyber-physical systems cannot be reflected by simulations alone, ISAAC provides a controlled environment for testing device resiliency to attack. The system is adaptive and re-configurable, and provides both evaluation as well as teaching opportunities.

Some of the problems associated with the heterogeneous nature of IoT are addressed in [36]. The authors develop a security testbed framework capable of evaluating devices of different types by incorporating machine learning techniques to perform standard and advanced security testings, effectively detecting compromised IoT devices.

### 3.2.3 Software Defined Networking (SDN) Testbeds

Software Defined Networking (SDN) is an important technology for IoT ecosystem in that it enables the management of network nodes through programming rather than through traditional methods involving node autonomy and system administration. This potentially provides for greater bandwidth flexibility and management of large IoT systems. Several methods have been explored to integrate SDN for IoT communication. In [37], authors propose SDN as a solution to consolidate disjoint IoT platforms wherein multiple service providers can use the same platform to supply services and share information. This allows for more rapid development of technology and optimizes utilization of resources. Their basic motivation for developing this IoT architecture is to promote the reuse of various resources and to allow the rapid introduction and deployment of new IoT services and applications. Their design principles emphasize layered architecture, openness and programmability, data provisioning and sharing at different levels, and interoperability.

Guo et. al. explain in [38] how SDN introduces a vehicle for raising the level of abstraction for network configuration, enabling the network control plane logic to be decoupled from the network forwarding hardware thus moving the control logic and state to a programmable software component; the controller. However, as networks become large, a single controller becomes a bottleneck as it is unable to manage all network elements. They propose a system in which controllers are layered vertically, with a master controller managing lower level controllers, and allowing a SDN network to become large, opening the door for better services to IoT devices.

### 3.2.4 Education-Oriented Testbeds

Numerous IoT testbeds focus specifically on education. [39] is an extension of aforementioned AssIUT testbed focusing on providing a platform to conduct student competitions. The competitions consist of registration, tutorial, missions tackling, evaluation, and scoring phases. After the announcement of the competitions, a duration of two weeks is given to teams to register in an online form. Tutorial sessions were conducted remotely in the form of video-conference online meetings to help the teams know more about testbed architecture and usage. The tutorial sessions included presentations, program demos, and question sessions from the teams.

Guo et. al. developed an IIoT testbed to allow students and researchers gain experience in security and smart manufacturing related topics with a focus on securing networked industrial systems and collecting, analyzing, and visualizing the machinery data [40]. Its stated purpose is research and education, forcing an emphasis on flexibility and ease of use. The platform is designed to be *friendly* toward the study of IIoT devices and security, and is made up of low-cost off-the-shelf components. However, while it addresses fog computing, it does not provide access to a programmable SDN controller.

As IIoT deals specifically with industrial control systems, security is crucial for safe operation of these infrastructures. Since there is a global shortage of qualified personnel with relevant skills, Celeda et. al. presented KYPO4INDUSTRY, a testbed that is customized specifically for training and education in IIoT systems for beginning and intermediate level computer science students to learn cybersecurity in a simulated industrial environment [41]. The system is constructed using open sources software and re-configurable modules to facilitate hands-on projects in a flipped classroom format.

LICSTER [42] is also a testbed built specifically for training in industrial control systems. Sauer et. al. describe three approaches to building effective IIoT testbeds; virtualized, real-world, and hybrid. Likewise, there are different tasks for which a testbed can be used. For instance, for security scenarios and attacks on Industrial Control Systems (ICS), a real-world testbed which utilizes physical processes is preferred, as it helps students to more fully understand the impact of different attack vectors on a production environment. This type of environment can be very expensive to build as it requires proprietary devices. LICSTER addresses this problem by providing a low-cost simulated environment using open source software and commercial off-the-shelf equipment.

### 3.2.5 Other Testbeds

Other frameworks developed for IoT testbeds include OpenTestBed [43], an open source and open hardware testbed that can be reproduced by anyone wishing to develop their own testbed environment. The authors attempt to share enough detail to allow interested parties to replicate their work, which includes commercially available state-of-the-art devices, and can emulate an environment which is representative of the users specific use case. The system is capable of loading arbitrary binary images on any device, and can send and receive serial bytes between them.

Finally, Raglin et. al. present a conceptual framework they call Smart CCR IoT, an IoT testbed [44]. Their goal is to design a scalable testbed consisting of IoT-based technologies, infrastructure, processes, data gathering, and location-specific information that can emulate a real-time understanding of a physical environment. The testbed is based on Arizona State University's *Blue Light Pole* system, a network of 700 public safety devices spread across the Tempe, Phoenix Downtown, West, and Polytechnic Campuses. The poles provide an emulated IoT network spanning campus, city, and regional scales capable of disseminating data from sensors, cameras, and other smart devices. It provides an environment for experimentation in optimal networking, computing, and storage technologies, along with techniques for edge computing, software defined networks, publish/subscribe data models, and emerging wireless technologies.

## 3.3 Honeypots and Malicious Traffic Collection

Honeypots have been widely used for collecting and analyzing the activities of malicious actors. They provide an effective tool for observing attacker behavior as it can be assumed no legitimate traffic should be exchanged with the honeypot services, and they have no production value [45] [46].

More than 67% of web servers and 71% of IoT devices connected to the Internet rely on Unix/Linux-based operating systems [47] [48]. In the work by Kambourakis et al. [49], regular updates to firmware for these systems are often overlooked, leaving opportunities for malicious actors to develop methods for unauthorized access and remote manipulation. In addition, source code for many well-known and scalable exploits are publicly available, providing hackers with ample resources to bypass security measures and subvert vulnerable systems. Honeypots can be placed inside of a network as a distraction, drawing attackers away from valued resources, or as stand-alone services exposing vulnerable ports for services such as SSH, telnet, HTTP, FTP, or SMTP. Services attempt to appear as legitimate to attackers, and log activity without implementing all of the service's logic and functionality, as shown by Bistarelli et al. [46]. Kumar [50] and Kyriakou [51] et al. demonstrate the advantages of deploying multiple honeypot tools and utilizing containers to produce a lightweight multi-service honeypot on a single virtual machine, server, or lightweight device (i.e. Raspberry Pi). In [52–54] examples of deployment and data collection from honeypots are detailed, and the basic functions of a botnet malware are examined based on scanning practices and the order of commands executed by an attacker once logged in.

Several open-source honeypot applications are available to emulate common services, provide limited functionality, and automatically log activity. Vetterl et al. discuss applications such as Kojoney and Klippo [55]. Other applications such as Dionaea, Whaler, and Cowrie provide access to services such as SMB, HTTP, FTP, TFTP, MSSQL, MySQL, SIP, SSH and the Docker API. Narwocki et al. [56] explain how by exposing the common ports for these services, attackers performing random scans of the Internet are often attracted to them within minutes, encouraging them to perform brute-force attacks using dictionaries of common usernames and passwords to gain access.

High value data can be collected, and detailed analysis is required to learn more about attack behavior. Fraunholz et al. [57] discuss analysis based on timing behavior by correlating the overall number of attacks with the number of unique IP addresses seen, as well as correlating the overall number of attacks with the number of attacks per unique IP address. Vakilinia et al. [58] discuss capturing commonly used passwords from brute force attacks and utilizing them as a feed for Cyber Threat Intelligence (CTI). Fan et al. [59] develop attack profiles by applying attack information to analyze malicious activity in order to unveil intruder motives. Fraunholz et al. [60] discuss the application of machine learning techniques for classifying attacks on honeypots.

A major concern is the fingerprinting and identification of deployed honeypots by attackers. Vetterl et al. [55] present a generic technique for fingerprinting honeypots at Internet scale with a single TCP packet. They conduct Internet-wide scans and are able to identify 7605 honeypot instances across nine separate implementations. They also discover most honeypot instances are not properly updated, making them even easier for attackers to identify. McCaughey et al. [61] note many open-source software tools are available to help identify honeypot devices that have been on the network for extended periods of time by noting timing differences between honeypots and actual machines. Vetterl et al. [55] discuss a project wherein they scan the Internet and discover thousands of honeypot devices. They also cover some of the legal issues involved in “logging into” honeypot machines, even for the purpose of identifying them. Cabral et al. [53] discuss how Cowrie in its standard state can be easily identified by attackers using nmap, Shodan, and OS fingerprinting, and require modification to be effective. Finally, Pitman et al. [62] discuss their tool that can quantify the ability of a honeypots to fingerprint its environment, capture valid data, deceive an adversary, and monitor itself and its surroundings.

To better understand attack behavior and to develop a more complete understanding of how adversaries are utilizing services left exposed by weak or default login credentials we collect traffic on a global scale over an extended period of time, both to amass a large body of data for the development of CTI tools, and to identify patterns in behavior as attackers access and utilize services presented by honeypots located in geographically separated regions.

## 3.4 IoT Traffic Fingerprinting

We divide related works in this section based on the various approaches to the problem that are found in the literature. These include fingerprinting with both machine learning (ML) and non-machine learning (non-ML) approaches. This section is followed by a detailed taxonomy of the various works from this same perspective which can be found in Table 3.1.

### 3.4.1 Temporal Features

In [63] Noguchi et al. monitor the state of a device within a network based on the time change pattern of the number of features which can be extracted from signals originating from the device. Several situations that limit device state detection are noted, such as changes in installation location, the presence of multiple and changing interfaces on a single device, software and OS updates, and network changes, which make it difficult to know and track the current state of a device or to detect anomalies. Their system automatically detects changes in state and identifies new devices as they join a network.

Similarly, in [64] Mazhar and Shafiq seek to characterize IoT traffic in terms of temporal patterns, volume, and target endpoints, and also consider security and privacy concerns. Their system collects traffic from over 200 home networks, helping to addresses the problem of identifying only traffic in a test bed environment by using data collected in the wild. They reveal that while smart home IoT ecosystems can appear fragmented, it is mostly centralized due to its reliance on a few popular cloud and DNS services. They note certain devices exhibit specific behaviors based on their particular functionality such as video streaming, diurnal usage patterns, etc. They also note device back-ends typically connect to a limited number of service providers making them more centralized than they appear.

In [65] Aneja et al. perform device fingerprinting based on packet inter-arrival times. A Raspberry-Pi is configured to sniff packets, and graphs are plotted for arrival times to two Apple devices, an iPhone and an iPad. A Convolutional Neural Network (CNN) is used to classify devices based on the generated graphs, achieving a device identification accuracy of 86.7%.

### 3.4.2 Network Protocol Features

Meidan et al. use supervised learning to train a multi-stage classifier in [66]. In the first stage, the classifier works to separate IoT traffic from non-IoT traffic. In the second stage it attempts to determine to which *class* of devices it belongs. Features are extracted from packets by analyzing distinct traffic flows, represented by source and destination IP addresses and port numbers from SYN to FIN. This data is then enriched with publicly available data sets such as *Alexa Rank* and *GeoIP*. Data is separated into 3 sets; two for training (single session and multi-session classifiers) as well as a test set for verification. They achieve a device identification accuracy of 99.28%.

In [6] Ullah and Mohmoud focus on the set of application layer services a device uses, such as ARP, SSL, LLC, EAPOL, HTTP, MDNS, and DNS to develop a static view of device behavior. Wireshark is employed to capture test data from a test bed, and both packet headers and payload are considered. Five steps are followed to identify devices; monitoring, building of a sensor profile, analysis of results, device classification, and prevention & recovery. 3 *K-fold* cross-validation tests are used to measure feasibility and model over-fitting.

In [67] Chowdhury et al. perform device identification through fingerprinting by extracting features from single TCP/IP packet headers. A feature vector is developed by generating a score for all available features based on variability, stability, and suitability of each bit. WEKA tool is then utilized to sample various ML algorithms for device classification. They test their work on two publicly available data sets (UNSW and Iot_Sentinel), and achieve an accuracy of 97.78%.

### 3.4.3 Initial Connection Phase Fingerprinting

Miettinen et al. develop IoT SENTINEL [68] with the goal of restricting communications within an IoT network to prevent an adversary from connecting to vulnerable devices, or to use a compromised device to communicate with or exfiltrate data from other vulnerable devices on the network. A series of fingerprints are constructed from devices while they conduct their initial connection phase, which are then mapped to device types. Rules can then be applied to device types based on external information about their potential vulnerability. Features used for developing fingerprints are extracted from IP packet headers, and do not include payload information to avoid issues with encrypted traffic. The method claims an overall average accuracy of 81.5%, but traffic must be captured during the initial connection phase.

Similarly, in [69] Marchal et al. seek to define policies for various classes of IoT devices based on device type. By monitoring network traffic, *AuDI* autonomously fingerprints devices in any state of operation *after* the initial connection phase. Since the goal of *AuDI* is network management, device classes rather than individual devices are identified as abstract *device types*. Once fingerprints are captured, an unsupervised clustering algorithm is used for classification. Policies are then formulated for appropriate device limitations which are stored in a database. These are used for anomaly detection, network resource allocation, and identification and isolation of vulnerable devices.

### 3.4.4 Flow-Based Feature Extraction

In [70] Salman et al. seek to manage security restraints and requirements for IoT by identifying connected devices through the extraction of statistical features from their generated network traffic. Their proposed framework extracts simple features per packet based on any network flow of 16 consecutive packets, allowing for device identification in real time. Extracted features include packet direction, size, timestamp, and transport protocol. To classify devices the authors utilize decision tree, random forest, recurrent neural networks, residual neural networks, and convolutional neural networks.

Silvanathan, Gharakheili, and Sivaraman develop a set of one class clustering models in [71] to identify devices from a real-time flow level telemetry. The primary features used for model training are average packet size and average flow rate. Per-flow packet and byte counts are captured each minute, and attributes are computed at time granularity increments of 1, 2, 4, and 8 minute intervals. The authors use their own packet-level parsing tool, which takes in raw *.pcap* files as input, develops flow tables, and exports byte and packet counters of each flow. finally, a vector of attributes is generated each minute which corresponding to each device. K-Means algorithm is utilized to identify clusters, which are then used to identify device instances.

In [72] Bao, Hamdaoui, and Wong address the issue of white listed, new, and anomalous devices in an IoT network by using a hybrid deep learning approach. The method combines deep neural networks with clustering to enable the classification of both previously seen and unseen devices, and employs an auto-encoder technique to reduce the dimensionality of the resulting data set. An unsupervised data clustering algorithm called OPTICS is utilized which is based on space density. The auto-encoder is a symmetrical artificial neural network for reconstructing a given input. Known devices are identified using a random forest classifier based on an input vector with 297 features extracted from the network traffic.

Similarly, Hamad et al. [73] address the problem of IoT device identification by assembling a series of packets from network traffic flows, and extract various features to create a fingerprint for devices. Supervised machine learning algorithms are then applied on these features to solve a multi-class classification problem. Once devices are identified, rules can be applied to restrict their privilege to communicate on the network. In this way, a white list of known devices can be applied, and abnormal or unwanted traffic can be identified and blocked. A vector of 67 features are selected, and algorithms such as Adaptive Boosting, Latent Dirichlet allocation, K-Nearest Neighbor, Decision Tree, Naive Bayesian, Support Vector Machine, Random Forest with 100 estimators, and Gradient Boosting are applied. In addition, summary statistics on network features are calculated.

### 3.4.5 Behavioral Fingerprinting

In [74] Bezawada et al. ask the question *what is this device* and *is this device the one it claims to be*. To accomplish this, the authors perform device behavior fingerprinting by extracting features from their network traffic. They attempt to approximate the device type based on an analysis of a collection of the protocols used, and by observing command and response sequences elicited from a device via its smart phone app. Features extracted include both packet headers and payload-based features. Various machine learning classifiers are employed from the *scikit learn* suite to identify device types. The authors claim a device can be identified with as few as 5 packets from a single device.

Seeking to identify anomalous traffic, Gill, Lindskog, and Zavarshy [75] develop a baseline *normal* profile using traffic discovery and classification, then work to detect traffic anomalies based on the following features; overall traffic, transport layer protocol traffic, traffic by destination socket, packet size, and IP flow. Variations on statistical measurements of these five areas are used to formulate a profile that is considered to be normal, with variations that fall outside of these parameters identified as anomalous.

Also interested in device behavior, Yousefnezhad, Malhi, and Framling [76] analyze packet header information to capture statistical information, and combine this with sensor measurements to form a feature set that is used in classifying IoT devices. Their method is used in both normal and under-attack scenarios, and are able to identify devices with a high degree of accuracy. Seven behavioral profiles are developed that are used to develop rules for network security. Various machine learning algorithms are applied depending on the degree of linearity of the captured data.

### 3.4.6 Limited Feature Extraction/Engineering

To address the problem of feature selection, Fan et al. [59] work on IoT device identification using semi-supervised learning. The main advantage of this approach is that rather than having to determine a large set of features for model training, a smaller set of labeled data is used along with a much larger body of unlabeled data, with labels inferred from the smaller set. To facilitate this, labels from the labeled set should adequately differentiate devices from one another. In addition, IoT devices should be differentiated from non-IoT devices that exist on the network to improve model accuracy. Features chosen include time intervals, traffic volume, protocol features, and transport layer security features. Transformed features are clustered per class to compensate for feature fluctuation. Convolutional Neural

Networks are used for dimension reduction and multi-task learning is employed, helping the classifier to distinguish IoT devices from the rest of the traffic. The authors claim to achieve an average accuracy of 99.81%.

For real-time monitoring, Aksoy and Gunes [77] develop a framework they call *SysID* which randomly selects single packets for analysis using a genetic algorithm (GA) to determine which features are relevant for classification. Features are then run on several different machine learning algorithms such as Decision Table, J48 Decision Trees, OneR, and PART for classification and identification. In this way, rules can be applied to limit communication from the gateway or firewall to the IoT device to provide device-appropriate security. SysID can perform single packet analysis on network traffic selecting features without the need for expert input, and achieves over 95% accuracy.

In a similarly innovative work Charyyev and Gunes [15] use locally sensitive hashes to determine similarity in network traffic flows, a useful technique in that feature extraction is not required. Hashes are generated and compared for similarity using the Nilsimsa hashing tool, achieving a high degree of precision and recall when compared against a set of devices used to generate traffic flow samples in the same network. The system is tested on traffic collected from 22 devices by the authors, as well as on other publicly available data sets. Variable lengths of traffic flows are tested to determine how much traffic is sufficient to generate a good fingerprint. We take this approach further by extracting each byte from generated hashes and use them as features to be used with various other machine learning approaches to improve performance.

### 3.4.7 Textual Features and Data Mining

In [78] Ammar, Noirie, and Tixeuil use a *Bag of Words* approach to identify IoT devices on a network. A series of feature sets are collected both actively and passively (depending on the device) from packets containing service discovery protocols, DHCP fingerprinting, and user agent information found in the HTTP headers. This textual information is modeled as binary data using Bag of Words. Devices are represented by a feature vector of $m$ words, and is set to 1 if a word is present in a device description, and 0 if it is not. Vectors are added to a database to construct an $p\ X\ m$ matrix where columns represent words and rows represent devices. These are then used as device labels.

Feng, Wang, and Sun [79] seek to build annotations for device types, vendors, and product names by leveraging application layer response data from IoT devices and relevant websites to obtain product descriptions. Honeypots are also used as input for eliciting response data from offending IP addresses found there. These annotations are used to build a rule set for devices within a network to control and restrict access and communication to and from devices.

In a separate work, [80] Ammar, Noirie, and Tixeuil utilize a supervised learning classifier to differentiate devices based on features from network flow, as well as textual features from packet payloads. Devices are identified within a home network to determine their specific capabilities. This method seeks to identify a device as it joins the network and is in its connection setup phase. Features that are captured include maximum and minimum packet length, average packet inter-arrival time, number of packets in a given flow, protocols used in the flow, device manufacturer name, model, friendly name, XML description, mDNS information, device OS, and device model. Textual features are modeled as a bag-of-words, with each device description making up a feature vector.

In a unique approach, Kotak and Elovici address the issue of BYOD, employees connecting their own personal devices to an enterprise network [81]. In this scenario, a white list of permitted devices must be constructed to identify and assign rules to connected devices, and non-permitted devices must be identified.

To accomplish this, the authors collect TCP traffic (UDP and other protocols are ignored in this study), which are processed to remove packet headers, and the resulting data is converted into a gray scale image with each pixel representing a hexadecimal value from the binary file. A single class classifier is applied to white listed devices, while a multi-class classifier is applied to non-white listed devices to identify anomalous traffic.

Finally, in [82] Desai et al. focus on the cost of choosing and selecting appropriate features for applying machine learning algorithms to network traffic for device identification. Their framework selects features based on their utility for meeting the needs of specific algorithms. Once features are selected, popular ML algorithms are applied. A statistical method is also utilized to screen features for their relative value.

## 3.5 Contribution to Current Research

Our main contribution to the current research centers around the identification of IoT devices based on their network traffic fingerprint. Many of the examples cited here achieve a high degree of success in identifying IoT devices through a variety of machine-learning based approaches, and leverage various aspects of the available data found in device traffic and metadata for model building. However, it is apparent that most require complex feature extraction and engineering to identify the appropriate data elements to work efficiently with a given machine learning approach. Another issue with many previous studies is the need for frequent model updating to compensate for traffic variations due to changes in the devices, such as periodic updates. In addition, several depend on capturing specific segments of network traffic, i.e. traffic generated when the device is first connected and is conducting its setup phase. Others are effective, but require an adequate amount of data is captured to generate a reliable fingerprint, and may not scale well as networks or sample sets of IoT devices grow in size. In addition, many are not equipped to provide for real-time monitoring and analysis, limiting their practical usefulness in network administration.

While achieving a higher accuracy and F1 score than many of these works, the main contribution of our approach is that it eliminates the need for complex feature extraction and engineering, can develop a resilient fingerprint that does not require frequent model retraining, scales well, and can identify devices with a high degree of accuracy using only a single network packet, allowing for real time monitoring and analysis with a very low computational overhead. We believe this approach can be applied to the problem of IoT device fingerprinting and identification with an increased simplicity and resilience, making it possible to conduct network device monitoring in a usable manner.

In addition, there were two areas that were generally neglected by the previous studies that are important when considering the implementation of IoT network traffic fingerprinting in a *real-world* scenario that we attempt to address. First, the major studies in this field use data consisting of heterogeneous devices, while in a live scenario, it is likely there would be multiple of the *exact same* device, i.e. 10 identical web cameras. With a group of heterogeneous device data with no other data, it appears a bit like “shooting fish in a barrel” in that, we know there are a fixed number of devices in the set, and we can map a sample to one of these devices. In this work we identify groups of homogeneous devices of the exact same make and model, a much more difficult task.

Secondly, previous studies work with data in a laboratory environment devoid of the kind of background noise that would always be present in a live network. To create an environment where we can test the efficiency of our approach in a more realistic setting, we introduce three types of noise in large volume; randomly generated noise, noise from unknown IoT devices, and live network noise. We show that our system of fingerprinting and identification works well even with these mitigating factors.

As will be mentioned chapter 9 on future research directions, we hope in the future to run this as a live framework for detecting and monitoring IoT devices, demonstrating the value of IoT network traffic fingerprinting as a means of security management when dealing with large numbers of such device. The greater the realism in this study, the more likely such a framework would be effective.

## 3.6 Taxonomy of Related Works on IoT Network Traffic Fingerprinting

Table 3.1 provides a taxonomy of significant related works in the area of IoT device identification through network traffic fingerprinting.

Table 3.1: Summary of related works based on identification method and key feature set

| Approach | Focus | Algorithm | Ref | Performance | Details |
|---|---|---|---|---|---|
| Non-Machine Learning | Temporal Feature Extraction | Euclidean Distance | [9] | 75% accuracy with packet header information. 89% with TCP header and a lower layer header or using HTTP header and lower layer in 4 homogeneous devices. | Focus on communication features found in packet headers. Similarity between samples measured by digitizing features then comparing Euclidean distance. The most unique features found in device traffic are given a greater weight and are used to identify traffic from the same device. Limitations can occur based on OS or software updates or changes in the network. |
| | | Comparison with expert rule set | [10] | Study focuses on dataset development using traffic from an unspecified number of home IoT devices. | Devices characterized by temporal traffic patterns, traffic volume, and target endpoints. Focus is on developing a dataset of live devices rather than devices in an artificial lab environment. Fingerprinting driven by an expert rule set. |
| | Rule-Based Feature Extraction | Statistical Analysis | [11] | No accuracy measurement. Study used a large body of botnet traffic. | Seeking to identify anomalous traffic, a baseline for normal behavior is built by traffic discovery and classification. Features such as overall traffic, transport layer protocols, dest. socket, packet size, and IP flow identify anomalies. |
| | | Natural Language Processing | [12] | Rule precision between 95.7% and 97.5% with a large body of heterogeneous and homogeneous devices. | Authors build annotations for device type, vendors, and product names by using application layer response data from IoT devices and relevant websites to obtain product descriptions, natural language processing and data mining are incorporated to drive an Annotation Rule-Based Engine (ARE). Annotations are used to build a rule set to control and restrict communications. |
| | Device Traffic Hashes | Bitwise Difference | [6] | Avg. precision and recall of 93% and 90% respectively using 22 heterogeneous devies. | Hashes of device traffic flows are generated using Locality Sensitive Hashing. Similarity metrics are used to determine whether traffic samples were produced by same device. Flow lengths are tested to find minimally accurate sample size. |
| Machine Learning | Temporal Feature Extraction | CNN | [13] | 86.7% accuracy achieved in 2 heterogeneous devices. | Heterogeneous traffic is collected using a Raspberry Pi. Device fingerprinting based on packet inter-arrival time with graphs generated by a CNN. |
| | Network Layer Features | Supervised Learning | [14] | Overall accuracy of 99.281% across 13 heterogeneous devices. | Two stage classifier; first stage identifies 'IoT' or 'Non-Iot'. Second stage attempts to identify one of 13 heterogeneous devices. Features are extracted from packet flows based on source and destination IP from SYN to FIN. Data is enriched with Alexa Rank and GeoIP information. |
| | Application Layer Features | Decision Trees | [1] | Authors claim 100% precision, recall and F-score for 3 heterogeneous devices. | Application layer features such as ARP, SSL, LLC, EAPOL, HTTP, MDNS, and DNS are captured to develop a static view of device behavior. |
| | Initial Connection Phase | Random Forest | [4] | Average accuracy of 81.5% in a set of 27 heterogeneous devices. | A fixed set of link and application layer features are captured during device initial connection phase to construct a device gateway to limit communication. |
| | Device Genre Identification | Unsupervised Clustering | [15] | 98.2% accuracy with 33 heterogeneous devices. | Authors define policies on device type rather than on specific devices. Traffic is collected after initial connection phase to identify device classes. |
| | Flow Based Feature Extraction | Ensemble Approach (multiple) | [16] | 94.5% accuracy for device type, 93.5% accuracy for traffic classification, 97% accuracy for the identification of anomalous traffic utilizing 7 heterogeneous devices. | Statistical analysis applied to flows of 16 consecutive packets, extracting 4 simple features; packet size, traffic direction, timestamp, and transport protocol. Utilizes decision trees, recurrent neural networks, random forest, residual neural networks, and convolutional neural networks. |
| | | K-Means | [17] | Overall accuracy of 94% in a set of 10 heterogeneous devices. | Set of one-class clustering models from a real time flow level telemetry. Primary features are average packet size and average flow rate. |
| | | Clustering and Deep Neural Networks | [18] | Avg. accuracy of 91.2%, 92.9%, and 81.8% with input features of 297, 234, and 179 respectively in 10 heterogeneous devices. | Clustering is combined with deep learning to classify devices. 297 features are extracted and used to build a classifier. An auto-encoder is used to reduce the dimensionality of the data set. Results are used to build a device white list. |
| | | Ensemble Approach (multiple) | [19] | White listed devices identified with an accuracy of 90.3% using 27 heterogeneous devices. | 67 features are extracted from a series of packets in a network traffic flow to build a one class classifier. Algorithms include adaptive boosting, latent dirichlet allocation, KNN, decision tree, naive bayesian, SVM, random forest. |
| | Image Analysis | Representation Learning | [20] | 99.86% accuracy using 10 heterogeneous devices. | Device traffic is collected from single TCP sessions. The hexadecimal value of the packet payload is taken from a .pcap file and is converted into an image. Image serves as a fingerprint for identifying traffic from the group of 10 devices. |
| | Single Packet Identification | Genetic Algorithm | [21] | 95% accuracy in a dataset of 23 heterogeneous devices. | 212 features are extracted from 23 devices and identification is done using single packets. Genetic algorithm is used to identify relevant features. |
| | | WEKA ML Analysis Tool (various) | [22] | 46 heterogeneous devices, achieves an average accuracy of 97.78%. | Features are extracted from TCP headers. Feature vectors are developed by generating a score for all available features based on variability, stability, and suitability of each bit. WEKA tool is then utilized to sample ML algorithms. |
| | Behavioral Analysis | Scikit Learn (various) | [23] | 14 heterogeneous devices are tested yielding an average accuracy of 99%. 5-packet min. | To answer the questions "what is this device" and "is this device what it claims to be", the authors perform device fingerprinting by extracting network protocols used from packet headers as well as information from payloads. They also observe command/response sequences from device to smart phone. |
| | | Ensemble Approach (multiple) | [24] | Accuracy between 74.12% and 93.91% in a set of 6 sensors. | Packet header information is analyzed for statistical information which is combined with sensor data to create a feature set. Method is tested in normal and under attack scenarios. Seven behavioral profiles are built to develop device access rules. ML algorithms are chosen based on linearity of the captured data. |
| | Limited Feature Extraction | Semi-Supervised Learning | [25] | An average accuracy of 99.81% is accomplished using a set of 25 heterogeneous devices. | Rather than determining a large set of features, a smaller set is used along with a larger body of unlabeled data, with labels inferred from the smaller set. Features extracted include time intervals, traffic volume, protocols, and transport layer security features. CNN's are used for feature space reduction. |
| | Textual Features and Data Mining | Bag of Words | [26] | 31 of 33 heterogeneous devices correctly identified. | Features are extracted from packet service discovery protocols, DHCP fingerprinting, and user agent information from HTTP headers. Vectors are added to a database forming a matrix where columns represent words and rows represent devices. |
| | | Decision Tree | [27] | 97% average accuracy in a 10 heterogeneous devices. | Features are extracted from a network flow as well as textual features from packet payloads. Packets are collected as devices join a network and performs its connection setup phase. Features are modeled as a bag-of-words with each device description making up a feature vector. |
| | Feature Cost Metrics | Various Algorithms | [28] | Average accuracy of 96.8% using 15 heterogeneous devices. | Authors focus on the cost of selecting appropriate features for developing a classifier. Features are selected on the basis of their utility for meeting the needs of specific algorithms. 111 features are collected from a set of 15 devices. |

# Chapter 4

# Vulnerabilities in a Smart City

## 4.1 Introduction

Implementations of smart city technologies are rapidly increasing around the world, allowing for the interconnection and cooperation of multiple devices within a system. The main goal of these technologies are to increase the well-being and resource efficiency of a municipality's citizens. With systems interconnected, a safer and more efficient environment is made possible. These smart technologies include a wide range of sectors spanning from smart traffic controls, parking, street lighting, public transportation, energy management, water management, waste management, and overall physical security [27]. With all sectors interconnected it is expected that residents would enjoy a greater quality of life. Furthermore, implementation of smart technologies should serve to reduce crime, prevent terrorism, and avoid civil unrest. The benefits are clear and have been shown to accomplish these goals when implemented correctly [83].

Unfortunately, when implementing new smart city technologies one of the most crucial pieces of infrastructure is often overlooked: the network integrity on which these technologies reside. Many of these systems are easily manipulable and have become an attractive target for hackers. Recently, there has been an uptick in cyber attacks on smart cities [26]. The three primary forms of attack include availability attacks, confidentiality attacks, and integrity attacks. Availability attacks attempt to close or deny service to a system, confidentiality attacks attempt to steal information or surreptitiously monitor activity, and integrity attacks attempt to enter a system to alter information and settings [29]. As a consequence of attempting to make cities as interconnected and safe as possible, smart city developers have also opened a wide range of new security concerns.

In this research we utilize the IoT test-bed built at the University of Nevada, Reno, to demonstrate how easily the software and hardware systems supporting smart city technologies can be exploited. With the exposed exploits, we show how the vulnerabilities can be easily be avoided and remedied when implementing correct and appropriate security measures.

## 4.2 Background

Previous research has placed primary emphasis on exposing the vulnerabilities of smart cities. One of the greatest challenges in conducting research on this topic has been replicating the smart city itself. Attempting to discover vulnerabilities in an actively employed smart city could lead to devastating consequences; therefore, any educational attempt to discover vulnerabilities must be done on a simulated environment. Research of this nature is typically conducted through the utilization of virtual test-beds.

Several simulated test-beds have been proposed in the literature. One of the most well-known and heavily researched test-bed projects is the SmartSantander Project [84]. Its name being derived from its original location, the Santander test-bed Project is located in the city of Santander, Spain. The Santander test-bed consists of IEEE 802.15.4 devices that are used to replicate wireless personal area networks (WPANs), GPRS modules, and joint RFID tag/QR code labels deployed at various locations in the city. It supports several applications concerning environmental monitoring, precision irrigation, augmented reality, and participatory sensing. Within Santander, the primary research goals relate to the implementation of the different applications; there is less emphasis on network security and the subsequent ramifications of proposed attacks on the network of the smart city.

Another test-bed example comes from Antwerp, Brussels. This test-bed setup, which has been replicated by numerous groups with various changes, allows for experiments on the network level wherein researchers have deployed their network protocols on top of existing nodes. They then evaluate their solutions in a realistic city-wide network [85]. It also facilitates experiments at the data level, allowing for research on the nodes implemented and provides for continual monitoring of the city's parameters. The Antwerp test-bed allows experiments on the user level providing for input on novel smart city applications.

Finally, there is literature based on test-beds setups most similar to the setup developed at the University of Nevada, Reno. This test-bed is on a much smaller scale, and is referred to as an *educational, research driven IoT test-bed* [86]. Test-beds of this nature subscribe to the build it, break it, fix it philosophy [86]. Our test-bed serves as a training ground for students who are aspiring to understand attacker behavior by scanning and foot printing network environments. Additionally, the test-bed provides an environment for students to practice utilizing and identifying honeypot devices. It relies on open-source tools and commercial off-the-shelf materials to emulate a real-world IoT environment. The smart city test bed at the University of Nevada, Reno is designed to accommodate multiple users performing research and analysis on IoT device networking and security. It provides a complex multi-layer network topology, a software-defined network, and numerous physical and virtual devices emulating real and decoy machines. While the university's smart city is not as complex as other test-beds in the literature, it does allow for testing on a smaller scale which could then be translated and tested further on much larger projects similar to those in Antwerp and Santander. A benefit to the smaller scale setup is that it reduces the processing of overwhelming amounts of data that can come with larger-scale test-beds.

From the test-beds presented, there is a general consensus that the vulnerabilities in smart cities are urgent and need to be addressed immediately; most of the vulnerabilities are easily exploitable. Research has found that the security capabilities of IoT devices are highly variable. Some systems are lacking the computational capacity to manage encryption or to properly manage basic access credentials such as usernames and passwords. Other systems are susceptible to infection from malware and firmware modification [26]. As IoT networks increase in complexity, the risk of exposure proportionally increases. The networks can expose a large attack surface and numerous vulnerabilities. In the Journal of Urban Technology, Kitchin and Dodge discover at least 14 different attack surfaces within the IoT networks, ranging from mobile applications to various device web interfaces [29].

While there is a variety of research on the vulnerabilities of smart cities, there is less concrete literature about potential solutions or patch options. Most research attributes outdated control systems, that contain legacy components, to the smart city vulnerabilities. These legacy components use outdated software which has not been regularly patched [26]. Remedies to common vulnerabilities can be categorized into technical implementations, preparation tactics, and educational resources.

The technical implementations involve five primary approaches. First, it is suggested to use proper access controls such as usernames and passwords that adhere to secure standards, two-stage authentication, and bio metric identifiers. Next they recommend proper maintenance and firewall placement. Prior research also encourages strong end-to-end encryption. Then virus scanners and removers, generally known as malware checkers, are expected to be implemented. Finally, the literature suggests the establishment of consistent procedures to ensure routine software patching.

In addition to the technical implementations, it is recommended that cities diligently monitor activity and prepare for cyber-attacks in advance. Through consistent monitoring, systems will be able to rapidly detect and then eradicate intrusions. Preparing for cyber-attacks is significant because the city cannot afford to be completely offline in the event of an attack. Tactics to execute monitoring and preparation include: responding with urgent updates to close exploits as they occur, auditing trails of usage and changelogs, having effective offsite backups, and establishing emergency recovery plans.

Finally, the literature recommends extended education for professionals working with smart city systems. Education should take the form of consistent and frequent training in cybersecurity awareness. The overall proficiency on topics such as adopting stronger passwords, routinely updating software, encrypting files, and avoiding phishing attacks is essential.

Since so many cities use similar network structures and Programmable Logic Controllers (PLCs), this work will address a few of these vulnerabilities [28], specifically in the transportation sector. For this research we employ our educational test-bed to demonstrate these vulnerabilities, as well as strategies on how to eliminate them. By accomplishing this research, it prepares the way for further research to be done with respect to scalable solutions on larger systems, like those in Santander and Antwerp. This research is intended to open the door for real-world implementation of solutions from the test-bed examples into actual smart city networks that are similar to the test-bed at the University of Nevada, Reno.

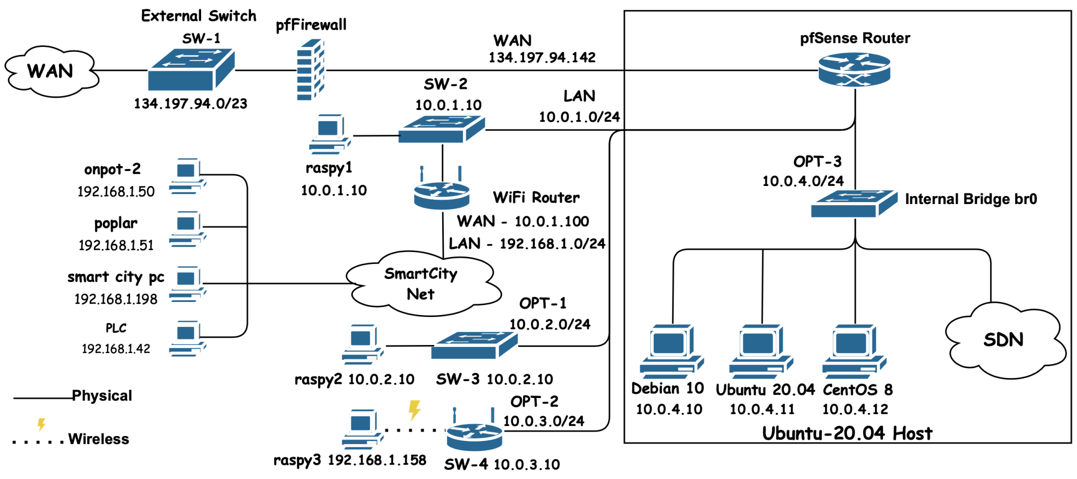


FIGURE 4.1: Network Physical/Virtual Topology

## 4.3 Methodology

This work utilizes the University of Nevada, Reno smart city test-bed which implements both physical and virtual devices on various platforms. The test-bed includes several open-source tools such as OpenvSwitch, KVM/QEMU, Virt-Manager, Linux Bridge-utils, and several versions of the Linux operating system such as Debian, Ubuntu, CentOS, and Rasbian as seen in Fig. 4.1. A virtual pfSense router is incorporated as a network gateway and firewall, and an OpenDayLight SDN controller using OpenFlow10 manages the software-defined network [86]. A *SmartCity* model utilizing a *DirectLogic* PLC that is attached to the network (Fig. 4.2). These methods will be brokend down into four different segments: finding the vulnerabilities on the perimeter of the network, finding vulnerabilities inside the network, finding solutions for the perimeter vulnerabilities, and finding solutions for vulnerabilities that exist inside the network.

### 4.3.1 Vulnerabilities on the Perimeter

To find vulnerabilities on the perimeter of the smart city network, nmap is used to create a map of the network [87]. Nmap provides for two types of low-level scans, and also a much more comprehensive scan. To do this scans are initiated from a computer located on a different subnet than that of the smart city. For this research, scans were run from the Debian-10 machine (4.2). The low-level scans provide for a rough overview of the network, including finding where the smart city subnet was in relation to the rest of the network. The purpose of the low-level scan was to reveal how many hosts or computers were on each subnet and where the routers were located. An example of the command sent to conduct these low-level scans was *sudo nmap 192.168.x.x/24.* The more thorough nmap scans were used later in the research process, after a rough map of the network was established. Thorough scans were not initially used due to the length of time it takes for these scans to run. However, once a rough map of the smart city subnet was established, more thorough scan commands to extrapolate which ports were open on each machine were used. The thorough command accomplished this by scanning each of the 65,535 TCP Ports. The port searched for specifically was port 503, the MODBUS port. An example of the more thorough command was *sudo nmap -p- -sV 192.168.x.x/24* [87].

### 4.3.2 Vulnerabilities Inside the Network

To identify vulnerabilities inside the network, a Linux Command Line Utility called *mbpoll* was used to communicate with the MODBUS port [88]. MODBUS is a data communications protocol that is used to exchange information with PLCs. MODBUS has become a standard communication protocol and is now a common way of connecting industrial electronic devices, including the *DirectLogic205* PLC used in the smart city [28]. Once access is gained to this port, one could theoretically control and manipulate any device that is controlled through the PLC. The only requirement is that the device attacking and manipulating this port must be connected to the Smart City network. In our case,the device could be established through a wired or wireless connection. To exploit the port a simple Dell laptop running Ubuntu to the city's network is connected, and then commands are sent directly to the PLC over the network.

### 4.3.3 Solutions for Perimeter Vulnerabilities

To remedy the perimeter vulnerabilities, a variety of potential options were tested before a method was discovered that worked properly. The intricacies of the testing conducted will be detailed in the results section and will only cover the successful methods in this section. To catch intruders trying to conduct network scans using software like Nmap, a laptop wired directly into the smart city router was configured to run the Intrusion Detection System (IDS) called Snort [87] [89]. The Snort IDS was configured to capture TCP intrusions on the smart city's subnet. To do this the following command was run from the Linux command line: *sudo gedit /etc/snort/snort.conf* [89]. This command opens the Snort configuration window.

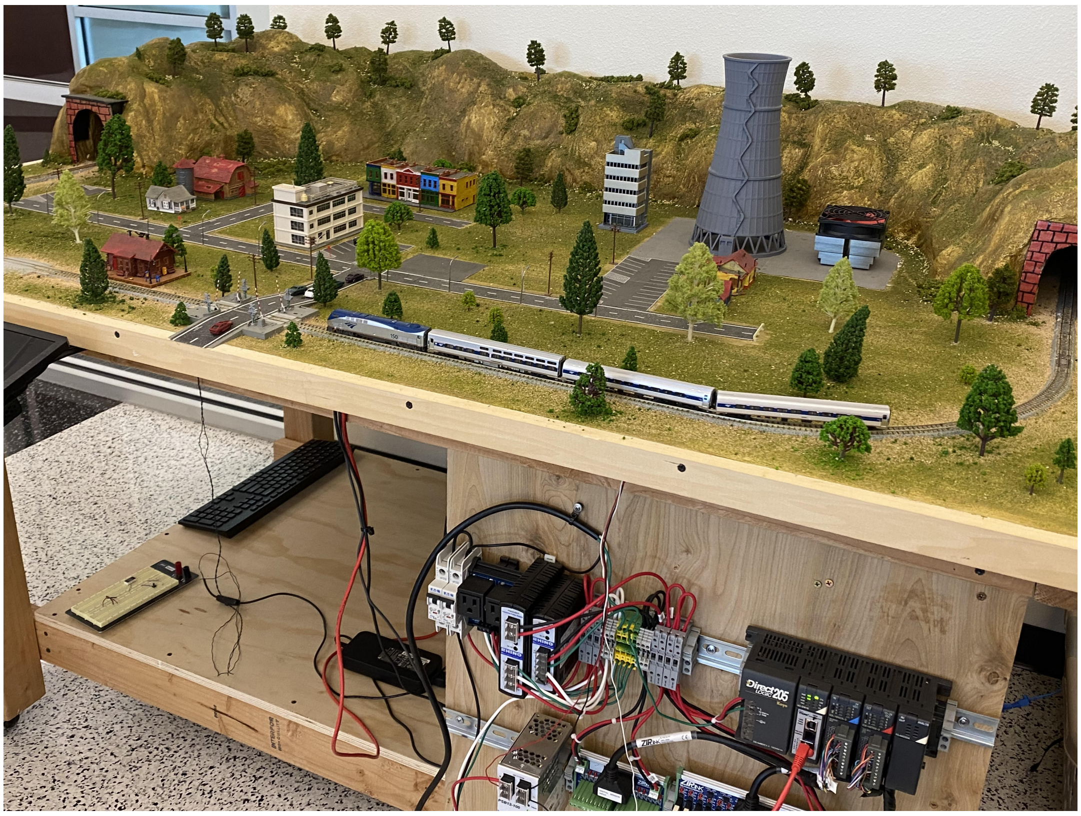
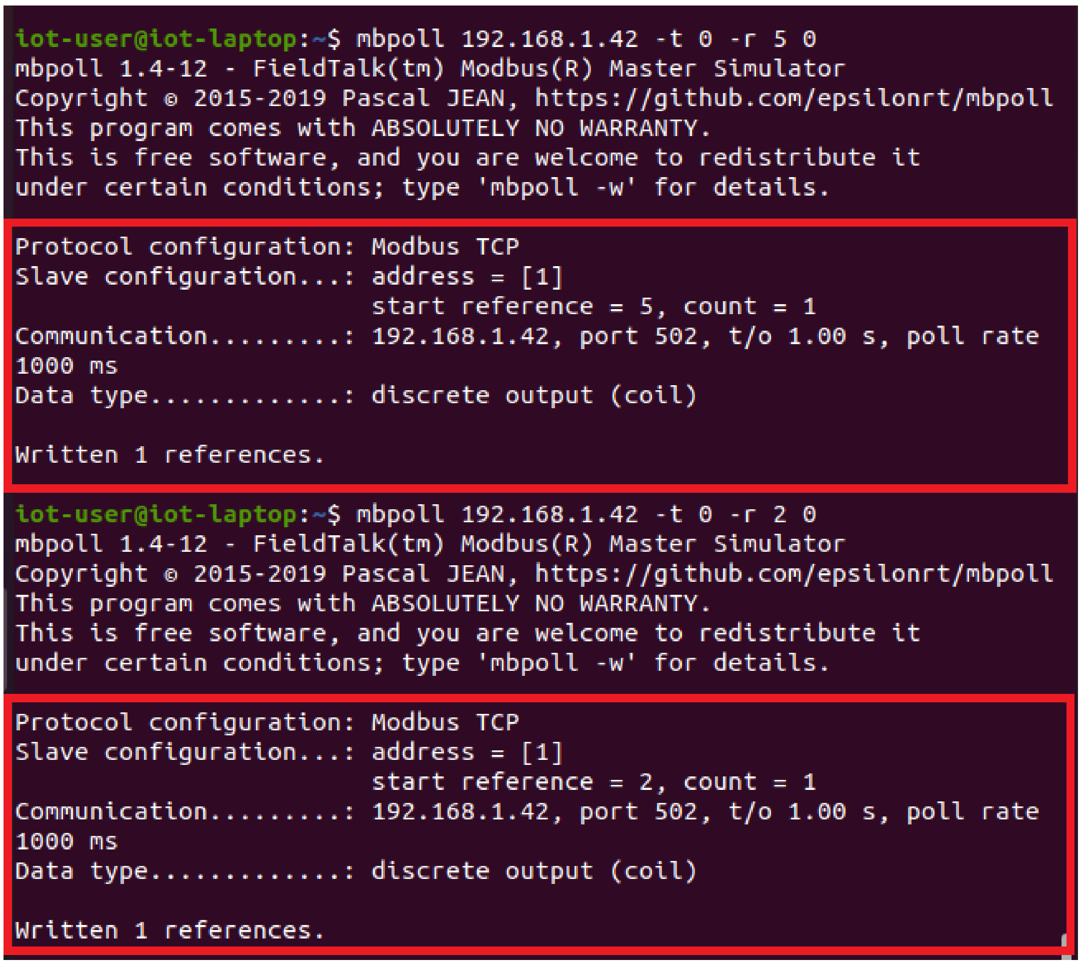


FIGURE 4.2: University of Nevada, Reno Educational Smart City

In the Snort configuration window, the network interface name and the IP range required for monitoring with Snort were set. In our case, the network interface name was *enp4s0* and the IP range of the smart city subnet was *192.168.1.0/24*. While Snort has a set of pre-configured rules to catch most network intrusions, local rules were added to assure that all Nmap scans were captured so they would not pass through as normal network traffic. To do this, the Snort local rules were modified using the command *sudo gedit/etc/snort/rules/local.rules* [89]. Three rules were addedfollowing the standard Snort rule configuration of *alert tcp any any -¿ any (msg:"TCP Scan"; sid:1000001; rev:1;)*. Following this rule syntax, modifications can be made to the rule action, source IP, source port, direction, destination IP, destination port, Snort message, Snort rule ID, revision number, and class type. Once the Snort IDS rules were configured, it was necessary to ensure the system would catch all traffic directed at the PLC. To confirm this, the preinstalled operating system was removed from our Linksys AC1900 V2 router (the router for the smart city network). The open-source router operating system OpenWRT was installed in its place. OpenWRT allows for port mirroring to be configured in the interfaces window when accessing 192.168.1.1 in any web browser. This allows the Enable Mirroring of Incoming and Outgoing setting in the Network tab and in the Switch options. The router was then configured such that the Snort machine would be the receiving machine and the PLC would be the monitored machine. Finally, the wireless functionality on the smart city router was disabled so that the only way to access the network is through a direct, wired access to the router.

### 4.3.4 Solutions for Inside the Network Vulnerabilities

Unfortunately, as will covered in more detail in the results section, there was no concrete solution to the vulnerabilities discovered inside the network. Instead, temporary methods were put in place until a better solution could be found. The first action performed was turning off the wireless functionality on the smart city router so that the only way to access the network is through a direct, wired access to the router. In the case that the smart city is unable to be removed from wireless access, it is recommended that a very complex router password that only trusted individuals know be utilized. Another security method used for inside the network vulnerabilities was configuring a password on the PLC configuration software. For the *DirectLogic* PLC a password was configured for the *Do-More Designer* software that is used to program the PLC. It is recommended this password follow the same requirements as those for the router password, both adhering to recommended best practices.



```
iot-user@iot-laptop:~$ mbpoll 192.168.1.42 -t 0 -r 5 0
mbpoll 1.4-12 - FieldTalk(tm) Modbus(R) Master Simulator
Copyright © 2015-2019 Pascal JEAN, https://github.com/epsilonrt/mbpoll
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'mbpoll -w' for details.

Protocol configuration: Modbus TCP
Slave configuration...: address = [1]
                        start reference = 5, count = 1
Communication.........: 192.168.1.42, port 502, t/o 1.00 s, poll rate
1000 ms
Data type.............: discrete output (coil)

Written 1 references.
iot-user@iot-laptop:~$ mbpoll 192.168.1.42 -t 0 -r 2 0
mbpoll 1.4-12 - FieldTalk(tm) Modbus(R) Master Simulator
Copyright © 2015-2019 Pascal JEAN, https://github.com/epsilonrt/mbpoll
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'mbpoll -w' for details.

Protocol configuration: Modbus TCP
Slave configuration...: address = [1]
                        start reference = 2, count = 1
Communication.........: 192.168.1.42, port 502, t/o 1.00 s, poll rate
1000 ms
Data type.............: discrete output (coil)

Written 1 references.
```

Figure 4.3: Example of exploiting the city's PLC through the Modbus port on the Linux command line

## 4.4 Results

The results section will be divided into the same format as the methods section. To determine the method used to find the results presented, please refer to the corresponding methods subsection.

### 4.4.1 Results for Perimeter Vulnerabilities

Using the low-level nmap scans as described in the methods section, where the smart city subnet was in comparison to the rest of the network was established [87]. While the network topology was already known, nmap scans were conducted as though it were unknown. The actual network map to verify our findings were use as a guide for future actions. Low-level scans revealed all the hosts on the smart city allowing the mapping of the subnet where more thorough scans should be run. When conducting the scan on the network, the IP address of 192.168.1.42 was the only host machine that had port 503, the MODBUS port [28]. Using the more thorough scans it was determined that the PLC was at the IP address 192.168.1.42. The map of the network created was derived from the nmap scans and matched exactly that of the network blueprint itself.

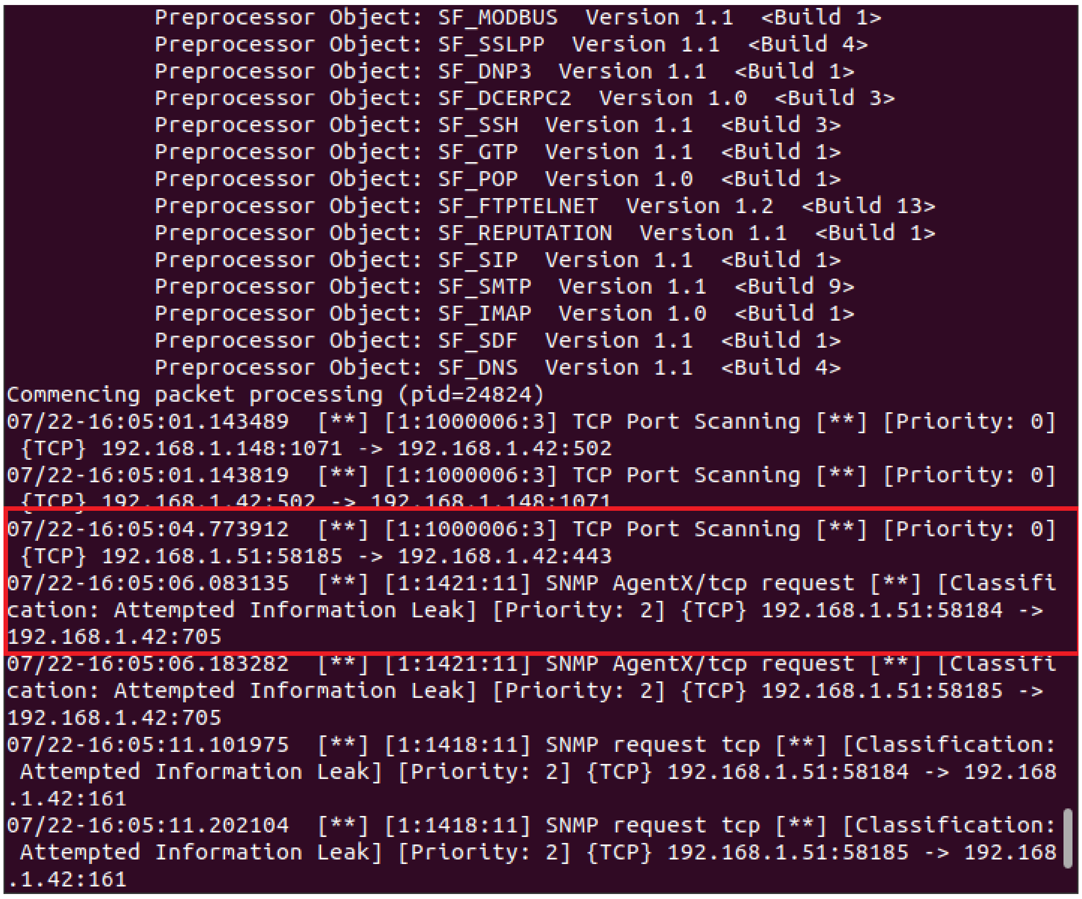
### 4.4.2 Results for Inside the Network Vulnerabilities

Once connected to the Smart City router, it was possible to send the mbpoll command to the MODBUS port on the PLC and control almost the entire city [88]. This was possible both using a laptop that was connected wirelessly and through a RaspberryPi machine that was connected using a Cat5e Ethernet cable to the smart city's router. The mbpoll command was used to read MODBUS coils, write to MODBUS coils, read MODBUS registers, and write to MODBUS registers. The command syntax for reading/writing to MODBUS coils was *mbpoll host -t 0 -r coil* *number 0/1* [88]. The command syntax for reading/writing to MODBUS registers was *mbpoll host -t 4 -r register number 0/1/other* [88]. The easiest way to tell the difference between what MODBUS coils do and what MODBUS registers do is to think of them in binary form. For MODBUS coils, the only change required is to send either 0 or 1, or turning a system off or on. However, for MODBUS registers, these are values that can be overwritten to something other than 0 or 1. In the smart city, the MODBUS coil examples were turning off or on the main power to the city, the train power, the streetlight power, the traffic signal power, or turning off/on each traffic signal light (turning on specifically the red light on the signal light). For the MODBUS register examples, a smaller number of more complicated actions were possible, like changing the direction of the train to off, forward, or backward using the values 0/1/2 respectively. Or, one could change the target temperature of the power generation reactor to an integer, which could cause the nuclear reactor to trigger a false overheat. It was also possible to change the temperature limit to an integer. If the temperature went beyond this limit, the main power would turn off. Finally, it was possible to change the traffic signal mode, so that they either acted like normal signal lights, stop signs, or our custom configuration, like having all the signal lights illuminate the green light at the same time. All of these vulnerabilities, if manipulated in an actual smart city by attacking the PLC, could have potentially life-threatening consequences.

### 4.4.3 Results for Perimeter Solutions

Finding the solutions for the vulnerabilities discovered on the perimeter security proved to be the most challenging task. The main issues encountered were in the technology itself; these will be covered in this section. First, the results for the solution that was presented in the method section that was found to be the most effective will be shared. Using the OpenWRT port mirroring feature, the Snort machine was able to capture almost all potentially manipulative network scans. To confirm the network monitoring abilities of the OpenWRT port mirroring function, the WireShark software was used to monitor the actual network traffic compared to what Snort was catching [90]. When thorough Nmap scans were directed toward the PLC or the entire Smart City subnet, all the traffic on WireShark and most of the nmap TCP pings were detected by Snort [87] [89] [90]. It was then possible to block the IP address of the would-be manipulator and eliminate the threat. This type of detection does require monitoring of the Snort machine on a fairly regular basis. In the context of a smart city, it is recommended that a trusted IT professional is monitoring the city's network traffic.

The issues encountered in implementing perimeter vulnerabilities came from the Linksys AC1900 V2 router used for the smart city's network. It was originally planned to set up a Snort Machine in a Demilitarized Zone (DMZ). The thought here was that the DMZ would enable access to the Snort machine from an external, untrusted network while securing the rest of the network behind a firewall [91]. However, it was determined that when configuring a two-firewall DMZ, it would become unnecessarily complicated; it was originally planned that the wireless functionality of the router would be disabled to eliminate other vulnerabilities from the research. Instead, the Snort machine was configured such that if any nmap scanning traffic was directed towards any device on the network, it would be captured [91]. Unfortunately, because a router and not a switch with port mirroring configured was used, the only nmap scans that the Snort machine would catch were scans directed at the Snort machine's IP, defeating the purpose of catching traffic that was trying to scan the PLC. So, to solve this issue the Linksys preinstalled router operating system was removed and an OpenWRT operating system was installed in its place. The new router operating system was chosen since it has a port mirroring feature, while it still able to function as a router. Once the router operating system was changed to OpenWRT, three unique Snort rules were created to detect different attempted scans of the PLC.



```
          Preprocessor Object: SF_MODBUS  Version 1.1  <Build 1>
          Preprocessor Object: SF_SSLPP  Version 1.1  <Build 4>
          Preprocessor Object: SF_DNP3  Version 1.1  <Build 1>
          Preprocessor Object: SF_DCERPC2  Version 1.0  <Build 3>
          Preprocessor Object: SF_SSH  Version 1.1  <Build 3>
          Preprocessor Object: SF_GTP  Version 1.1  <Build 1>
          Preprocessor Object: SF_POP  Version 1.0  <Build 1>
          Preprocessor Object: SF_FTPTELNET  Version 1.2  <Build 13>
          Preprocessor Object: SF_REPUTATION  Version 1.1  <Build 1>
          Preprocessor Object: SF_SIP  Version 1.1  <Build 1>
          Preprocessor Object: SF_SMTP  Version 1.1  <Build 9>
          Preprocessor Object: SF_IMAP  Version 1.0  <Build 1>
          Preprocessor Object: SF_SDF  Version 1.1  <Build 1>
          Preprocessor Object: SF_DNS  Version 1.1  <Build 4>
Commencing packet processing (pid=24824)
07/22-16:05:01.143489  [**] [1:1000006:3] TCP Port Scanning [**] [Priority: 0]
 {TCP} 192.168.1.148:1071 -> 192.168.1.42:502
07/22-16:05:01.143819  [**] [1:1000006:3] TCP Port Scanning [**] [Priority: 0]
 {TCP} 192.168.1.42:502 -> 192.168.1.148:1071
07/22-16:05:04.773912  [**] [1:1000006:3] TCP Port Scanning [**] [Priority: 0]
 {TCP} 192.168.1.51:58185 -> 192.168.1.42:443
07/22-16:05:06.083135  [**] [1:1421:11] SNMP AgentX/tcp request [**] [Classifi
cation: Attempted Information Leak] [Priority: 2] {TCP} 192.168.1.51:58184 ->
192.168.1.42:705
07/22-16:05:06.183282  [**] [1:1421:11] SNMP AgentX/tcp request [**] [Classifi
cation: Attempted Information Leak] [Priority: 2] {TCP} 192.168.1.51:58185 ->
192.168.1.42:705
07/22-16:05:11.101975  [**] [1:1418:11] SNMP request tcp [**] [Classification:
 Attempted Information Leak] [Priority: 2] {TCP} 192.168.1.51:58184 -> 192.168
.1.42:161
07/22-16:05:11.202104  [**] [1:1418:11] SNMP request tcp [**] [Classification:
 Attempted Information Leak] [Priority: 2] {TCP} 192.168.1.51:58185 -> 192.168
.1.42:161
```

FIGURE 4.4: Example of catching an intruder (192.168.1.51) scanning the PLC (192.168.1.42) using OpenWRT Port Mirroring and and Snort IDS

### 4.4.4 Results for Inside the Network Solutions

Inside the network, the results that we had originally hypothesized were not possible. Using the *DirectLogic 205* PLC, there is a lack of available security measures, because, like many other PLCs, it was deployed with outdated SCADA systems. These SCADA systems' purpose is to monitor and control devices, like the smart city IoT devices, at remote sites. SCADA systems are necessary because they help maintain efficiency by collecting and processing real-time data [28]. They also allow for real-time manipulation and adjustments so that systems can stay online while updates are sent out to devices. Despite these benefits, they lack security features. The United States Department of Energy has even acknowledged their weakness, stating that performance, reliability, flexibility and safety of distributed control/SCADA systems are robust, while the security of these systems is often weak [92]. There is even a specific advisory published about the *DirectLogic* PLCs [93]. In this specific smart city setup, using the corresponding methods section about Inside the Network solutions was the most secure the PLC could be made. Some level of encryption or even a machine learning Intrusion Protection System (IPS) was desired, but options were limited. It was found that turning off the wireless functionality for the router that the PLC is connected to was a good temporary solution and allowed for strict monitoring of who was on the network. It was also found that configuring a password for the *Do-More Designer* software that is used to program the PLC allowed for at least some added security, but the passwords themselves are very basic. The *Do-More Designer* only allows for eight-character numeric passwords, and most of the time the password is preset to all zeros.

## 4.5 Future Solutions

When comparing the internal vulnerabilities exploited by the mbpoll command to the internal vulnerabilities solutions, it is apparent that the solutions were minimal. As examined before, the best options for securing these vulnerabilities were to isolate the smart city network or employing strong password etiquette. To isolate the smart city network, the network administrators must turn off or hide the current wireless functionality of the entire smart city. Additionally, network administrators should ensure that any password associated with the PLC software abides by secure standards. Future research can be conducted to further discover and test new solutions for the internal vulnerabilities. The security implications of interconnected smart cities also can be taken into consideration for future research.

### 4.5.1 OpenPLC

In the search for internal vulnerability solutions, an open source software solution called OpenPLC was found which is an alternate to the legacy components of the SCADA software based systems. Traditional PLC hardware architectures have reserved their documentation which makes it difficult for researchers and educators to completely explore the existing vulnerabilities and test developing solutions to these exploits. Contrary to the traditional PLC hardware architectures, the open source capabilities of OpenPLC would allow researchers to assess network vulnerabilities and test solutions with a hands-on methodology at the hardware level. After extensive additional research, this software could replace the security shortcomings of the outdated SCADA software. The OpenPLC project was created specifically for this purpose [94].

A key functionality embodied in the OpenPLC software is its aptitude for cryptography. The most traditional sense of security stems from well-developed and unique cryptography in place for static and dynamically operating networks [1]. OpenPLC adheres to a AES-256 implementation encryption process. This encryption technique requires that both the sender and receiver of data must have the same secret key in order to gain access to the information, which creates what is known as a symmetric cipher. Because both the sender and receiver of data are required to be able to decrypt information in the same way, there is an additional step needed to allow the PLC system to benefit from the OpenPLC. The additional step requires the OpenPLC project to implement a localhost Gateway to allow the supervisory software of the PLC to be able to decrypt the data encrypted by OpenPLC. By enabling the OpenPLC localhost gateway and further designating the PLC IP address within the supervisory system as *localhost*, the supposed unsecured channel between the designated gateway and the main components of the PLC is nonexistent [1], thus protecting the system.

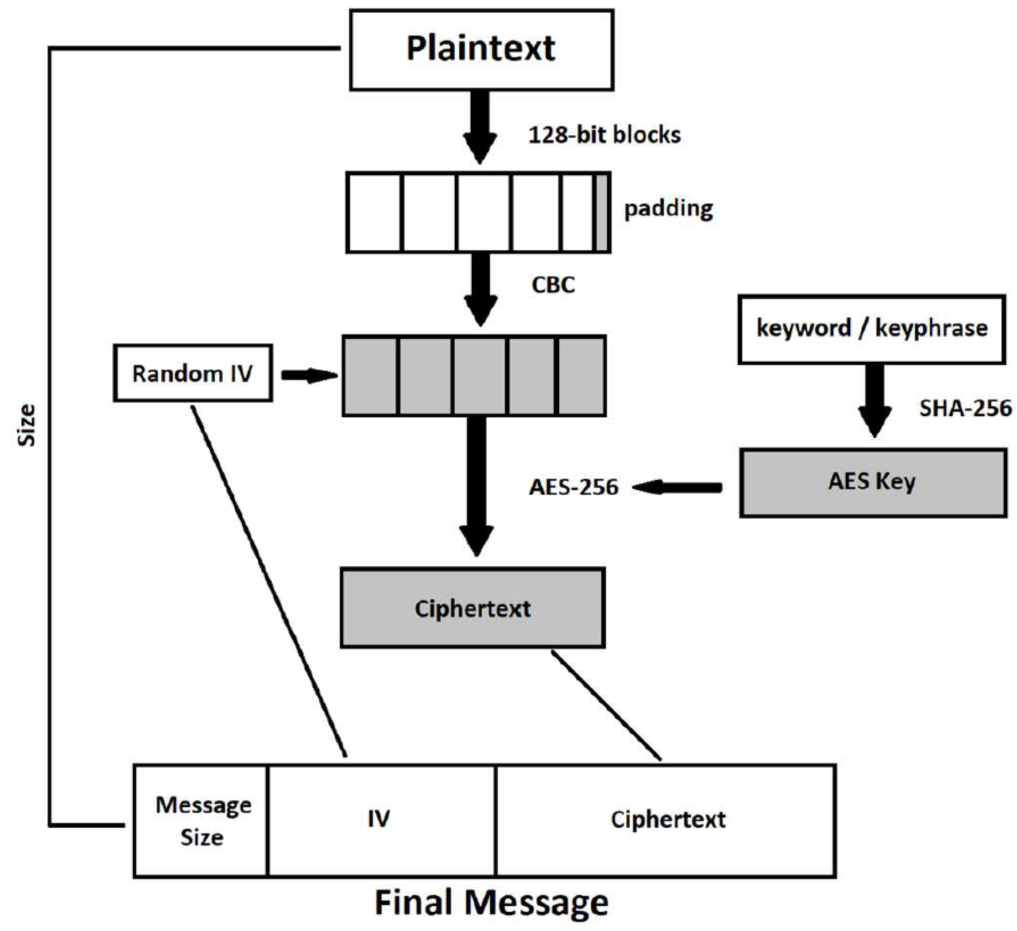


FIGURE 4.5: OpenPLC Neo Encryption Process [1]

## 4.5.2 Interconnected Smart Cities

Another challenge created by the emergence of smart cities is the evaluation of security threats within the scope of multiple smart cities being interconnected. The extensive inter-connectivity of smart cities exponentially increases the systems' endpoint complexity. In order to be completely secure each additional device to the network has to be operated to the same standard of security as all other devices on the network as the level of security is only guaranteed by the weakest link [29]. Another consideration for the system of systems approach for interconnected smart cities, is the increased complexity of maintaining the vast quantity of devices. This allows for a single-points of failure in the event of routine program bugs or human mistakes that would have a cascade effect [29] on the entire system. The disastrous consequence of this possibility is that the entire system would be wiped out rather than a single segment of the system. While smart cities are a promising advance in many ways, it is apparent there are significant security concerns to be addressed. This research is continued with the development of a software defined IoT testbed, wherein advanced networking techniques can be explored to better secure connected systems in the future.

# Chapter 5

# A Software Defined Networking Testbed for IoT Research and Education

## 5.1 Introduction

Around the turn of the 21st century, the idea of an *Internet of Things* (IoT) began to emerge as a concept; a vision of billions of devices such as low-powered sensors, cameras, watches, household devices, cars and even airplanes all connected simultaneously to the Internet and able to communicate and share data with one another. Since that early vision IoT has continued to expand and is considered, by some, to be the next industrial revolution. Today IoT devices are everywhere, effectively surrounding us in every aspect of our lives. It has become apparent that this trend is here to stay, with reports indicating that there are as many as 22 billion IoT devices operating worldwide, and we can expect as many as 38.6 billion by the end of 2025 and over 50 billion by 2030 [95]. As pervasive as this technology has become, it is still under development and is changing the way we view many aspects of technology, including networking and communications, data processing and sharing, and power consumption.

One of the areas IoT is finding wide acceptance is in household devices. The concept of a *smart home* has become commonplace, seamlessly integrating multiple household systems. Features such as temperature controls, security, access controls, lighting, entertainment, and appliances are network connected and can be remotely managed. These heterogeneous systems are linked to their respective control centers using wireless technologies, but still utilize a largely disjoint architecture. This means each set of systems utilize their own connections to control and data storage servers rather than working together. Future growth will require more *horizontal* coordination, with systems sharing control plane resources to improve resource efficiency and facilitate data sharing.

Other areas of growth include the Industrial Internet of Things (IIoT) as manufacturing, healthcare, and military applications are becoming more distributed. In addition, the concept of *Smart Cities* has gained traction, with increased coordination in public safety systems, transportation, and traffic controls in an effort to accommodate a growing population. A *smart power grid* has also been largely implemented, synchronizing power production and utilization for more effective delivery. The growth of these technologies raise many questions about IoT devices, their implementation, networking, data storage, control, and system security. In addition, rapidly evolving multi-faceted technologies make it difficult for students and new researchers to gain necessary experience and engage in research and development.

Many of today's IoT systems rely on specialized platforms and application domains, and could greatly benefit from a more integrated architecture. To address these concerns, IoT testbeds must be built to allow hands-on experience and the development of expertise in fields such as programming, networking, circuits, and micro-controllers [33]. Although there exists IoT testbeds to address some of these requirements, they are often very expensive to build and aim to address a subset of desired functionalities, demanding low-cost environments that provide multi-use capabilities combining research, development, testing, and education in a single system are still needed. This work aims to address this critical gap by developing a low-cost, flexible, and realistic IoT testbed that will significantly lower the learning curve for students and new researchers to participate in IoT research.

## 5.2 Contributions

While IoT technologies are on the rise, there is still a wide gap between state-of-the-art technologies and research/training environments accessible for many universities. Network test-beds can be expensive to build, limited in their flexibility, and difficult to set up. In addition, many of the proposed test-beds in the literature address limited areas of focus; for instance, device testing, network research, or education and training. In addition, none of the previous works address IoT, virtual and physical networking, SDN, and security in a single, inexpensive format that can be used for research, testing, and education simultaneously. It is our aim to provide a complex and multi-level network test-bed environment for IoT wherein students and researchers can interact with both virtual and physical devices in a realistic and easily re-configurable setting. On one hand, network and traffic optimization can be performed on a hybrid virtual/physical software defined network, while concurrently generating and collecting traffic for research in anomaly detection utilizing machine learning techniques. The system incorporates both physical and virtual IoT and IIoT devices to provide real-world data, and also hosts a number of honeypot devices. This provides the ability to conduct both research on the development of honeypot stealth techniques, and also allows for attack and defense training. Students are able to interact with and probe the system through access to dedicated gateway devices located throughout the network. The network provides interaction with both physical and virtual devices; some actual targets and some honeypots. Students are able to practice security techniques (i.e. scanning, fingerprinting, etc.) across the network, while research on aspects such as honeypot design, IoT communications, and development of SDN networks (i.e. controller design and traffic routing) can be conducted simultaneously.

## 5.3 System Model

To emulate a realistic IoT environment that can simultaneously address research and development, education, and security, a complex environment incorporating multiple network elements is developed. The network contains multiple levels, is re-configurable, and addresses Ethernet, wireless, and software-defined networking architectures. The testbed utilizes both physical and virtual devices on various platforms, and includes a number of open-source tools such as OpenvSwitch,

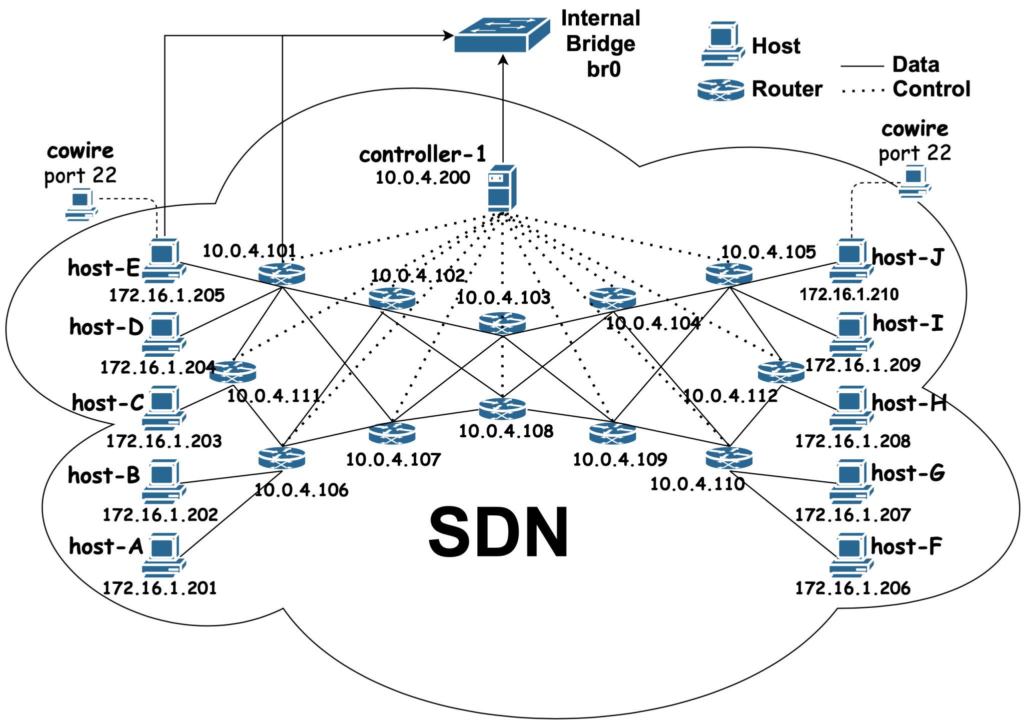


Figure 5.1: Topology of the virtual Software Defined Network.

KVM/QEMU, Virt-Manager, Linux Bridge-utils, and several versions of the Linux operating system such as Debian, Ubuntu, CentOS, and Rasbian. A virtual pf-Sense router is incorporated as a network gateway and firewall, and an OpenDay-Light SDN controller using OpenFlow10 manages the software defined network. A *Smart City* model utilizing a Direct Logic programmable logic controller (PLC) is attached the network, allowing students and researchers to interact with mechanical elements via the modbus protocol.

A 24-port managed switch connects the test-bed to a wide area network, which is in turn connected to a server with 5 external Ethernet ports. This host machine supports the virtual router, and utilizes a 4-port Ethernet card with the additional ports passed directly to the router; one for the WAN and three sub nets. In addition, a fourth sub net is connected to a virtual bridge in the host. Within the firewall a *gateway* device is located in each sub net, allowing for internal network access by students to probe the network and gain experience with foot-printing techniques. This also provides students an opportunity to monitor traffic via honeypots, which are distributed throughout the network.

The three physical ports are connected to external network elements through managed switches and wireless routers, while the internal bridge is connected to numerous virtual machines. The internal bridge supports the virtual SDN. A diagram of the SDN can be seen in Figure 5.1. The physical network connects several Raspberry Pi hosts with various operating systems which emulate desktop computers, IoT devices, and honeypots. In addition, a *Smart City* model with a Programmable Logic Controller (PLC) is connected on its own sub-net, see Figure 4.2. A diagram of the network topology can be seen in Figure 4.1.

### 5.3.1 Smart City Controller

To emulate realistic industrial controls, the *Smart City* component of the test-bed relies on a Direct Logic PLC which communicates over the network using the *modbus* protocol. The PLC is made up of four modules; networking, digital input, digital output, and an analog I/O module. To offer a visual/tangible aspect the PLC controls an electric train, crossing guards, street lighting, realistic traffic signals, and a simulated nuclear power plant. The power plant has a heating unit and temperature set-points managed by a command and control server, and utilizes a cooling fan and smoke generator. All functions are accessible through the network, giving students an opportunity to view functionality remotely, and to test their skills at both hacking and defending the infrastructure by sending modbus commands directly to the smart city and altering its behavior. A graphic user interface and programming tools are hosted on the command and control server. The sub net hosting the Smart City also contains decoy honeypots (Conpot) emulating similar protocols for both masking the city, and for collecting attack data from the network.

### 5.3.2 Honeypot Devices

One of the goals of the IoT network is to facilitate research and education in cybersecurity. For this reason numerous honeypots of varying types are run in the network to act as decoy devices, and to collect data on attacker activity. Students are given access to both the honeypots and the gateway device in the network, and can develop their skills both at scanning and attempting to gain access to devices, as well as learning to monitor the activities of potential attackers. Devices are hosted both virtually on physical and virtual devices in the network, and also as stand-alone devices hosted on Raspberry Pis. Cowrie, Conpot, Dionaea, and custom honeypots are operated to gather SSH attempts, FTP logins, traffic tunneling attempts, attacker behavior, etc. The goal is to teach students both red-team and blue-team skills and to expose them to various security technologies, as well as to conduct active research on honeypot fingerprinting, device masking, and attack analysis.

### 5.3.3 Software Defined Networks

Traditional networks are typically composed of autonomous fixed-function network devices. These devices have hard-wired functionality that doesn't provide enough flexibility for satisfying the requirements of modern networks [96]. These conventional networks have both control and data plane built into the same device. Software Defined Networking (SDN) on the other hand takes a different approach wherein the control and the data planes are decoupled. Networking devices such as switches and routers implement the data plane which is controlled by the centralized control plane software typically known as a controller. Hence, the software based control plane makes decisions on how the network packets are forwarded and the networking devices execute the policy set by the control plane [97]. This approach improves network management, scalability, programmability, agility, and overall performance of modern SDN networks.

With flexibility provided by a SDN, the collection of network information is greatly simplified. Having a central view of the network helps better understand network status and activities. This information can be used to improve the algorithms designed to detect attacks [98]. With agility and fine-grained control provided by the SDN, responding to detected attacks becomes a simpler task. For example, if a botnet command-and-control (C&C) communication is discovered by the detection algorithm, the control plane can install policies in the data plane that can drop packets related to that specific C&C, effectively terminating the existing communication and eliminating the threat. Hence, SDN can provide an active security layer in the network that was not easily realizable in the traditional networking paradigms.

IoT devices are much more vulnerable than traditional compute devices. The Mirai botnet, which was responsible for the largest distributed denial of service (DDoS) attack recorded, was able to amass a large number of IoT devices such as IP cameras, routers, printers, DVRs etc. [9]. This demonstrates the presence of critical vulnerabilities in IoT devices which can be exploited. SDN can be utilized to leverage a global view of the network to understand the behavior of IoT devices and and to leverage this to employ attack detection and prevention mechanisms at the network level. Our proposed IoT test-bed caters to the goal of understanding the behavior of IoT devices and build better solutions to actively detect and prevent security attacks using SDN. Figure 5.1 shows the SDN network currently running in our test-bed which can be customized according to the simulation requirements.

### 5.3.4 Traffic Analysis and Network Probing

The test-bed provides multiple points where traffic monitoring and data collection can be performed. The use of a virtual environment provides for a simplified means to probe the traffic directly in the device or host as required. A device (either virtual or Raspberry Pi-hosted) is located within each network segment to act as a gateway device which students can be given access to. In this way, they are able to gain a foothold within the network firewall to act as a starting point for network foot-printing. Using scanning tools such as *nmap* or *Zmap*, students can probe the test-bed to discover network topology, device types, etc. and to attempt brute-force attacks or vulnerability exploits on the various elements. Simultaneously, honeypot devices are able to record some of this activity, and analysis can be performed on honeypot logs to learn attacker behavior and to provide research and development opportunities on honeypot technologies. Skills and understanding can be gained for both improved device stealth, as well as skills in device fingerprinting. To test the setup, university cyber-club students can be given access to the network from both an offensive and defensive perspective for exercises in attack and network defense.

## 5.4 Simulations

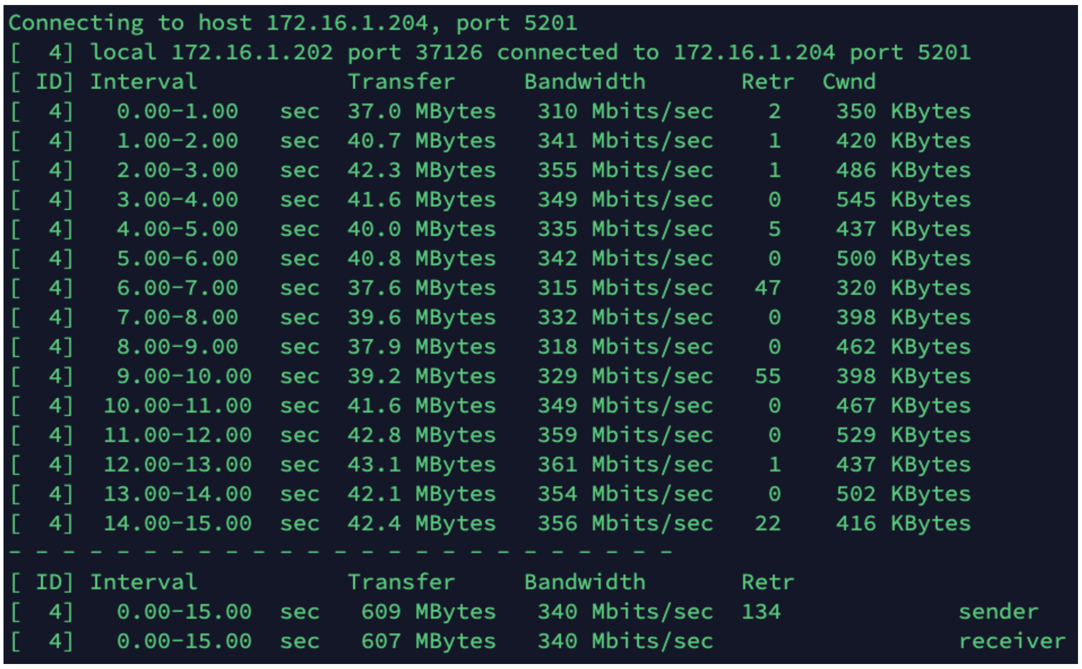
In the following section simulations are performed on the test-bed environment to demonstrate functionality and proof-of-concept. The test-bed is designed to support multiple experiments simultaneously by students and researchers for network analysis, security analysis, and education/training. The following is a detail of such activity.

### 5.4.1 Traffic and Flow Analysis

To showcase the functionality of the developed IoT test-bed, IoT traffic and flow information within the software defined network is captured. There are two case scenarios that are analyzed to highlight the operability of the proposed network:

- When network operations are uninterrupted
- When network operations are interrupted

Interruptions can refer to any network traffic that may influence normally accepted operations. This can include multiple traffics flows through the same network device, causing additional congestion and increased resource sharing. Another instance of interruption can be device failure, malfunction, or tampering which



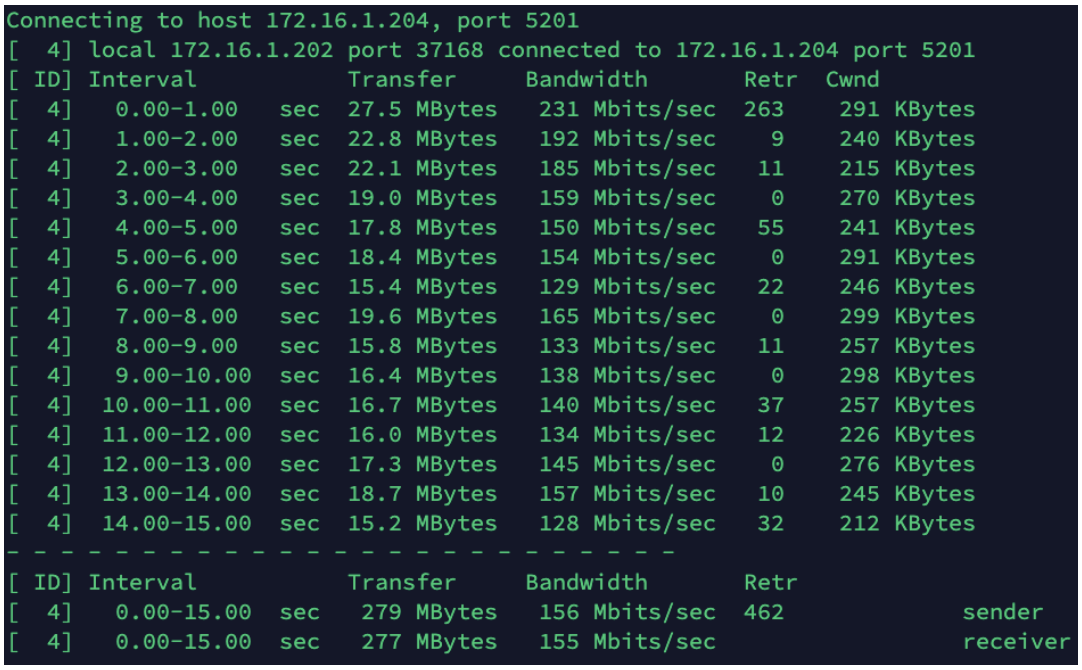
```
Connecting to host 172.16.1.204, port 5201
[  4] local 172.16.1.202 port 37126 connected to 172.16.1.204 port 5201
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  37.0 MBytes  310 Mbits/sec    2    350 KBytes
[  4]   1.00-2.00   sec  40.7 MBytes  341 Mbits/sec    1    420 KBytes
[  4]   2.00-3.00   sec  42.3 MBytes  355 Mbits/sec    1    486 KBytes
[  4]   3.00-4.00   sec  41.6 MBytes  349 Mbits/sec    0    545 KBytes
[  4]   4.00-5.00   sec  40.0 MBytes  335 Mbits/sec    5    437 KBytes
[  4]   5.00-6.00   sec  40.8 MBytes  342 Mbits/sec    0    500 KBytes
[  4]   6.00-7.00   sec  37.6 MBytes  315 Mbits/sec   47    320 KBytes
[  4]   7.00-8.00   sec  39.6 MBytes  332 Mbits/sec    0    398 KBytes
[  4]   8.00-9.00   sec  37.9 MBytes  318 Mbits/sec    0    462 KBytes
[  4]   9.00-10.00  sec  39.2 MBytes  329 Mbits/sec   55    398 KBytes
[  4]  10.00-11.00  sec  41.6 MBytes  349 Mbits/sec    0    467 KBytes
[  4]  11.00-12.00  sec  42.8 MBytes  359 Mbits/sec    0    529 KBytes
[  4]  12.00-13.00  sec  43.1 MBytes  361 Mbits/sec    1    437 KBytes
[  4]  13.00-14.00  sec  42.1 MBytes  354 Mbits/sec    0    502 KBytes
[  4]  14.00-15.00  sec  42.4 MBytes  356 Mbits/sec   22    416 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-15.00  sec   609 MBytes  340 Mbits/sec  134             sender
[  4]   0.00-15.00  sec   607 MBytes  340 Mbits/sec                  receiver
```

FIGURE 5.2: Bandwidth Captured from Host B to Host D during uninterrupted operation

can cause variability in network traffic statistics. For our simulations, a focus is placed on the network traffic between Host-B and Host-D.

To generate network statistics related to uninterrupted flow, network flow information is captured from Host-B to Host-D. During this time there is no other traffic flow occurring between any other hosts in the entirety of the network. Figure 5.2 illustrates the observed uninterrupted network bandwidth, transferred bytes, packet re-tries, and TCP congestion from Host-B to Host-D during a predetermined time interval. It can seen that during uninterrupted network operations, the bandwidth from host-B to host-D remains between 310 - 361 megabits per second. We also observe that the TCP congestion remains consistent with packet loss in the transmission, as TCP congestion window drops whenever there are packet losses.

Along the same lines, network statistics are generated for interrupted flows. To create the interruption, simultaneously transfers are performed between host-A and host-C, while host-B and host-D were also running. Figure 5.3 shows the observed interrupted network bandwidth, transferred bytes, packet re-tries, and



```
Connecting to host 172.16.1.204, port 5201
[  4] local 172.16.1.202 port 37168 connected to 172.16.1.204 port 5201
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  27.5 MBytes   231 Mbits/sec  263    291 KBytes
[  4]   1.00-2.00   sec  22.8 MBytes   192 Mbits/sec    9    240 KBytes
[  4]   2.00-3.00   sec  22.1 MBytes   185 Mbits/sec   11    215 KBytes
[  4]   3.00-4.00   sec  19.0 MBytes   159 Mbits/sec    0    270 KBytes
[  4]   4.00-5.00   sec  17.8 MBytes   150 Mbits/sec   55    241 KBytes
[  4]   5.00-6.00   sec  18.4 MBytes   154 Mbits/sec    0    291 KBytes
[  4]   6.00-7.00   sec  15.4 MBytes   129 Mbits/sec   22    246 KBytes
[  4]   7.00-8.00   sec  19.6 MBytes   165 Mbits/sec    0    299 KBytes
[  4]   8.00-9.00   sec  15.8 MBytes   133 Mbits/sec   11    257 KBytes
[  4]   9.00-10.00  sec  16.4 MBytes   138 Mbits/sec    0    298 KBytes
[  4]  10.00-11.00  sec  16.7 MBytes   140 Mbits/sec   37    257 KBytes
[  4]  11.00-12.00  sec  16.0 MBytes   134 Mbits/sec   12    226 KBytes
[  4]  12.00-13.00  sec  17.3 MBytes   145 Mbits/sec    0    276 KBytes
[  4]  13.00-14.00  sec  18.7 MBytes   157 Mbits/sec   10    245 KBytes
[  4]  14.00-15.00  sec  15.2 MBytes   128 Mbits/sec   32    212 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-15.00  sec   279 MBytes   156 Mbits/sec  462             sender
[  4]   0.00-15.00  sec   277 MBytes   155 Mbits/sec                  receiver
```

FIGURE 5.3: Bandwidth Captured from Host B to Host D during interrupted operation

TCP congestion from host-B to host-D during a certain time interval. It can seen that during interrupted network operations, the bandwidth from host-B to host-D is reduced to between 128 - 231 megabits per second, less than half of uninterrupted network statistics. It can also be observed that TCP congestion remains consistent with the packet loss in the transmission, as TCP congestion window drops whenever there are packet losses.

Two metrics are gathered to measure the network flow performance between the interrupted and uninterrupted flows: throughput and round trip time. To capture the throughput, network traffic is captured and analyzed under the two case scenarios. The results are shown in Figure 5.4. It can be seen that introducing the additional flow reduced the throughput below 50% of that of the uninterrupted flow. In fact, the mean throughput measured for the uninterrupted flow was 306.13 Mb/s, while that of the interrupted flow was 122.46 Mb/s.

Another metric that was gathered was the round trip time for the network packets under interrupted and uninterrupted flows. Both simulations were run for a period of 50 seconds. This interval was chosen as it provided an adequate time for generalization of the network activity under both circumstances. The data gathered is illustrated in Figure 5.5. From this, we noted that there were ≈52,000 network packets transferred during the uninterrupted case, while the packets dropped to ≈23,000 during the interrupted case. This demonstrates that the uninterrupted flow was able to transmit more network packets than the interrupted flow. This is because of the reduced throughput and congestion on the network. Also, it is noted that the magnitude of the round trip times are lower in the uninterrupted circumstance than that of the interrupted. This can also be attributed to the congestion on the network. In fact, the mean round trip time on the uninterrupted flow was measured to be 0.079 ms while that of the interrupted measured 0.127 ms.

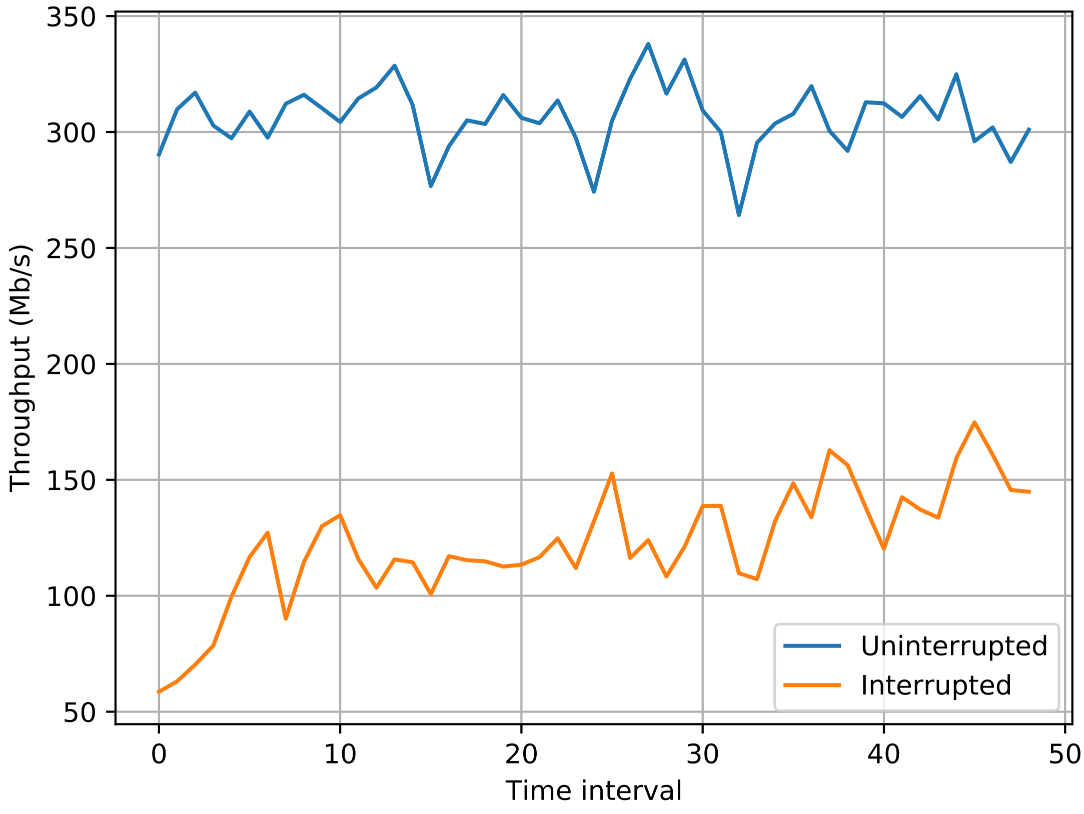
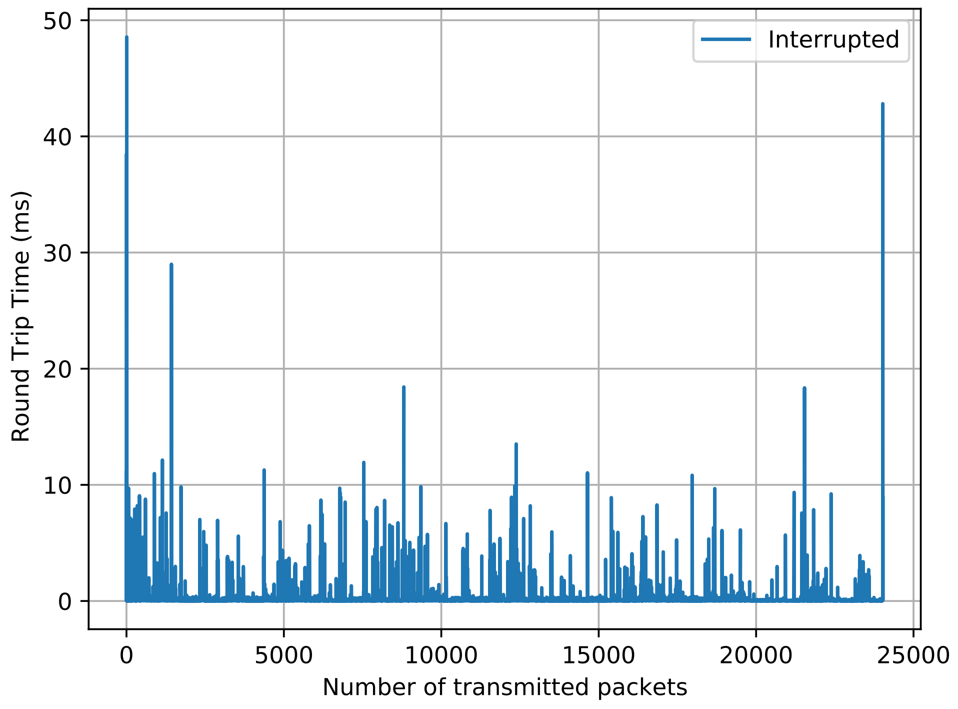
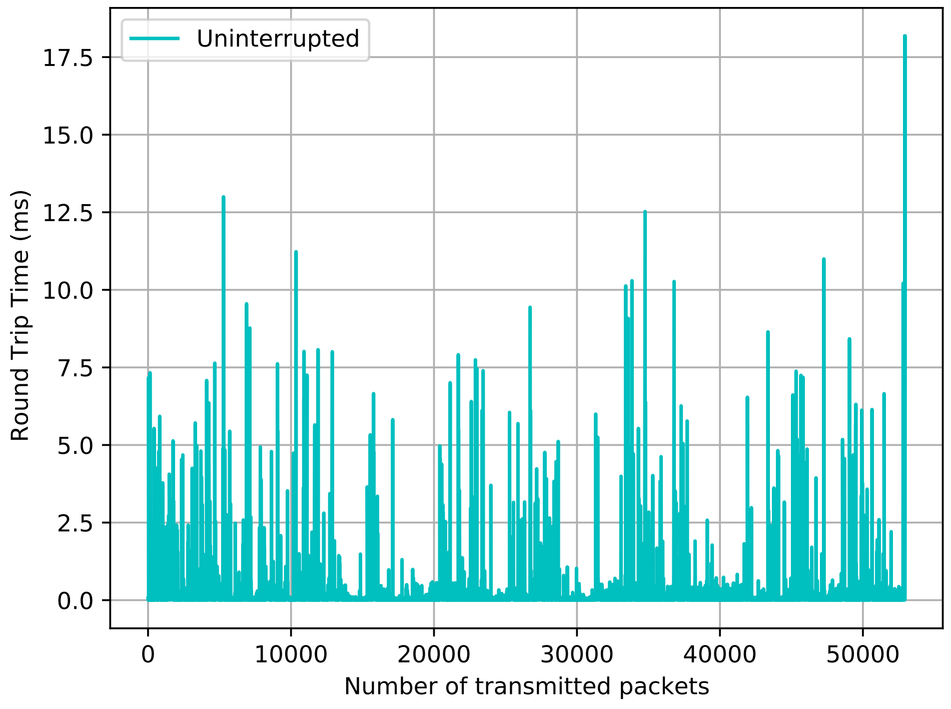


FIGURE 5.4: Measured Throughput between Interrupted and Uninterrupted flows

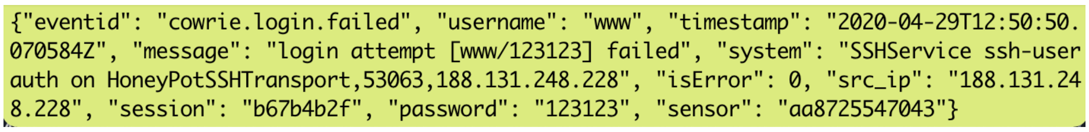


((A)) Interrupted Traffic Flows



((B)) Uninterrupted Traffic Flows

Figure 5.5: Round Trip Times for Network Traffic Flows



```
{"eventid": "cowrie.login.failed", "username": "www", "timestamp": "2020-04-29T12:50:50.
070584Z", "message": "login attempt [www/123123] failed", "system": "SSHService ssh-user
auth on HoneyPotSSHTransport,53063,188.131.248.228", "isError": 0, "src_ip": "188.131.24
8.228", "session": "b67b4b2f", "password": "123123", "sensor": "aa8725547043"}
```

FIGURE 5.6: Cowrie .json log segment showing a failed login attempt on a honeypot.

### 5.4.2 Network Foot-printing and Honeypot Detection

By granting a user (student) access to a gateway machine inside of the test-bed firewall, it is possible to practice techniques for discovering network topology and makeup with the help of scanning tools such as nmap or Zmap and potentially discover vulnerable hosts or services on specific ports. In this case the objective is to identify honeypots that have been placed on the network, namely Cowrie honeypots that will be running SSH and FTP on ports 22 and 23. By running the Zmap command: ***zmap –interface=ens3 -p 22 10.0.4.0/24 –output-file=targets.txt*** or the nmap command: ***nmap -p- 10.0.4.0/24*** a list of hosts can be assembled for further analysis. Honeypots can be fingerprinted in a number of ways by identifying key attributes, in particular those associated with default settings included with the Cowrie installation. One such feature is the key exchange algorithms presented by a host during the SSH handshake. By using the -v flag (for *verbose, or debug* a listing of available algorithms is given to allow hosts to agree on a suitable algorithm. The default algorithms available in Cowrie by default under the *kex-input-ext-info: server-sig-algs* tag are *rsa-sha2-256* and *rsa-sha2-512.* For a non-honeypot SSH installation this list is typically (although not necessarily always) longer, containing several possible key-exchange algorithms. This short list is one possible means of fingerprinting a potential instance of a Cowrie honeypot.

Cowrie can be set to respond to login attempts by reading from a default list of username and password combinations, although in most cases (by default) it is set to accept any combination of username and password after a pre-set random number of attempts. The goal is to give the appearance of security, but ultimately it is desirable to allow an attacker to access the honeypot. Typically an SSH installation will be given a good password, will not allow root logins, and will often block login attempts for a certain amount of time after three failed entries. By brute-forcing suspected instances of Cowrie based on what is returned in the key-exchange stage of the handshake it is possible to successfully log into a honeypot.

Cowrie is capable of logging all activities received, and keeps track of the various attack-command types such as successful and failed login attempts, usernames and passwords for each attempt, file downloads, system commands, etc. By examining these logs it is possible to learn attacker behaviors, and to discover some of the activities an attacker is conducting prior to, during, and after an attack. These logs are stored in .json format and are easily parsed either with user-defined script, or by using tools such as the ELK stack for storing, parsing, and displaying log information. An example of a Cowrie honeypot indicating a failed login attempt can be seen in Figure 5.6. In this example the user *www* attempted to log in with the password *123123* unsuccessfully. This is very useful for helping students to understand the makeup of a cyber attack, and to view real-time attacks while they are in progress. In a honeypot exposed to the Internet these attacks often come in a near-continuous stream from bots (and occasionally live attackers) seeking to compromise exposed servers and IoT devices. By making these tools accessible in a testbed environment, they can be used for instruction and training, and can prepare students for deploying and analyzing honeypots and honeypot data in the real world.

A second honeypot deployed in our testbed is the Conpot honeypot, which emulates a number of IIoT devices on ports supporting device-specific services and protocols. Access to these devices allows researchers and students to develop techniques for fingerprinting devices, and also for learning to mask their presence in an effort to learn as much about an attacker as possible before they determine they are dealing with a honeypot and not a *real* device.

In the following section this research is extended as we move to deploying a series of globally positioned honeypots in six different countries to collect and analyze a large body of malicious traffic data on the Digital Ocean network. This will allow for a greater understanding of IoT botnet activity as attacker activity is examined on a large scale.

# Chapter 6

# Utilizing Global Honeypots for Malicious Traffic Collection

## 6.1 Introduction

Scanning and brute-force attacks on Internet facing services such as SSH, Telnet, FTP, and HTTP have become so common that within minutes or even seconds of connecting devices to the global network, attacks are being launched to compromise them. An unwary administrator might find their work is actively being compromised even as they are in the process of setting it up. Much of this malicious scanning can be attributed to the Mirai Botnet malware and its variants. Shortly after Mirai made headlines in 2016, the code behind the botnet was released as open-source, and has been modified by various hackers seeking to build their own zombie-armies [49]. The original version of Mirai targeted vulnerable Internet of Things (IoT) devices using a short word list based on default username and password combinations, and by leveraging open Telnet and SSH services was able to assemble an army of hundreds of thousands of connected devices using just 60 username/password combinations [12]. The subsequent release of Mirai's source code lead to numerous clones and modifications by other malicious actors which have expanded upon the attacks present in the original version of Mirai. Despite receiving much attention after the 2016 attacks, IoT security is still a major issue and new botnets appear regularly. As a result, there is a continued need to monitor developments in IoT botnets so current attacks can be appropriately dealt with. Evidence of this code and the sequence of actions indicating its installation and use are evident in data collected by SSH and Telnet honeypots and account for a significant portion of scan and attack activity detected, along with a host of other botnet-related malware. One method for analyzing botnet activity is through the use of honeypots for data collection.

Honeypots are a useful tool for capturing such events by providing a realistic environment for attack, and then logging activity for later analysis. They have been extensively employed for such tasks as attack pattern comparisons, attack frequency analysis, attack origin analysis, root cause identification, and risk assessment [12]. For SSH and telnet attacks Cowrie is a popular medium interaction sandbox environment which provides a simulated file system and shell, and allows access with random credentials after a variable number of brute force attempts. Once inside, an attacker is deceived into believing it has accessed a real system, and is observed while carrying out whatever their intention is; changing or creating files, downloading software, or altering passwords or user accounts. This type of environment is especially effective with bots, as they are automated and generally less able to identify the environment as a honeypot than a live attacker would be.

We seek to identify patterns in the activities seen in these environments and to discover any correlations, similarities, or differences in attacks identified across a globally distributed set of honeypots. This is accomplished by utilizing a series of docker containers running Cowrie to detect SSH attacks without compromising the host machine in an effort to answer these questions. In addition to Cowrie, the honeypots also utilize an Apache web server and an FTP server running in containerized environments, allowing for the collection of associated logs and tracking access attempts. Linux kernel logs from the host machines are also collected to monitor for compromise of the host device. Each honeypot is built on a Debian 10 virtual machine running in one of six Digital Ocean data centers located in geographically separate areas, including London, New York, Toronto, Amsterdam, Bangalore, and Singapore. Fake websites with domain names provided by Google Domains are utilized with names related to higher education in an attempt to attract specific kinds of traffic. Logs from these services are transferred using a cron tab and rsync on a nightly basis to a repository server where all logs are consolidated and analyzed. This study will look at data collected for the time period between March 2020 and December 2021, with the data being continuously amalgamated. Apache, FTP, and Linux kernel logs are in their standard form (apache.json.log, ftp.json.log, kern.log), while Cowrie returns a .json or .log file containing a variety of tags, including event-ID, session-ID, source-IP, destination-IP, username and password (that were used to access the honeypot), source-port, destination-port, message (commands passed to the honeypot), file hashes, and a variety of other data points.

By collecting data from the honeypots over many months we observe data and patterns of activity, and attempt to draw correlations between honeypots and regions to learn more about attacks by identifying their objectives, methods, and intent. Standard .json log files are also restructured so that session-IDs rather than event-IDs are made the key value. This allows for the collection of command chains, and makes it easier to view attack patterns and analyze attacker behavior.

## 6.2 System Implementation

To collect data on a global scale the Digital Ocean Developer Cloud is utilized, allowing for the deployment of Low-cost virtual machines placed in various data centers around the world. In order to limit resources and reduce cost, specific honeypot services are run as containers on a Debian 10 virtual machine, and are exposed to the Internet on the standard ports. Log files from each of the services are collected and forwarded via rsync to a central repository server, allowing for the periodic deletion of local files to save space. Containerized images of the running services sandbox malicious activity from the host machine. Rayson-Cowrie [99] is a version of the Cowrie honeypot running on Docker and maintained by Rayson Zhu. The Cowrie container captures log files for SSH and telnet activity in both .log and .json format, and simulates a real file system allowing attackers to execute commands, create and download files, and forward traffic, but confines these activities to the honeypot. The official Docker image of httpd, the Apache HTTP server project [100] is run in a container to host a fake website and allow exposure to the Internet without risking the host machine. Apache log files of interactions with the web services are captured and stored. The Docker image stillard-pure-ftpd [101] obtained from Github is used to host an FTP server with a few sample files is exposed and logs collected. In addition to logs from sand-boxed services, Linux kernel logs from the host machine are collected to track changes in the host and to help determine if it has been compromised.

A central repository server collects data from each honeypot, and processes log files to generate statistics about malicious traffic and to make comparisons in honeypot activity. For this work we will focus primarily on log files generated by Cowrie.

## 6.3 Analysis and Insights from Cowrie Data

Cowrie generates daily logs in both json and log formats. Json logs are built around events with each action initiated with the honeypot generating an event ID which defines a command executed by an attacker. Within an event are numerous data points, including source IP, destination IP, source port, destination port, session ID, username, password, messages (which contain commands issued by the attacker), timestamp, and many others. In this section we cover details of the collected data.

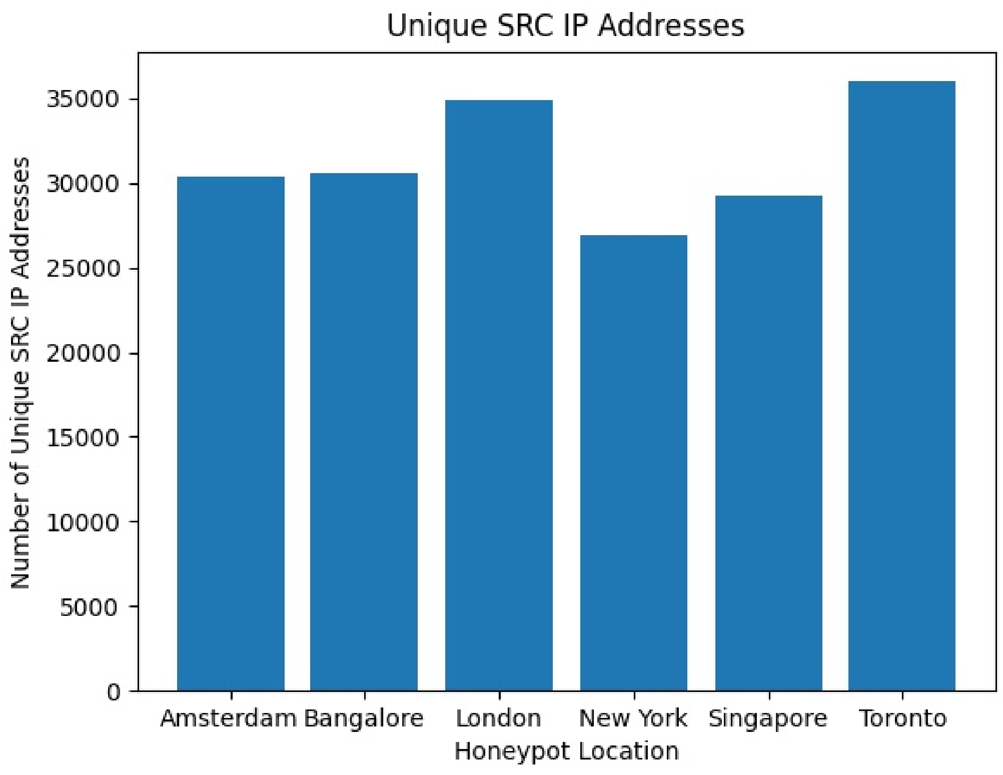


Figure 6.1: Unique source IP addresses per honeypot location.

### 6.3.1 Source IP Addresses and Port Numbers

When contact is made with the honeypot, the source IP address and port number of the machine initiating the contact is stored. From the series of commands (covered in more detail later), it appears the most common types of activity are illicit login attempts and requests to forward traffic to another device. In the first scenario, the machine initiating the contact (possibly infected by a bot) is randomly scanning the network, then performing brute force attacks on susceptible hosts using a dictionary of usernames and passwords, with the owner of the offending device remaining unaware of this activity. In the second scenario, attackers are logging in and passing traffic through the honeypot to hide their location. Attacker machines are often masked by one or more proxy, VPN, or VPS devices, so the source IP recorded may not be the actual identity of the attack origin. Recorded in Figure 6.1 are the unique source IP addresses globally that accessed or attempted to access one of the honeypots. 170865 unique IP addresses were recorded in total, with an average of 28478 unique IP addresses seen at each honeypot. These addresses were globally diverse, and appear to be random. There were also 6527 addresses which were present in all six honeypots. Random addresses, which make up the bulk of the observed source IP addresses collected, would be expected as bots scan the Internet seeking new victims. However, we see many addresses targeting the same machines which would seem to indicate credentials are being shared across a network of devices, which are in turn accessing them for the purpose of recruitment or traffic forwarding.

While there are a variety of events occurring at each honeypot, a majority of the activity seen after a successful login are *session.connect*, *direct-tcp.forward* and *direct-tcp.data* requests. Many of the login attempts are being made for the purpose of utilizing compromised machines for routing traffic through SSH tunnels. By default, SSH sets the *AllowTcpForwarding* flag to *yes*, enabling others to use

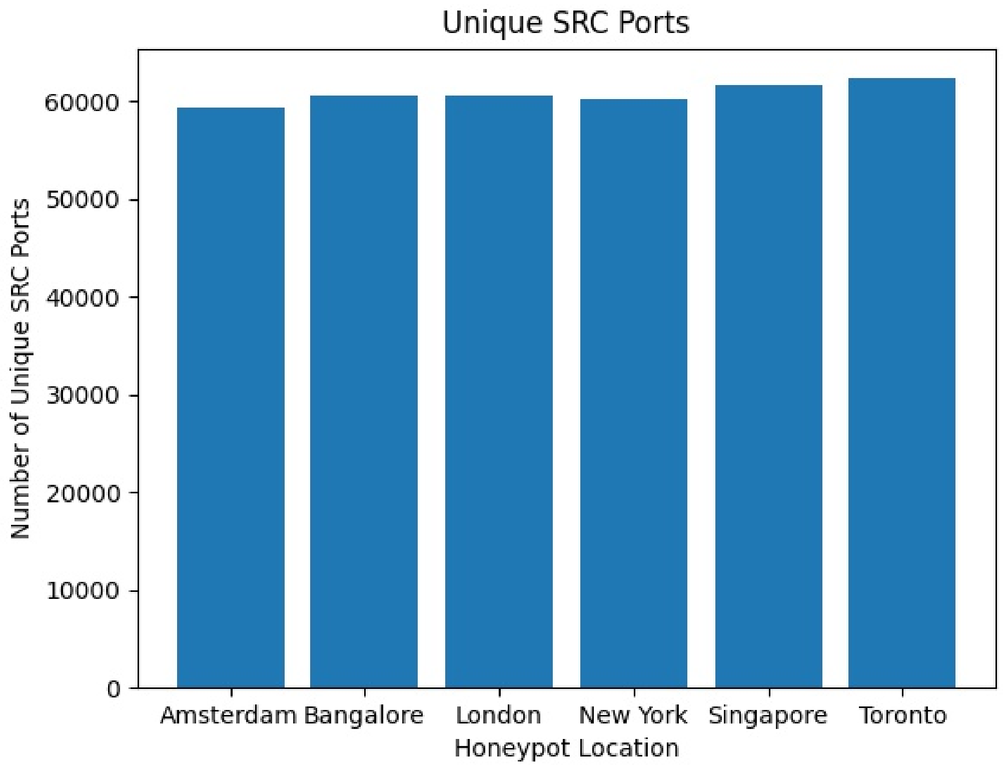


FIGURE 6.2: Unique source port numbers per honeypot location.

the victim machine as a SOCKS proxy to route any type of traffic generated by any protocol or program. To prevent this, the forwarding flag should be set to *no*. In addition, a limit on the number of login attempts allowed should be set to prevent brute force attacks. These forwarding requests are logged by Cowrie, but are not actually forwarded. Requested destination IP addresses appear to be randomly distributed between those sent directly to targeted devices and with messages routed through a known proxy device to further mask an attacker's actual location. Source port numbers on the order of 60k were present in each honeypot as seen in Figure 6.2 as bots were utilizing random port numbers to initiate contact.

Table 6.1: Series of commands per session and frequency of occurrence.

| | London | Amsterdam | Toronto | New York | Singapore | Bangalore |
|---|---|---|---|---|---|---|
| ***FADBEFADBE*** | 40.3% | 67.7% | 67.8% | 28.7% | 53.4% | 66.7% |
| ***FAGEFAGE*** | 36.1% | 21.5% | 19.3% | 36.2% | 33.5% | 15.6% |
| ***FAEFAE*** | 10.5% | 2.5% | 0.41% | 22.3% | 5.5% | 10.4% |
| ***FAGGGEFAGGGE*** | 4.1% | 0.1% | 2.6% | 5.3% | 1.6% | 3.2% |
| ***FEFE*** | 4.5% | 0.7% | 2.8% | 3.4% | 3.0% | 1.6% |
| ***FADBCEFADBCE*** | 0.1% | 0.5% | 0.8% | 0.1% | 0.2% | 0.3% |

### 6.3.2 Destination IP Addresses and Port Numbers

In order to better understand this forwarding behavior, an analysis is performed mapping attackers to their targets, and targets to attackers. Table 6.2 gives a numeric example of these mappings from the London honeypot. The left side of the table lists the top 20 attackers in terms of the number of targets each attacker can be mapped to, while the left side of the table lists the top 20 targets in terms of the number of attackers each target can be mapped to. It can be seen here there are far more targets per attacker than vise-versa, suggesting that this IP address has identified the honeypot as available for tunneling activity and is running through a list of targets using this host as a proxy. At the same time, we see these targets are being contacted by what would appear a coordinated network of attackers, some targets being accessed through all six honeypot locations. This suggests information about the honeypot and its compromised credentials are being shared across members of a botnet.

The honeypot in London saw far fewer unique destination IPs than the other honeypots (12746), while Bangalore saw the most (100252). Totals for all honeypots can be seen in Figure 6.3 This is probably an indication of the types of attacks being carried out, possibly recruitment versus tunneling activity, although it is unclear why the London honeypot was being utilized differently over this time period. As for destination ports, there were 1038 port numbers targeted from Bangalore and 718 from Amsterdam, while the other honeypots ranged between 29-57, see Figure 6.4. It is likely these two locations were engaged in scanning

Table 6.2: Mapping of attackers to targets and targets to attackers.

| ***Attackers to Targets*** | ***Targets to Attackers*** |
|---|---|
| ***5.182.39.88***:*114839* | ***google.com***:*872* |
| ***5.182.39.61***:*50318* | ***ya.ru***:*871* |
| ***5.182.39.62***:*25441* | **208.95.112.1**:609 |
| ***5.182.39.64***:*16741* | ***216.239.32.21***:*492* |
| ***5.188.62.11***:*13714* | ***216.239.36.21***:*443* |
| ***5.182.39.6***:*13049* | ***216.239.38.21***:*374* |
| ***45.227.255.163***:*7023* | ***216.239.34.21***:*338* |
| ***88.214.26.90***:*5120* | ***v4.ident.me***:*270* |
| ***5.182.39.185***:*3081* | ***video-weaver.arn03.hls.ttvnw.net***:*138* |
| ***45.227.255.205***:*762* | ***ipinfo.io***:*118* |
| ***5.182.39.96***:*443* | ***www.instagram.com***:*113* |
| ***5.188.86.172***:*233* | ***www.youtube.com***:*102* |
| ***88.214.26.93***:*166* | ***ip.bablosoft.com***:*101* |
| ***193.105.134.45***:*119* | ***video-weaver.waw01.hls.ttvnw.net***:*101* |
| ***103.114.104.68***:*60* | ***104.16.119.50***:*101* |
| ***51.158.111.157***:*53* | ***104.16.120.50***:*101* |
| ***45.155.205.87***:*44* | ***speedtest.tele2.net***:*96* |
| ***79.173.88.244***:*36* | ***87.250.250.242***:*93* |
| ***14.177.178.248***:*30* | ***m.youtube.com***:*90* |
| ***14.186.28.128***:*29* | ***check2.zennolab.com***:*89* |

activity, while the others were focused on tunneling data to selected targets. London, Toronto, and New York were targeting mainly common services such as web, telnet, smtp, ssh, etc. Bangalore and Amsterdam seemed to target these, as well as many non-common port numbers associated with specific services (i.e. bo2k or Ghidra) that could have been identified by nmap scans.

We look specifically at tunneled traffic, examining the number of attackers and targets present in each honeypot, and then looking for their presence across all honeypots. Figure 6.5 shows that there are no attackers that are found in every location (although most are found in more than one honeypot), while Figure 6.6 shows there are 1045 targeted IP addresses that all six honeypots have in common. It would appear that while there are high value targets being sought by more than one attacking entity or botnet, no individual attacking IP is seen in every honeypot.

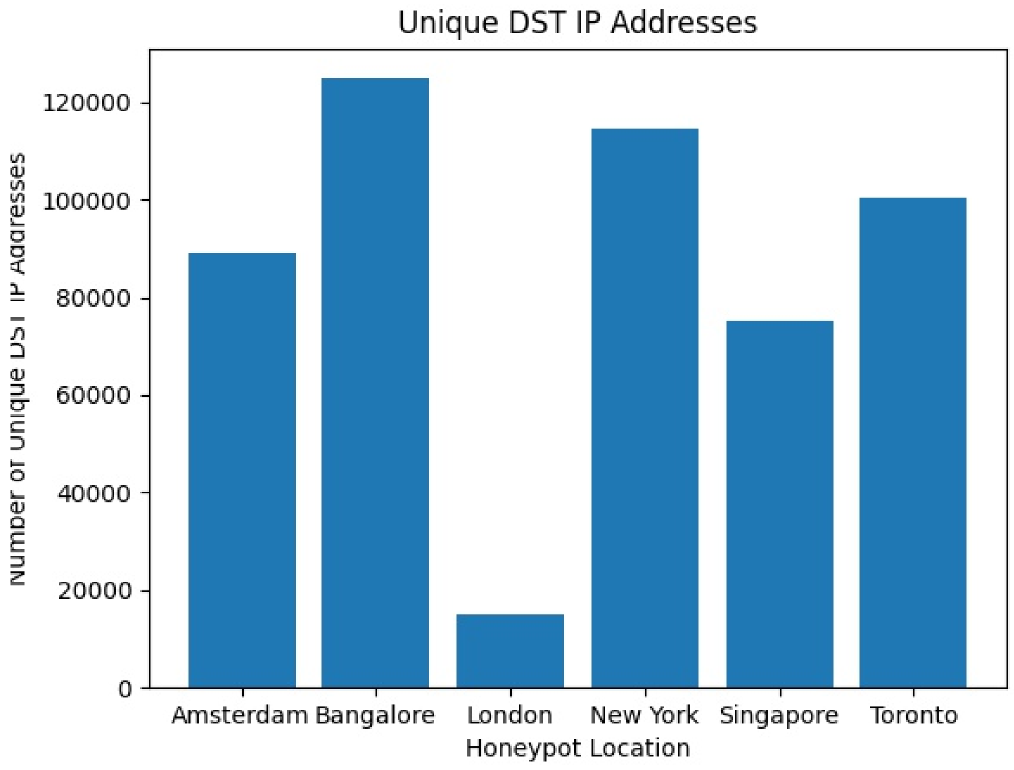


Figure 6.3: Unique destination IP addresses per honeypot location.

### 6.3.3 Daily Events

As mentioned previously, Cowrie .json logs are built around events, with multiple events often contained within a single session. A unique session ID is created when a connection is initially established, and the session is terminated when the connection is eventually closed. Figure 6.7 shows the number of unique events per day across all honeypots, while Figure 6.8 shows the total of all unique sessions per day across honeypots. Interestingly, there are several spikes in both events and sessions, indicating increased activity across different honeypots on the same days even though they are located in geographically separated regions. An attempt was made to correlate these peak days with other events such as news items or known attacks, but with no convincing results. There were several spikes in April 2020, likely due to the beginning of the Covid-19 pandemic and a surge in the number of people working on machines no longer protected by workplace security, as well as quarantined individuals being online at home more than usual.

### 6.3.4 Cowrie Sessions

To bring event IDs into perspective, Cowrie sessions can be used to aggregate a series of events under a single session, giving a better view of command patterns and attacker behavior. By restructuring the Cowrie log as a dictionary with the session ID as a key rather than the event ID, all events containing the same session ID can be consolidated, and an order of events can be captured. There were 16 possible event IDs logged by Cowrie indicating actions such as a new connection, the success of a login attempt, a file download, a message being forwarded, etc. To make these aggregated lists of events easier to analyze, we assign a letter value *A*-*P* to each command, then build a string based on the sequence of commands

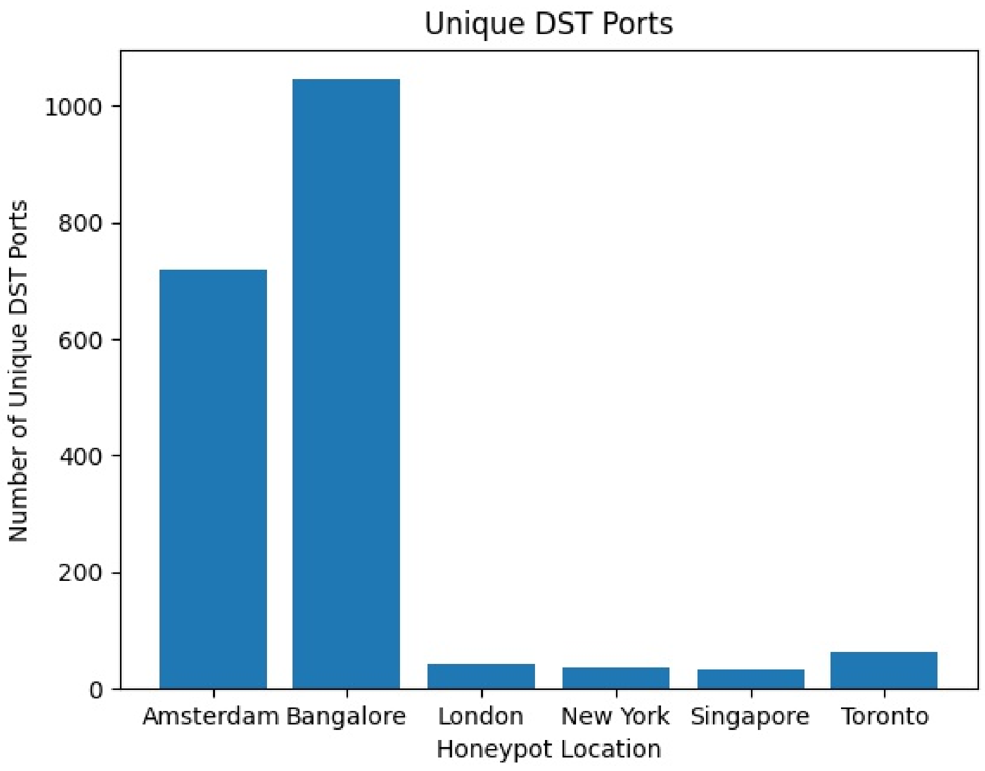


FIGURE 6.4: Unique destination port numbers per honeypot location.

executed during each distinct session. Table 6.3 lists the most common of these $A$-$G$, the others were omitted for space. The top six command sequences are listed in table 6.1, along with the frequency of their occurrence compared to all identified sequences.

TABLE 6.3: Command legend.

| | |
|---|---|
| ***A*** | ***cowrie.client.version*** |
| ***B*** | ***cowrie.direct-tcp.request*** |
| ***C*** | ***cowrie.direct-tcp.data*** |
| ***D*** | ***cowrie.login.success*** |
| ***E*** | ***cowrie.session.closed*** |
| ***F*** | ***cowrie.session.connect*** |
| ***G*** | ***cowrie.login.failed*** |

The most common sequence involves a connection (cowrie.session.connect), followed by the cowrie version supplied as part of the ssh handshake (*cowrie.client.version*),

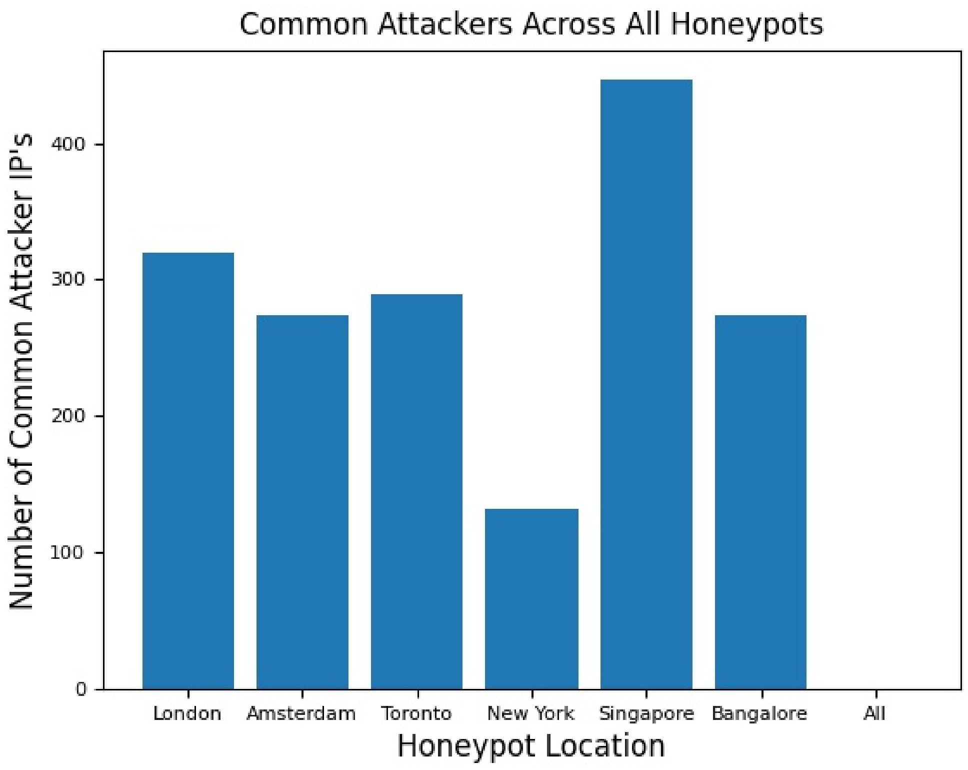


FIGURE 6.5: Common attacker IPs across honeypots.

an indication of successful login (*cowrie.client.success*), then a request for a direct-tcp connection (*cowrie.direct-tcp.request*) allowing the attacker to pass data through the honeypot to another destination. Finally, the connection is closed (cowrie.session.close). For the honeypot in London, of the 874782 sessions on April 12, 2020, about 40.3 percent, or 352537 of these followed this pattern. For the London honeypot there were 4459 unique command patterns captured in total. Most were of the common type indicated here, which were quite specific and relatively short. However, many individual unique command patterns found that were very long and exhibited repetitive patterns (some entailing hundreds of individual commands). We believe these likely contain repeated patterns that could be associated with an individual bot. This analysis will make an interesting future work as it could help to identify behaviors indicating malicious activity and aid in attacker identification.

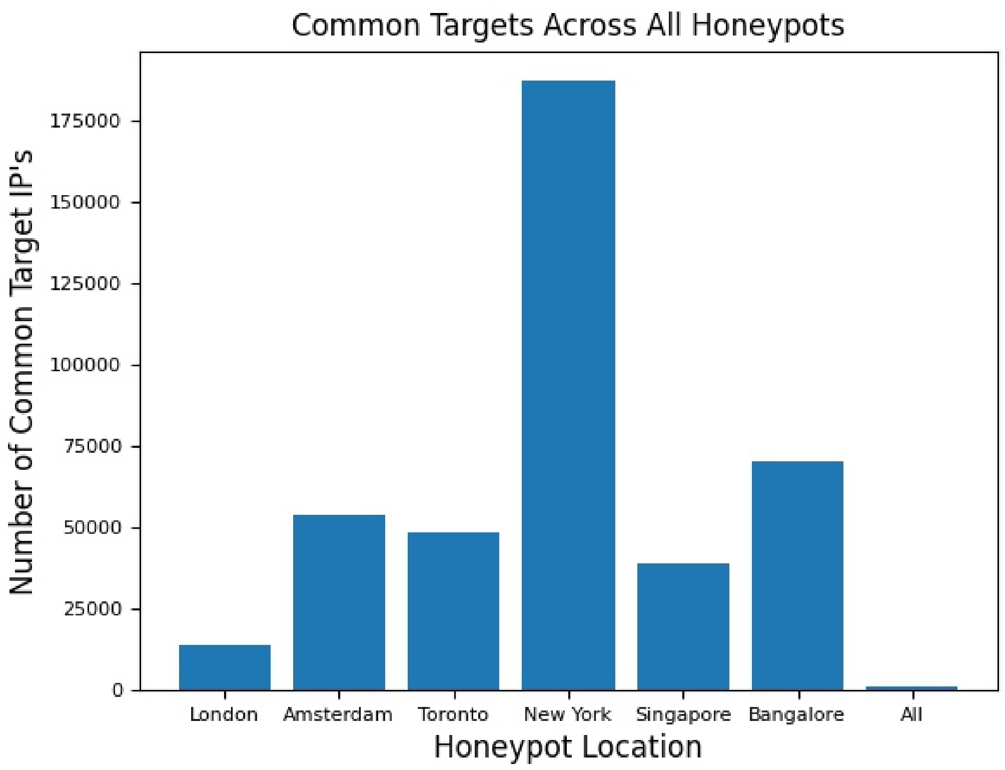


FIGURE 6.6: Common target IPs across honeypots.

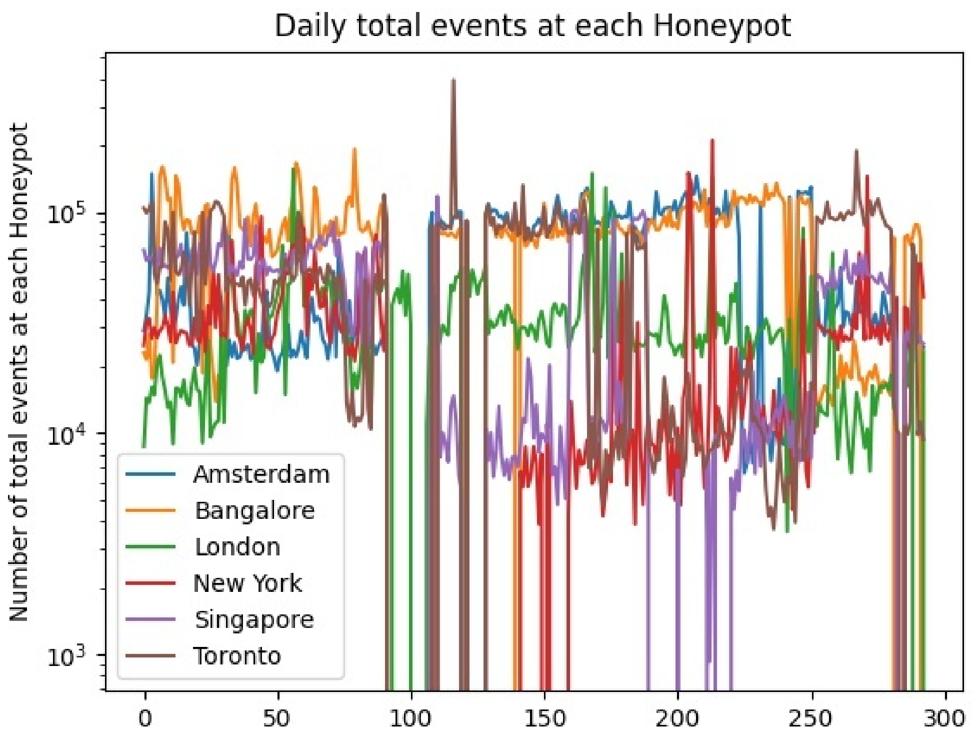


FIGURE 6.7: Daily events per honeypot.

### 6.3.5 Usenames and Passwords

Each login attempt, whether successful or not, captures the credentials that were given. A majority of these credentials attempt privileged access (either 'root', 'admin', or some other known default credential such as 'ubnt' for a Ubiquiti device), but many do not and instead utilize random usernames and a wide variety of passwords. As discussed in [58], this data is very useful as a Cyber Threat Intelligence (CTI) feed, allowing for compromised usernames and passwords to be black-listed from a system. Figure 6.9 shows the number of unique usernames found in each honeypot over the duration of this study, while Figure 6.10 shows all unique passwords used to gain access. Cowrie randomly accepts any login credentials after a variable number of attempts (set at three for our application), in an effort to "fool" attackers into believing they have correctly guessed login

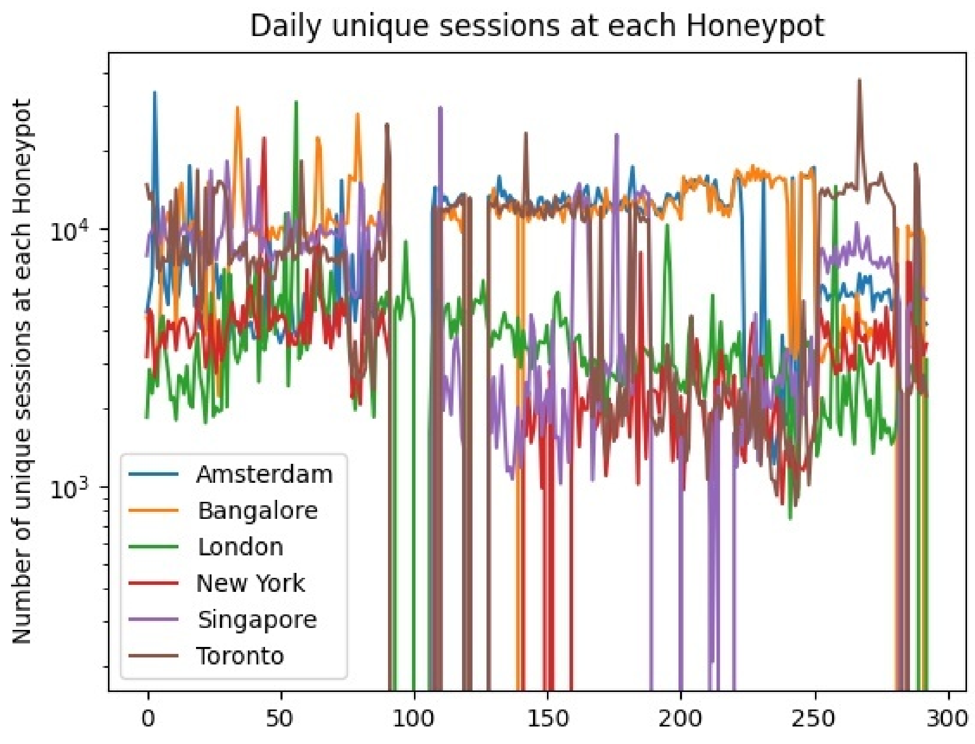


FIGURE 6.8: Daily sessions per honeypot.

credentials. The most commonly used usernames and passwords are listed in table 6.4.

### 6.3.6 Malicious Downloads

Some Cowrie event contains a key *messages* that detail a command being executed in the honeypot. If a search is done for the string *wget* one can find attempts by an attacker to retrieve files from a remote host and download it to the honeypot. These are typically shell scripts that are then made executable using a *chmod* command either in the same message, or in a subsequent message, and are then run in the compromised machine. A list of these compromised files is gathered and could be used as a CTI feed to identify known malicious filenames. The IP addresses or URLs indicated in these download commands can be considered highly malicious, either as command and control devices, or more likely as file storage devices used for downloading malicious software. Again, these are often routed through a VPN or VPS. To help pinpoint likely sources for these downloads, a list of known VPN and data center addresses, and an api (getipintel.net) are used, and a list of possible actual download IP addresses is compiled. 1564 unique IP addresses are identified that do not appear in the known VPN or data center list. As a future work, it would be interesting to retrieve these shell scripts and analyze them forensically. The top 20 malicious file names, all of which appear in each of the six honeypots is given in table 6.4.

Given the commonality of source and destination IP addresses seen in attacks on the honeypots and the fact that many of the attack patterns present in collected honeypot data are consistent with the Mirai botnet and its variants, it is apparent that botnet activity continues to be a major security concern. With simple IoT

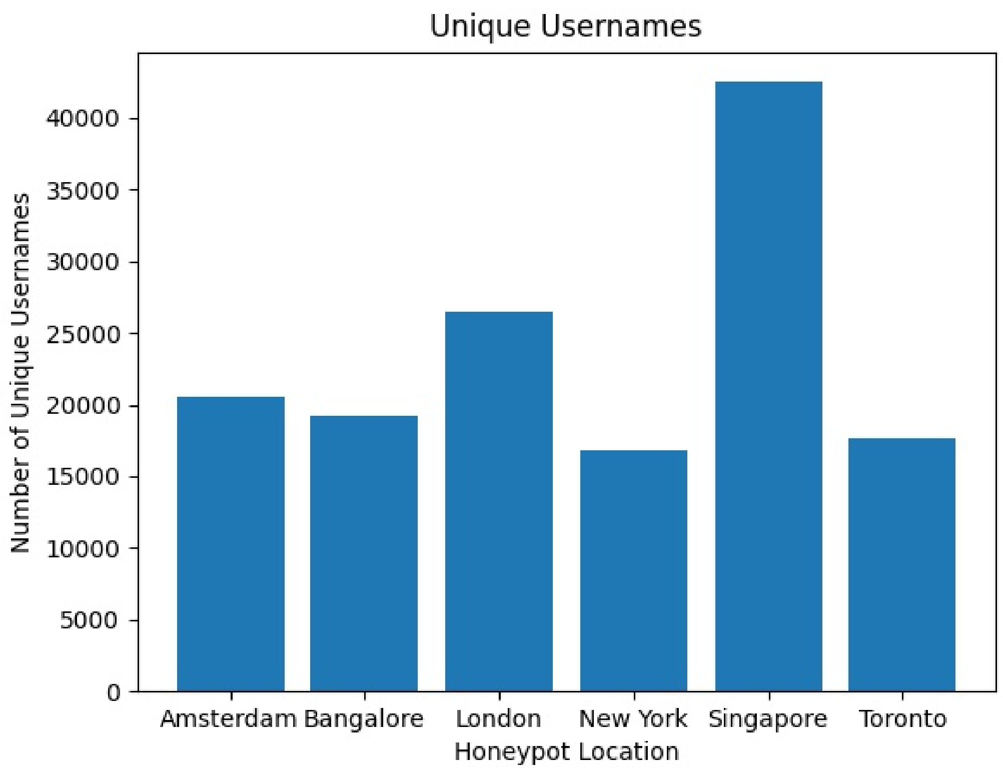


FIGURE 6.9: Unique usernames per honeypot.

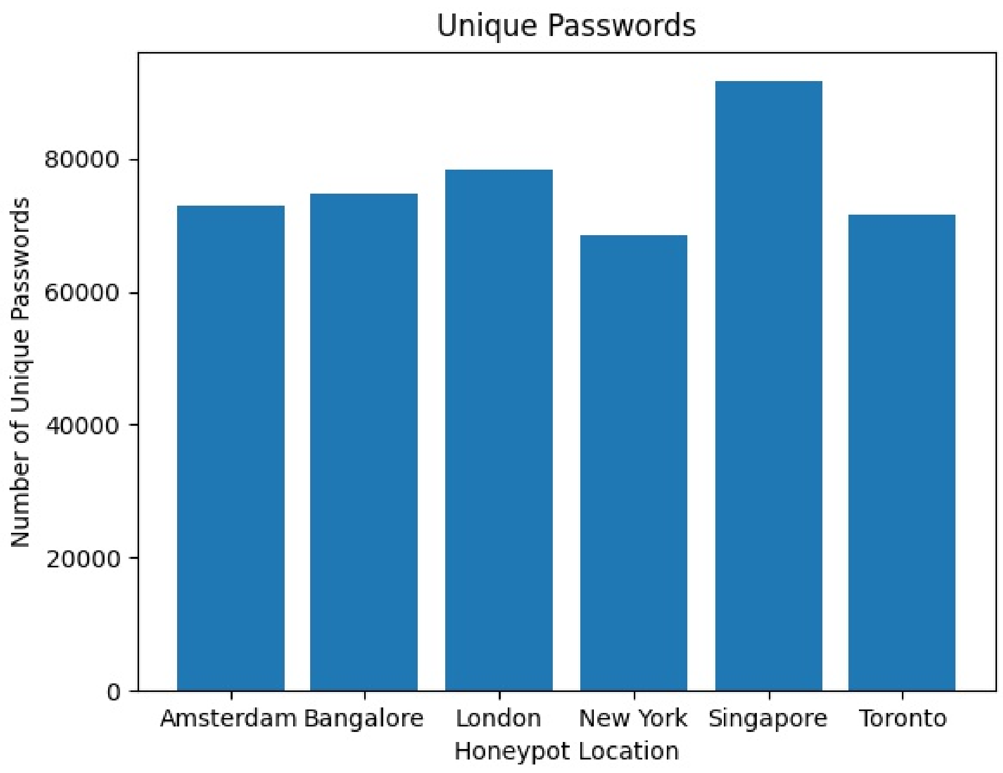


FIGURE 6.10: Unique passwords per honeypot.

devices being the prime target for such attacks, continued development of techniques for mitigating these attacks is crucial. In our opinion, the first step for management and defense must be proper accounting and monitoring of IoT devices within networks to detect abnormal behavior as well as to identify unknown devices. To accomplish this, fingerprinting devices based on network traffic is essential as other identifiers such as IP and MAC addresses can be spoofed making detection using these parameters unreliable.

## 6.4 Conclusion

To better understand attack patterns and behavior, honeypots can be deployed to monitor and log activity by simulating actual Internet facing services. By examining traffic patterns, downloads, and traffic forwarding across a series of

Table 6.4: Most used usernames, passwords and download filenames.

| ***Usernames*** | ***Passwords*** | ***Malicious Filenames*** |
|---|---|---|
| ***root*** | ***guest*** | ***bins.sh*** |
| ***guest*** | ***admin*** | ***GhOul.sh*** |
| ***admin*** | ***root*** | ***yoyobins.sh*** |
| ***test*** | ***test*** | ***SnOopy.sh*** |
| ***user*** | ***123456*** | ***axisbins.sh*** |
| ***ubnt*** | ***1234*** | ***EkSgbins.sh*** |
| ***0101*** | ***user*** | ***8UsA.sh*** |
| ***nproc*** | ***password*** | ***Pemex.sh*** |
| ***22*** | ***ubnt*** | ***Sakura.sh*** |
| ***support*** | ***0101*** | ***sh*** |
| ***oracle*** | ***123*** | ***skid.sh*** |
| ***postgres*** | ***nproc*** | ***UwU.sh*** |
| ***ubuntu*** | | ***zeros6x.sh*** |
| ***usario*** | ***support*** | ***mavscock.sh*** |
| | ***matrix*** | ***KigaNet.x86*** |
| ***git*** | ***12345*** | ***infn.x86*** |
| ***Administrator*** | ***usario*** | ***ISIS.sh*** |
| ***pi*** | ***123456789*** | ***installer.sh*** |
| ***1234*** | ***12345678*** | ***Gummy.sh*** |
| ***ftpuser*** | ***1*** | ***gtop.sh*** |

geographically separate devices we attempt to better understand malicious activity and work to identify patterns that can be useful in identifying malicious traffic in actual servers or IoT devices. In addition, collected data such as source IP addresses, download filenames, and login credentials can be useful as a threat intelligence feed to protect digital assets. As a future work more analysis on command series patterns could be conducted to possibly predict attack behavior and to identify threats in real time on production servers and IoT assets.

The Cowrie honeypot is a valuable tool for capturing samples of traffic and real-time interaction with botnets. Analysis has shown much of the traffic occurring across these honeypots are generated from Mirai and Mirai-variant bots, and it is likely that such threats will persist for some time to come [12]. Given the reality of this persistent threat, more work will need to be done to develop methods for monitoring IoT devices and device membership on networks to assure secure operation and management. In the following section we study methods for developing network traffic fingerprinting on IoT as a means to this end.

# Chapter 7

# Heterogeneous Device Fingerprinting

## 7.1 Introduction

Keeping track of IoT devices in a network can prove to be a challenging task as device identity can be easily masked. In order to accomplish this we focus on identifying devices using a fingerprint of their network traffic that can be correlated to a specific device. Our approach utilizes the principle of *locality sensitive hashing* to generate a feature vector which will be used by a classifier in fingerprinting IoT devices based on their generated network traffic. In this section we will describe this technique and provide details on both the Nilsimsa hash and how it is used in combination with a convolutional neural network to generate such a fingerprint.

Unlike cryptographic hashes which produce an entirely different result even if a very small change is made to the input data, locality sensitive hashing (LSH) is a hashing algorithm wherein small changes to the input data produce a very similar output. This is accomplished by placing similar input items into common buckets with a high probability, with hash collisions being maximized rather than minimized. LSH has been effectively utilized for tasks such as electronic identification [102], anomaly detection [103], malware classification [104], and spam email detection [105] [106]. There are several tools available for generating locality sensitive hashes, such as *Nilsimsa*, *ssdeep*, *sdhash* and *tlsh* [107]. For our work we utilize Nilsimsa as it provides a digest which displays little variance for similar inputs, and can easily be broken into a fixed number of octets representing the input source.

In the following we discuss locality sensitive and locality preserving algorithms in general as they make up a vital core of our work. Here we will consider the families of locality sensitive hashing approaches, and the major applications in use. The most widely used n-gram versions of these algorithms are ssdeep [108], sdhash [109], tlsh [110], and Nilsimsa [105].We cover these for reference, then focus more specifically on Nilsimsa as the starting point for our work fingerprinting network traffic, and the inspiration for our novel algorithm *FlexHash*.

## 7.2 Locality Sensitive Hashing

Hashing techniques provide a means of mapping high-dimensional data to a fixed length digest, i.e. all resultant hashes of a given hash function will be of length $X$. Locality sensitive hashing (LSH), also called *fuzzy hashing* is a specific set of algorithms that apply a probabilistic function to perform dimensional reduction while preserving the relative distance between objects. This is done by mapping the input data to a set of buckets in such a way as to maximize the probability of collisions, thus grouping similar items. This is contrary to the more familiar cryptographic hashes (MD5, SHA-1, SHA-2, etc.) wherein even a small change to the input data results in a completely different output hash by minimizing collisions and applying a cascading effect. In other words, for a locality sensitive hash a small change in the input data will result in a small change in the output hash. Rather than assigning a unique identifier to an item, LSH seeks to approximate a match to items that are likely similar, as in the case of data clustering or a nearest neighbor search. There are several *families* of hashing functions that can be applied based on the type of input data.

### 7.2.1 LSH Families

There are several common approaches to LSH depending on the data in question:

- E2 LSH
- MinHash
- SimHash
- Random Binary Projections
- K-Means LSH
- Bayesian LSH
- Hamming LSH

#### 7.2.1.1 E2 LSH

E2 LSH is an LSH algorithm that provides a randomized solution for finding the nearest neighbor in Euclidean space $l_2$. This approach is useful when it is necessary to solve the nearest neighbor problem where, given a query $q$, the data should return the point `P` that is nearest $q$ [111]. Euclidean distance is useful when measuring the length of a line segment in space, and can be an indication of similarity between two data points. By preprocessing objects in a dataset, subsequent items

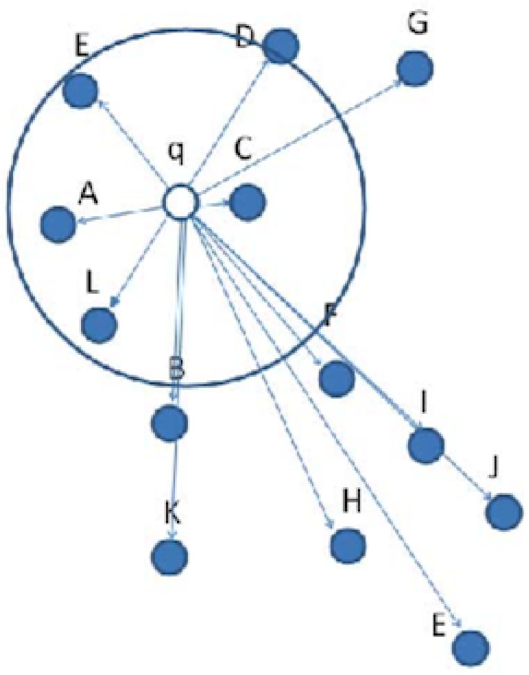


Figure 7.1: Data points are hashed in such a way that similar items will fall into buckets that are near one another. When mapped to Euclidean space, points within a given radius (nearest neighbors) should be similar [2].

can be compared to find a nearest neighbor to determine likely similarity. This is done by applying hashing functions that with a high probability map similar items to nearby *buckets* or memory locations. Their Euclidean distance from one another then represents how similar they likely are. A representation of similar items mapped in Euclidean space can be seen in Figure 7.1 where points similar to $q$ should fall within a given radius of $q$.

#### 7.2.1.2 MinHash

MinHash, or *min-wise independent permutations locality sensitive hashing scheme* is an algorithm for estimating the similarity between two sets. This LSH scheme was first used in the Alta Vista search engine and could detect and remove duplicate web pages for more efficient search results. It has also been used as a clustering tool when searching large numbers of documents based on similarities in sets of words they contain [112]. MinHash relies on the Jaccard similarity coefficient, which is a ratio of the number of elements found in the intersection and the union of two sets. Specifically, the Jaccard coefficient is the size of the intersection divided by the size of the union. Using a technique known as *shingling*, this method allows for the random sampling of words from a large body of sets, hashing them as integers, then arriving at an acceptable distribution of hash codes to represent similarity. In this way, documents can be quickly compared for reasonable similarity without the time and computational constraints of a brute-force search for matches.

#### 7.2.1.3 SimHash

SimHash is an algorithm for calculating near-duplicates by tokenizing strings, then creating vectors using term frequency-inverse document frequency (TF-IDF). TF-IDF is a widely used method for text representation. Weights are assigned to words in a document based on their frequency in the document and inverse frequency in the corpus. Words that occur more frequent in a document but less frequently in the corpus are deemed to be important for that document's meaning. This approach is commonly used for information retrieval and text classification tasks. SimHash develops a binary fingerprint by comparing these weighted feature vectors and assigning a `1` to a positive value, otherwise it is set to `0`. Hamming distance measure can be applied to the resultant binary string, with a small distance between strings representing greater similarity. This is a useful algorithm when performing document de-duplication, spam detection, or near-duplicate (plagiarism) detection, and has been utilized by the Google crawler to eliminate duplicate web pages in their index.

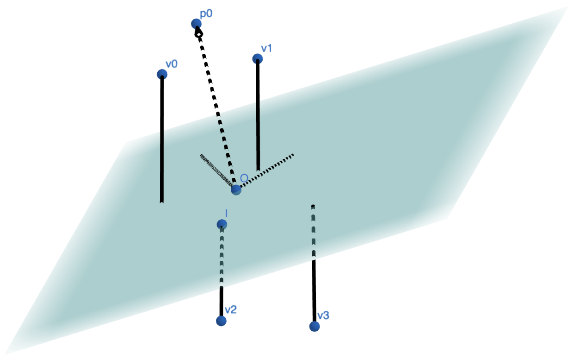


FIGURE 7.2: Data points are separated by random hyperplanes, and points are assigned a *1* or *0* depending on whether they fall on the same side of the plane as a normal vector [3].

#### 7.2.1.4 Random Binary Projection

LSH using random binary projection works by reducing highly-dimensional vectors into low-dimensional binary vectors, making it useful for approximate similarity searches in very large data sets. Hamming distance can then be calculated on the binary vectors to estimate their similarity. Hyperplanes are created through the data points, and point on one side of the plane or the other are assigned either a `0` or a `1`, essentially splitting the data into two groups. To determine which side of the hyperplane data resides, a normal vector of the plane is calculated, and is combined with a data point vector in a dot product function. If the two vectors share the same direction, the dot product will be positive, otherwise it will be negative. As successive hyperplanes are added and this process is repeated, the amount of encoded data increases as seen in Figure 7.2. Binary vectors can then be compared using Hamming distance to determine similarity.

#### 7.2.1.5 K-Means LSH

K-Means LSH uses the K-Means algorithm to cluster data by partitioning $n$ observations into $k$ clusters. Vector quantization is used to assign every observation (data point) to the nearest randomly chosen centroid, or *mean*, which serves as the

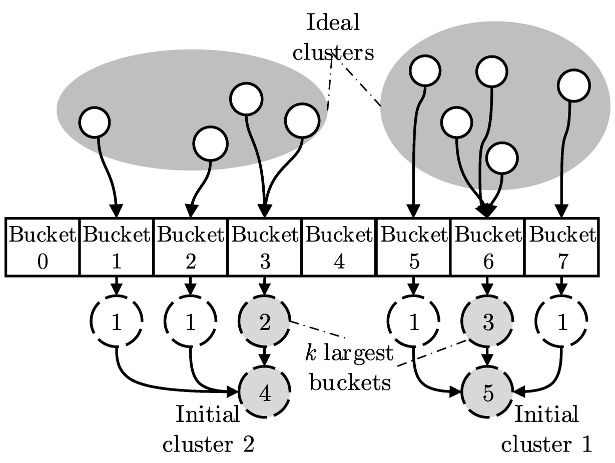


FIGURE 7.3: Data points are grouped into *clusters* by randomly selecting means, or centroids, then iteratively moving these centroids until they most closely represent the center of the cluster [4].

temporary prototype for the cluster. Through an iterative process, the centroid or mean is moved to minimize the sum of squared distances between the data points in the cluster and their corresponding centroid. The goal is to create clusters that are as distinct as possible, separating clusters from one another and maximizing the distance between centroids. Data that falls within a given cluster represents similarity between the data points, thus achieving the goal of locality sensitive hashing. Points within a given cluster are mapped to similar buckets as shown in Figure 7.3

#### 7.2.1.6 Bayesian LSH

Bayesian optimization is a method for tuning hyperparameters by learning from past performance metrics to improve search speeds. When applying a similarity measure to a collection of objects, the goal is to find all pairs of objects with a similarity greater than a given threshold. Bayesian LSH performs candidate pruning and similarity estimation using LSH to remove a large number of the false-positive candidate pairs leading to faster similarity searches.

#### 7.2.1.7 Hamming LSH

Hamming distance is a way to measure similarity in binary strings. The distance between two strings or vectors of numeric values of equal length can be defined as the number of positions in the vector where the corresponding characters are different. More simply put, the Hamming Distance is a measure of the difference in symbols in equal length strings or vectors. In the case of a binary string, the Hamming Distance would be equal to the number of 1's that result from an XOR operation between the two strings. A low distance indicates greater similarity, making this approach favorable to strings or data that can be represented as numeric vectors [113]. This measure was utilize by [15] in developing fingerprints for IoT traffic and served as a starting point for our work. This measure was applied using Nilsimsa[114] to produce fingerprints of network traffic, applying similarity to determine matches in traffic samples stored in a database.

### 7.2.2 Significant n-gram Based LSH Tools

Here we consider the aforementioned LSH tools (ssdeep, sdhash, tlsh, Nilsimsa) in view of the hashing families they utilize. We cover these four for perspective, then give more in depth detail on Nilsimsa and how it applies to this work.

#### 7.2.2.1 ssdeep

ssdeep was introduced by Jesse Kornblum in 2006 [115]. ssdeep is based on work done by Michael Rabin on data fingerprinting with random polynomials at Harvard University in 1981. The tool was one of the original works on context triggered piecewise hashes (CTPH), also referred to as fuzzy hashes. The approach follows the following steps:

- Use a rolling hash function, or shingling, to break a file into smaller pieces

- Use a function to produce a hash for each small piece
- Concatenate the smaller hash functions into a signature representing the entire input data

Using this method, files that exibit similarity should produce similar hash signatures. Files that are dissimilar should produce dissimilar hash signatures. It is important that in the process of breaking the input data into smaller pieces that small changes in input produce only small changes in output. To accomplish this the *rolling hash* function uses a few bytes of data at each iteration, incrementing one byte each time (sliding window), with the results stored in an accumulator of a set length, in this case 80 characters. This becomes the file signature, or fingerprint.

To compare file signatures for similarity, the edit distance between two signatures is calculated. It would be expected that, for related (similar) files, it would take fewer editing operations (insert/delete/change/swap) to transform between signatures.

#### 7.2.2.2 sdhash

sdhash [116] takes a different approach to develop, mangage, and compare its similarity hashes, also known as similarity digests. This approach is similar to a random projection method where the distance between two feature vectors is the cosine distance between them. sdhash also uses a sliding window to perform shingling, with the window size set to 64 bytes. To determine similarity, sdhash tries to find the 64-byte sequences from each neighborhood of the input data that have the lowest probability of being encountered by chance (most unique features). Each of these is hashed and is entered into a Bloom filter. Once the filter reaches capacity a new filter is started until all of the features are accounted for. Ultimately a digest is formed of the sequence of Bloom filters that represents about 3% of the length of the original data, mapping the original input to a compressed output. As Bloom filters have predictable probabilistic properties [117], two filters can be compared for similarity using Hamming distance measurement. For comparison, each filter in the first digest is compared with its maximum match in the second, and the resulting matches are averaged. sdhash then computes a normalized Shannon entropy measure and places features into 1000 classes of equivalence, with statistics for similarity being calculated based on this approximation. sdhash outperforms ssdeep in every category [118].

#### 7.2.2.3 tlsh

tlsh constructs a digest for similarity measure using the following steps [110]:

- Populate an array of buckets by processing the data byte for byte with a 5-byte sliding window
- Calculate quartile points *q1, q2* and *q3*
- Construct the digest header values
- Construct the digest body by processing bucket arrays

The sliding window operates as in the above examples, with a 5-byte window passing over the entirety of the data one byte at a time. When this is complete, there remains an array of bucket counts. Quartiles of these counts are calculated such that:

- 75% of the bucket counts are greater than or equal to *q1*
- 50% of the bucket counts are greater than or equal to *q2*
- 25% of the bucket counts are greater than or equal to *q3*

The first three bytes of the hash are considered to be the header. Byte number 1 is a modulo 256 checksum of the byte string. Byte number 2 is equal to a logarithmic representation of the length of the string modulo 256. Byte number 3 is made up of 2-16 bit quantities calculated from the quartiles *q1, q2* and *q3* in the following way:

- q1-ratio = (q1*100/q3) MOD 16
- q2-ratio = (q2*100/q3) MOD 16

The rest of the digest is calculated using the bucket array with:

```
For bi = 0 to 127
    if bucket[bi] less than or equal to q1: Emit(00)
        else if bucket[bi] less than or equal to q2: Emit(01)
        else if bucket[bi] less than or equal to q3: Emit(10)
        else Emit(11)
```

Finally, the tlsh digest used for determining similarity is constructed by concatenating:

- The hexadecimal representation of the digest header values
- The hexadecimal representation of the binary strings (bucket array)

Leaving out some of the complexities of the process, tlsh uses a hamming distance measurement to determine similarity between two digests; a greater distance equates to less similarity.

#### 7.2.2.4 Nilsimsa

Nilsimsa is an open source n-gram based LSH generator that was originally designed to identify spam email [114]. As such, it is capable of taking in a text or byte string of any length and returning a fixed length digest. Nilsimsa utilizes a 5 character window that moves over the text of the message one character (or byte) at a time (see Figure 7.4). Each time a new character enters the window, the algorithm generates all possible trigrams (t) associated with the characters in the window and passes each them individually to a hash function h(). The function h() computes a hash value i = h(t) between 0 and 255 that corresponds to the i-th cell in an array of integers of size 255, called accumulator, and whose value at that location is increased by 1. Each cell acts as a counter for the number of collisions at that cell. Once all characters have been processed, an average is calculated for all cells, and a median value is assigned. If the value in the cell is less than or equal to the median, the cell value is set to 0, otherwise it is set to 1. The result is a 256 bit digest in binary form. This can be converted to a 32 byte hexadecimal code.

To determine similarity, the digests are compared, checking the number of bits in the same cell position for two different digests. The Nilsimsa Compare Value is the number of bits that are equal in two digests minus 128. For two randomly

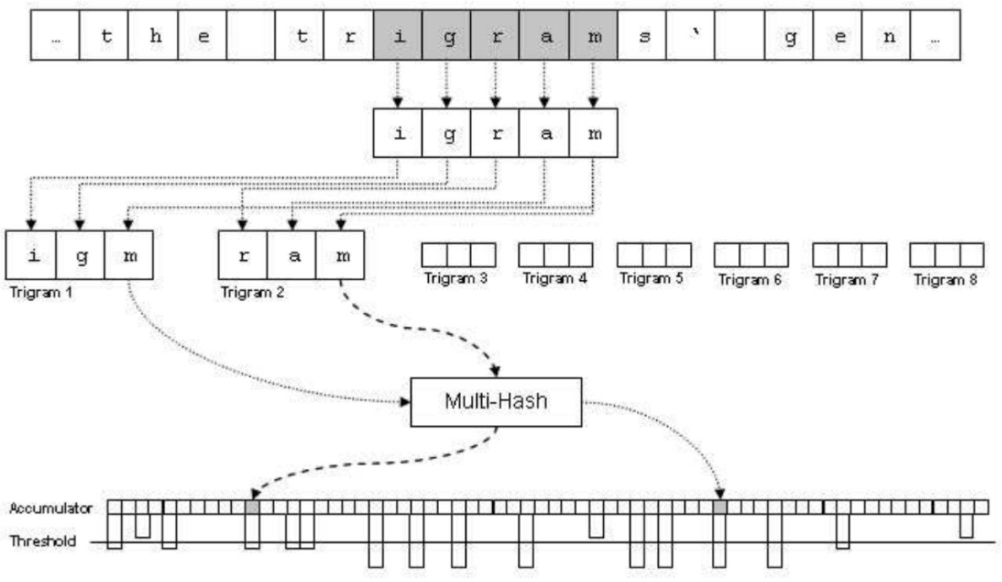


FIGURE 7.4: Construction of a Nilsimsa hash

chosen 256-bit sequences, we expect an average of 128 equal bits, that is, a Nilsimsa Compare Value equal to zero. The maximum value of the Nilsimsa Compare Value is 128, for two identical digests. This is essentially a variant of the Hamming distance.

As seen in Figure 7.4 Nilsimsa uses a fixed-size sliding window of 5 bytes which analyzes input data on a per-byte basis, producing a series of trigrams of possible combinations of the input characters. These trigrams are mapped to a 256 bit array, or accumulator, to ultimately create a hash of the data. Each time a *bucket* in the array is accessed its value is incremented, and at the end of the process if the cumulative value of the bucket exceeds a given threshold its position in the array is reset to 1, otherwise it is set to 0 producing a 32 byte digest. Typically these digests are measured against one another for similarity by comparing each bit and producing a score of between -128 and 128 by checking the number of bits that are identical in the same positions and either adding or subtracting them, with the highest score indicating the greatest similarity. In our approach we do not utilize this scoring system, but instead use the resulting 32 bytes of the hash to produce a set of *features* for use with a more complex machine learning algorithm.

At this time it would be appropriate to mention the work done by Batyr Charyyev et. al in [15] wherein Nilsimsa similarity scores were generated for network traffic from a group of 22 heterogeneous IoT devices. Signatures (or fingerprints) for each device were stored in a database, then when unknown traffic was analyzed, it was compared to the stored fingerprint, and if the similarity was above a given threshold, it could be assumed the traffic was generated by that device. Although we don't use a similarity score for identification, the preliminary steps Nilsimsa takes to generate a digest are very interesting, and we have found that these can be transformed into feature vectors that can then be used with a convolutional neural network to develop a model for device classification. We test this method against the same 22 devices to demonstrate its effectiveness while using only a single packet of data rather than requiring a longer traffic sample.

## 7.3 Methodology

While locality sensitive hashes have been shown to be useful on their own in classifying network traffic, they have not been able to offer good performance with smaller data flows and are at a disadvantage to more complex approaches such as neural networks. Given a complex input feature space, a reasonable number of samples, and a network architecture of sufficient size a neural network can do the heavy lifting necessary for discovering correlations between various input variables which are necessary for separating the data into the labeled classes. However, if the input feature space is too large it will exceed even a neural network's ability to recognize patterns. For this reason it is difficult to directly apply neural networks to raw .pcap (network traffic flow) files without extensive feature extraction and engineering. To simplify the process our approach utilizes Nilsimsa to reduce the dimensionality of the raw data and form features that a neural network can process. By breaking Nilsimsa's 32 byte digest into individual bytes and converting these into integer values between 0-255 that are readable by the classifier, a single packet or an entire traffic flow can be interpreted as a suitable feature vector, as seen in Figure 7.5. The advantage of this approach, apart from increased accuracy, is that rather than requiring a 10-minute flow of traffic to capture a good similarity sample, a single packet is all that is needed for accurate device identification. This approach requires no feature extraction, no particular domain expertise for feature engineering, and is resistant to changes in the network.

We elect to use a *Multi-Layer Perceptron* (MLP) [119], an example of a relatively simple neural network. The MLP is made up of four primary components: an input layer, hidden layers, non-linear activation function, and an output layer. In

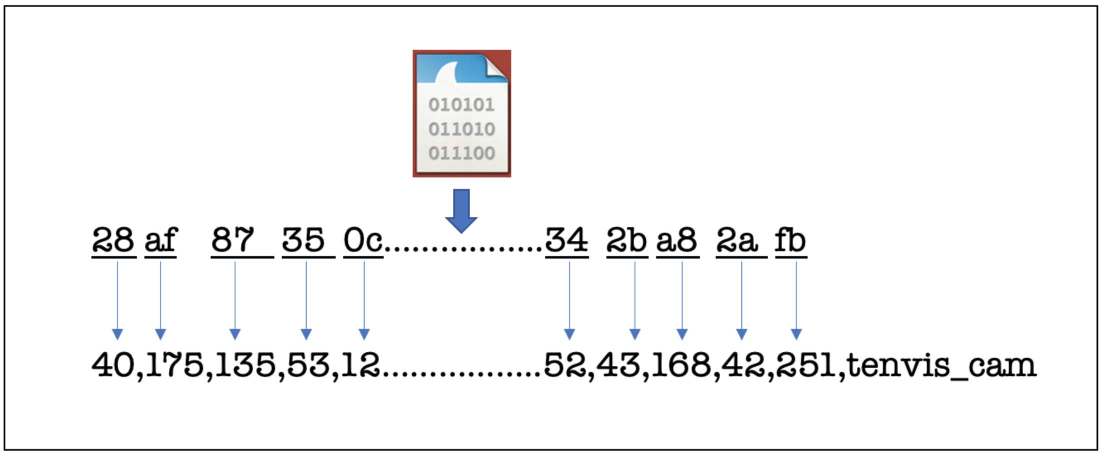

010101
011010
011100
28 af 87 35 0c..................34 2b a8 2a fb
40,175,135,53,12..................52,43,168,42,251,tenvis_cam

FIGURE 7.5: .pcap files are converted of to Nilsimsa hash digests and then to integer strings for processing by a classifier.

the context of this work the input layer is made up of 32 integers with values between 0 and 255. These values correspond to the 32 bytes which are output from the Nilsimsa hash. Next, we utilize an extremely concise single hidden layer containing 100 hidden units. The outputs of each unit in the network are passed through the rectified linear unit (ReLU) activation function so that the model can learn a non-linear decision function. Lastly, the network outputs a 22 length vector where each value in the output vector represents a possible class correlating with our 22 sample devices. The *Softmax* function is applied to the output vector which translates output values to probabilities. The class with the highest probability is identified as the predicted class. The network weights are learned in a single epoch with the *Adam* optimizer and a constant learning rate of 0.001.

## 7.4 Results and Analysis

We test our method using a data set collected by Charyyev et al. [107], publicly available at https://github.com/netlab-stevens/LSIF/. The set contains twenty 24-hour traffic samples collected from 22 different IoT devices. Each sample consists of a file captured using *tcpdump*, and are stored in .pcap form. For our experiment these larger flows are also converted into shorter time segments using *tshark*, yielding descending samples ranging in time slices from 10-minute down to 1-minute, and finally a sample of the original 24 hour data set at a per-packet level; in other words, one .pcap file for each packet. Next, each of these .pcap files is processed into a 32-length feature vector using Nilsimsa as described above. A final field representing the name of the device that produced it (the *class*) is appended to the vector, as seen in Figure 7.5. These are categorized by time slice and are combined, producing a single file of strings for each time slice from all 22 devices for processing by the MLP.

In each of our experiments the model is trained in a balanced 5-fold cross validation scheme where the data set is split into 5 equal segments. Our MLP is randomly initialized and trained from scratch 5 times. In each iteration, 4 of the segments are used for training (80%) and the remaining segment (20%) is used for evaluating the model. Here, balanced means that each fold contains an equal, but random,

TABLE 7.1: Model performance on cleaned per-packet data set

| Device | Accur. | F1 | Sample wt. |
|---|---|---|---|
| oossxx SmartPlug | 1.00 | 1.00 | 0.071 |
| Lumiman 600 Bulb | 0.99 | 0.99 | 0.071 |
| Lumiman 900 Bulb | 0.99 | 0.99 | 0.071 |
| Gosuna Bulb | 0.99 | 0.99 | 0.070 |
| Smart Lamp | 0.99 | 0.99 | 0.070 |
| OceanDigital Radio | 0.99 | 0.99 | 0.039 |
| Smart Light Strip | 0.99 | 0.99 | 0.071 |
| Gosuna Socket | 0.99 | 0.99 | 0.071 |
| Renpho SmartPlug | 0.99 | 0.99 | 0.071 |
| Lumiman SmartPlug | 0.99 | 0.99 | 0.071 |
| Goumia Coffee Pot | 0.98 | 0.99 | 0.012 |
| Wans Cam | 0.98 | 0.99 | 0.222 |
| D-Link 936L Cam | 0.97 | 0.98 | 0.015 |
| Minger Light Strip | 0.77 | 0.87 | 0.019 |
| LaCrosse Alarm Clock | 0.76 | 0.87 | 0.017 |
| itTiot Cam | 0.76 | 0.87 | 0.016 |
| Tenvis Cam | 0.61 | 0.76 | 0.008 |
| tp-link SmartPlug | 0.53 | 0.70 | 0.003 |
| Chime Doorbell | 0.34 | 0.51 | 0.007 |
| Ring Doorbell | 0.28 | 0.44 | 0.0007 |
| Wemo SmartPlug | 0.21 | 0.35 | 0.003 |
| tp-link Bulb | 0.09 | 0.16 | 0.001 |

representation of each class. This training regiment greatly reduces the likelihood that our results are the product of a beneficial random selection of data points or initial network weights.

Performance results for the per-packet time slice by device are shown in Table 7.1. The table displays device name, weighted accuracy score, weighted f1 score and sample weight. Here, sample weight is the percent contribution of each device to the total number of samples. Important findings in this table are as follows. First, the Smart Recon method exceeds our expectations with 13 out of 22 devices classified with over 98% accuracy. Second, the devices which perform poorly are those with less than 15,585, or one percent, of samples available. Third, the OceanRadio and Goumia Coffee Pot appear to be exceptions to the *limited data* problem. This suggests that the devices have significantly different transmission patterns than the others in the data set.

In Figure 7.6 and 7.7 accuracy and f1 scores for each time slice are visualized across all time slices respectively. Upon inspection it appears that these performance results show an *unusual trend*, with 10-minute flows performing very well, but with degrading results as flow size decreases to 1-minute. Finally, at the per-packet level performance spikes sharply upward to around 98%. We will discuss this unusual trend further later in the paper.

To eliminate any possible sample bias when analyzing the various traffic flows, all of the input data is replicated in its entirety with all IP and MAC addresses generalized as either ***1.1.1.1*** or ***11:11:11:11:11:11***. In this way it can be shown that the model is not learning to identify trends based on device-specific or flow-specific attributes. Average accuracy and F1 score for both uncleaned (non-anonymized) and cleaned (anonymized) data can be seen in Figure 7.6 and 7.7. We see there is some bias present as the cleaned data set yields an accuracy of 96.9% and F1 score of 96.7%, slightly lower than the uncleaned with an accuracy of 98%

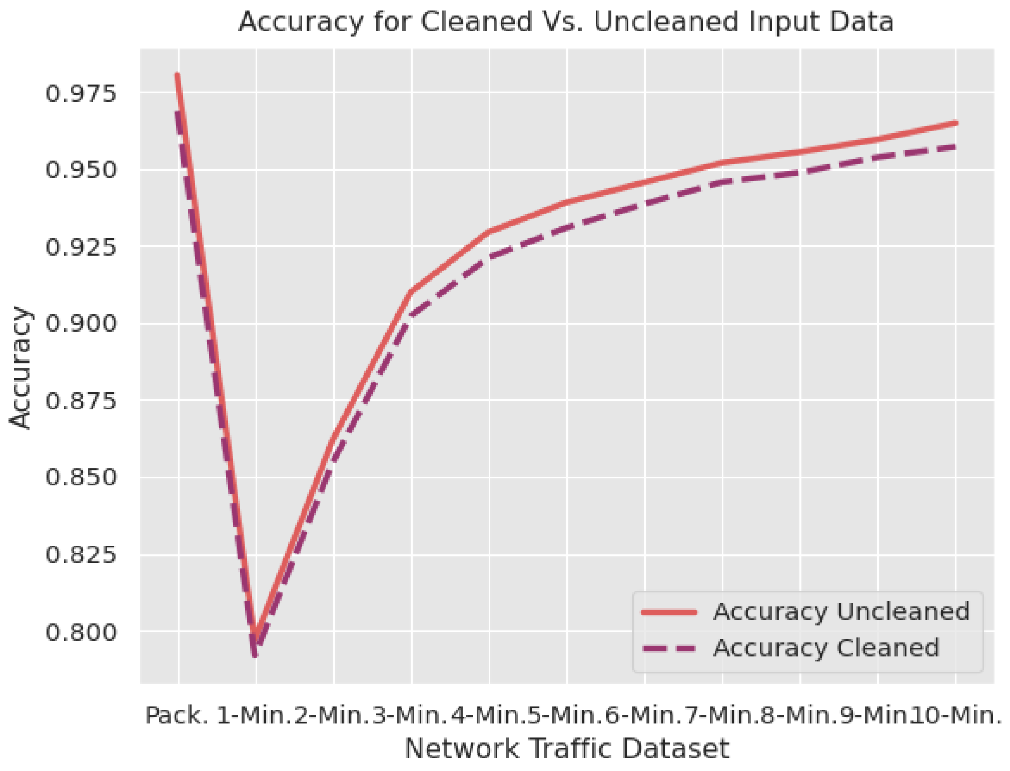


Figure 7.6: Accuracy for all time slices (10-minute through 1-minute and per-packet) in the cleaned and uncleaned data. These metrics were computed using a balanced 5-Fold Cross Validation technique. Here, balanced means that each device has equal representation in each fold.

and F1 score of 98%. Note this bias has a negligible effect on the *unusual trend*. As bias seems to play little role in this, we consider the possibility there could be some imbalance in the number of samples representing each device possibly contributing to this *unusual trend*.

To understand the *unusual trend* one needs to understand two important pieces of information: how these metrics are calculated, and how the percentage of samples differs between the minute/multi-minute time slices and the per-packet time slice.

There are two steps in computing the metric scores. First, metrics are calculated for each label, and their averages are weighted by support (the number of instances for each label divided by the total number of samples across all devices). Second, the mean score is computed across devices. Simply put, the performance of devices with a relatively high number of samples will strongly contribute to the final score of a given time slice. Weighted averaging is effective for computing metrics because it accounts for label imbalance, giving a more accurate value for assessing the actual performance of the method. This is shown mathematically below, where $D$ is the number of devices, $S_D$ is the number of samples for the current device, $M_D$ is the metric score for the current device and $N$ is the total number of samples in the current time slice.

$$\frac{\Sigma_0^D (M_D * \frac{S_D}{N})}{D}$$

In regard to the differences between time slices, the nature of the Nilsimsa hash

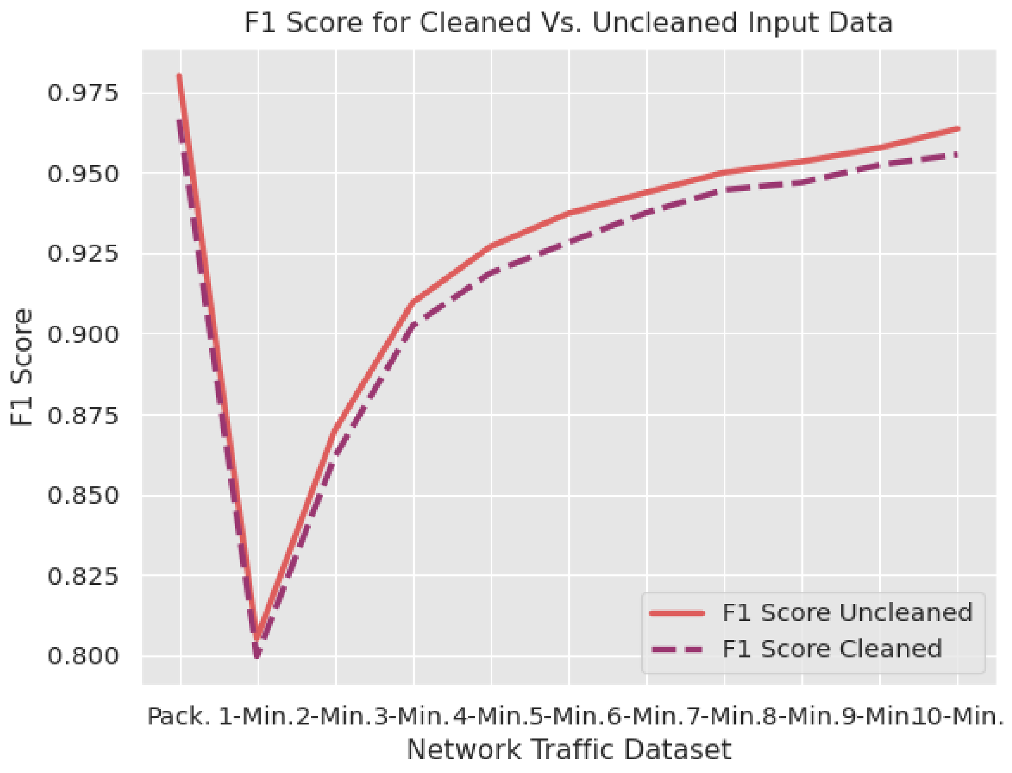


FIGURE 7.7: F1 Score for all time slices (10-minute through 1-minute and per-packet) in the cleaned and uncleaned data. These metrics were computed using a balanced 5-Fold Cross Validation technique. Here, balanced means that each device has equal representation in each fold.

and how it is applied to the data set causes all flows to have the same percentage representation for each device, with an increasing number of samples as the time slices become shorter. It should be noted that while each time segment may contain varying numbers of packets, it will still produce a single hash. If, therefore, a time segment contains very little data, it will still produce a hash which when measured in an un-weighted manner can produce deceptive performance results if the device appears to be performing poorly. This is seen in the decreasing performance as the time slices shrink from 10-minute to 1-minute and partially accounts for the *unusual trend*.

While there is some variance in the amount of traffic generated by the different devices, we find that with one exception, all devices contribute about 4% of the samples in each data set between 10-minute and 1-minute. We see that the OceanRadio device contributes only 0.05% of the data, yet still performs very well despite the lack of traffic available. This indicates that imbalance in samples size is not solely responsible for the *unusual trend*.

We therefore consider the possibility that some devices may be performing more poorly overall than others which might affect the *unusual trend* by bringing down the average performance. To explore this we analyze per device statistics across all time slices, and discover the Ring Doorbell and the tp-link Bulb are often misidentified. In many cases our model wrongly predicts these two devices to be other devices, largely because they contribute considerably less data overall. To see what effect this is having on the observe *unusual trend* we re-run the model without these devices as seen in Figure 7.8.

We see that when these two devices are removed, overall accuracy and F1 score improve for the time slices between 10-minutes and 1-minute, indicating that their poor performance is bringing down the overall average performance. However the removal has no real effect on the per-packet performance. Because the percentage

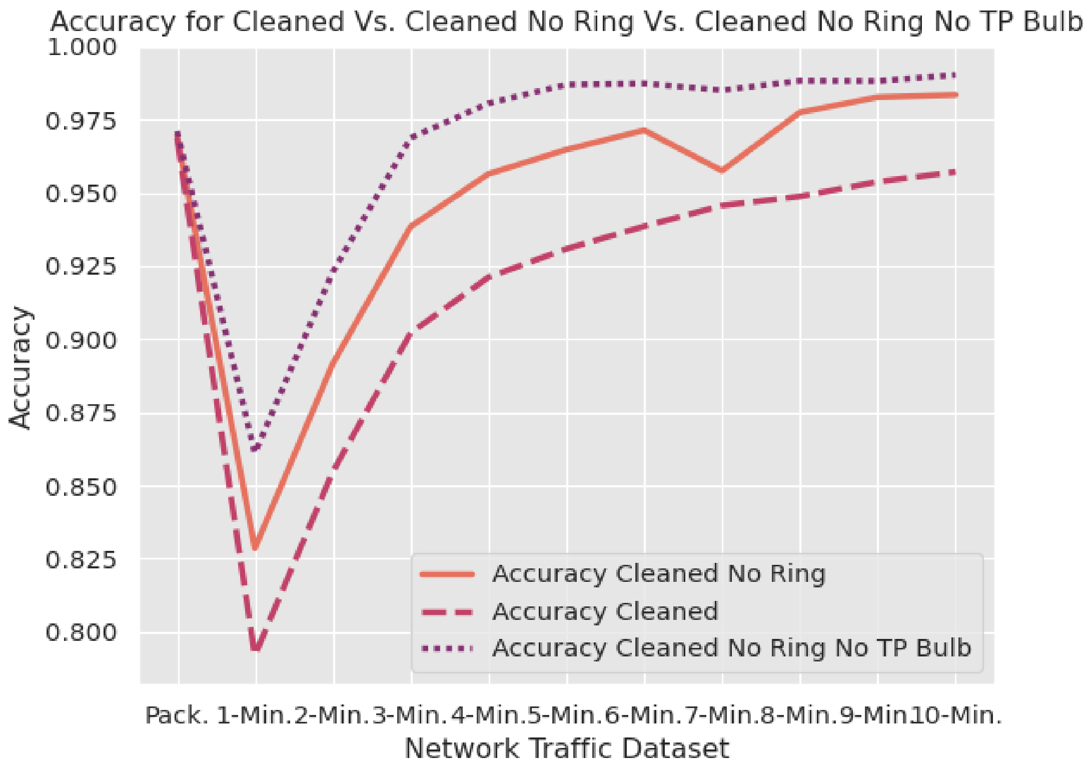


FIGURE 7.8: Model performance excluding results from Ring Doorbell and tp-link Bulb (devices with poorest results)

representation for each device in the 10-minute to 1-minute time slices is equal these poorly performing devices skew the overall average. When we reach the per-packet flows, performance depends on how many packets were sent in the 24-hour time period, giving a more accurate *weighted* view of the average performance. Since these devices transmitted significantly fewer packets, there is less effect on the overall average performance.

In order to better understand this behavior, we generate visualizations to represent the 32-dimensional data in two dimensions using Uniform Manifold Approximation and Projection (UMAP) [120], a tool for dimensional reduction and visualization. UMAP reduces the dimensionality of data while maintaining spacial relationships, allowing a two-dimensional view of clustering. Visualizations of the data are generated at the 10-minute, 5-minute, and 1-minute flows to demonstrate the effect of poorly performing devices on the overall average. Two devices, Ring Doorbell

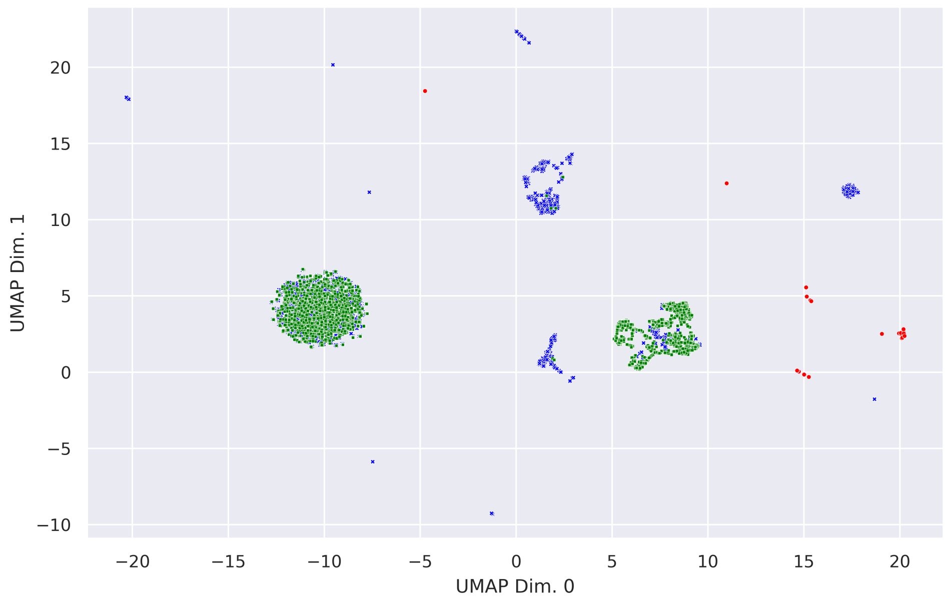


FIGURE 7.9: 10-minute flow clustering. The blue and green classes respectively represent the ring doorbell and TP Link Light Bulb. At 10 minutes enough unique behavior is captured to create clustering, which helps the MLP to identify individual samples.

and tp-link Bulb perform poorly, while OceanRadio performs quite well. This is despite the fact that all three contribute relatively small amounts of data as seen in Figures 7.9, 7.10, and 7.11.

We see the clusters for the poorly performing devices degrading as the time slices become shorter, explaining the downward trend in average performance (the blue dots are almost completely obscured by the green dots in Figure 7.11). As the sample sizes are evenly distributed, these devices (along with the others that perform poorly) drag down the overall average. Per-packet performance is less affected as these devices also contribute less overall traffic as mentioned above.

Because of the unusual trend noted it may be assumed there is an apparent weakness in the data set as some devices are contributing small amounts of data for analysis. In addition, some devices (Ring Doorbell and tp-link Bulb) are functioning in a way that cause them to appear similar to all other devices and to one another which is certainly cause for further investigation into their actual network activity. In fact, with these devices removed performance is nearly unchanged. However, to gain more accurate results future data collection should consider why some devices are contributing less traffic and some technique for mitigating this should be applied. It does however serve to validate the results found at the per-packet level, which is consistently performing at a very high level allowing single-packet identification of devices, the first step in developing a system for better management and oversight of IoT connected to a network. Our proposed research explores the possibility of improving and expanding this fingerprinting technique to improve IoT device identification, and for the development of tools and methods to improve device monitoring and accounting capabilities.

The other factor to be considered when analyzing this data is that at this early

0.65

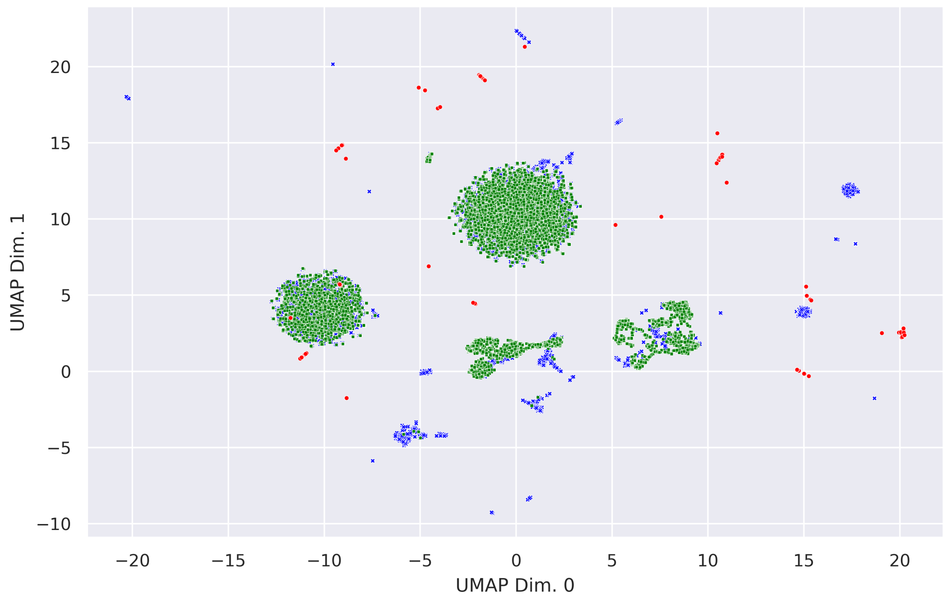


Figure 7.10: 5-minute flow clustering. The blue and green classes respectively represent the ring doorbell and TP Link Light Bulb. This figure shows the transition as the time slices become shorter, and individual samples become more difficult to identify.

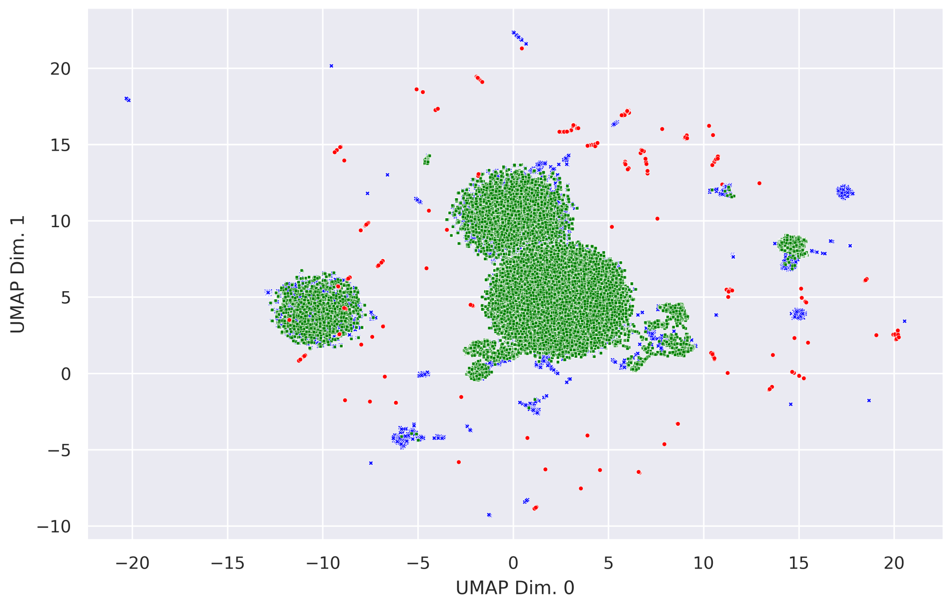


Figure 7.11: 1-minute flow clustering. The blue and green classes respectively represent the ring doorbell and TP Link Light Bulb. In this short time slice the blue dots are almost completely obscured by the green dots as a pattern of behavior is difficult to distinguish.

stage of our research, we were approaching the problem of IoT traffic fingerprinting in a manner similar to [107], where traffic was collected in time slices. For comparison based on similarity in the manner Nilsimsa was designed to be used this made sense; a single packet would never generate enough uniqueness to be distinct when compared to all other samples, there would be far too many possibilities for this approach to work. For that reason, a 10-minute time slice made sense in that over a period of 10 minutes some of the repetition in the activities of the IoT device in question could be captured. This would also explain why in [107] it was discovered that when time slices were reduced from 10-minutes to 1-minute performance degraded; less of the device behavior was being captured, and so was less distinct when compared to all other samples.

What was learned from further analysis is that with the use of machine learning in the form of a simple neural network, repetition at the per-packet level could be captured effectively. Packets from a particular device were repetitive enough even at this level that individual device behaviors could be captured and identified with a very high degree of accuracy (higher than the aforementioned study). This was a significant breakthrough as at the per-packet level a system could be implemented that would have the capacity to sniff and identify traffic in real time paving the way for a viable framework for system monitoring.

In this study we analyze the performance of our method on 22 heterogeneous devices and find it is capable of device identification with a hight degree of accuracy. As with most previous studies the data samples collected are from devices of different types, are from different manufacturers, or are different models from the same manufacturer. As real-time monitoring of networks to manage IoT device behavior would likely require the monitoring of many *identical* devices, we collect a data set from 3 groups of 8 identical devices to test performance in this scenario and find results to be poor. In the following chapter we give details on *hybrid locality sensitive hashing* along with changes to the machine learning approach to improve results and propose a framework for identifying devices in a much more realistic environment.

# Chapter 8

# Homogeneous Device Fingerprinting

## 8.1 Introduction

In this section we will discuss the continuation of our initial efforts, extending the fingerprinting of heterogeneous devices using LSH and a multi-layer perceptron algorithm to identifying individuals from 3 groups of 8 identical devices using hybrid locality sensitive hashing and an ensemble machine learning approach. While good results have been achieved by other studies seeking to utilize network traffic fingerprinting for device identification, they typically focus on *heterogeneous devices* in a laboratory environment devoid of the *background noises* present in live networks. Much of the previous research has relied on data sets provided by seminal studies such as IotSentinel [68], which are made up either of completely different (heterogeneous) devices or of different types of devices from the same manufacturer. In a realistic network setting, an administrator will likely need to ***track many devices of different types, but also multiple devices of***

***identical make and model (i.e. many identical web cameras) all producing very similar traffic. In addition, this traffic will be mixed in the presence of background traffic noise, possibly from other unknown or un-modeled IoT devices.*** In our previous work [121] we found that while our initial efforts were highly effective with heterogeneous devices they performed poorly against groups of identical (homogeneous) devices on the same network, leading us to this current work.

We present FlexHash, a novel system combining the data generalizing capability of novel hybrid locality-sensitive hashing with the power of machine learning to achieve a highly accurate network traffic fingerprinting model for identical device identification. FlexHash requires only a single packet from network traffic to achieve accurate results, and also performs well in noisy environments. It also achieves high accuracy in identifying devices from groups of identical devices, a challenging task for traditional fingerprinting methods as identical devices are expected to generate similar network traffic. FlexHash avoids the complexity of feature selection and engineering by hashing an entire packet and then converting the resultant digest directly into a feature vector. Subtle differences in the typically repetitive traffic produced by IoT devices are captured by hybrid similarity hashing, and can then be modeled with the help of machine learning algorithms to produce a device fingerprint. The ability to achieve high accuracy using only single packets for device identification will allow for system monitoring by periodically sampling traffic to detect device membership, device genre, the presence of unknown devices, and changes in device behavior indicating traffic anomalies and possible compromise.

Contributions of this work are as follows:

- We develop FlexHash, a novel locality-sensitive hashing algorithm that enables adjustments to the hashing parameters (accumulator length, window size, and n-grams).

- We implement a network traffic fingerprinting method combining FlexHash with machine learning, and perform accurate IoT device identification requiring only a *single packet* of network traffic.
- We evaluate this system by classifying device genre and *identical devices* in the presence of identical peers while also including realistic *background noise.*
- We collect traffic data from three categories of 8 identical IoT devices, which we share with the research community at: *https://github.com/UNR-IoT-Fingerprinting/FlexHash.*

### 8.1.1 Difference of Our Study from Previous Works

Non-ML approaches offer the advantage of simplicity, but often require larger traffic samples for comparison, and potentially large databases which must be searched to find matching device profiles. Machine learning solutions offers the advantage of highly accurate identification with smaller traffic flows and lend themselves to real-time monitoring which is more difficult when a database is required for comparisons, when larger flows are required, or when rules must be applied for identification. While effective, machine learning approaches rely on feature extraction, selection, and engineering, a process that can be computationally expensive and typically requires a high degree of domain knowledge both in machine learning and networking to select appropriate and relevant features for a given algorithm. FlexHash takes advantage of both approaches by combining a hybrid similarity hashing technique with machine learning, giving it the strength and real-time monitoring capability using only a single packet of network traffic for highly accurate identification, but requires no feature extraction or engineering.

Overall our work is different from existing studies in the following aspects:

- Our method does not depend on feature selection from the data which is computationally expensive and requires domain knowledge.
- We mainly focus on fingerprinting homogenous (identical) devices rather than heterogeneous (devices with different manufacturers and types) devices. Note that the identification of homogenous devices is a complex problem compared to the identification of heterogeneous devices as similar devices are expected to have similar traffic characteristics.
- Our approach works when there exists traffic noise, which is an important aspect as in real-world deployments the traffic noise is unavoidable.
- Finally our method can be tuned to fit various networks with different network characteristics.

Our work addresses some of the main challenges in traffic fingerprinting such as identification of homogenous devices, identification in noisy environments, and feature selection from the network traffic data.

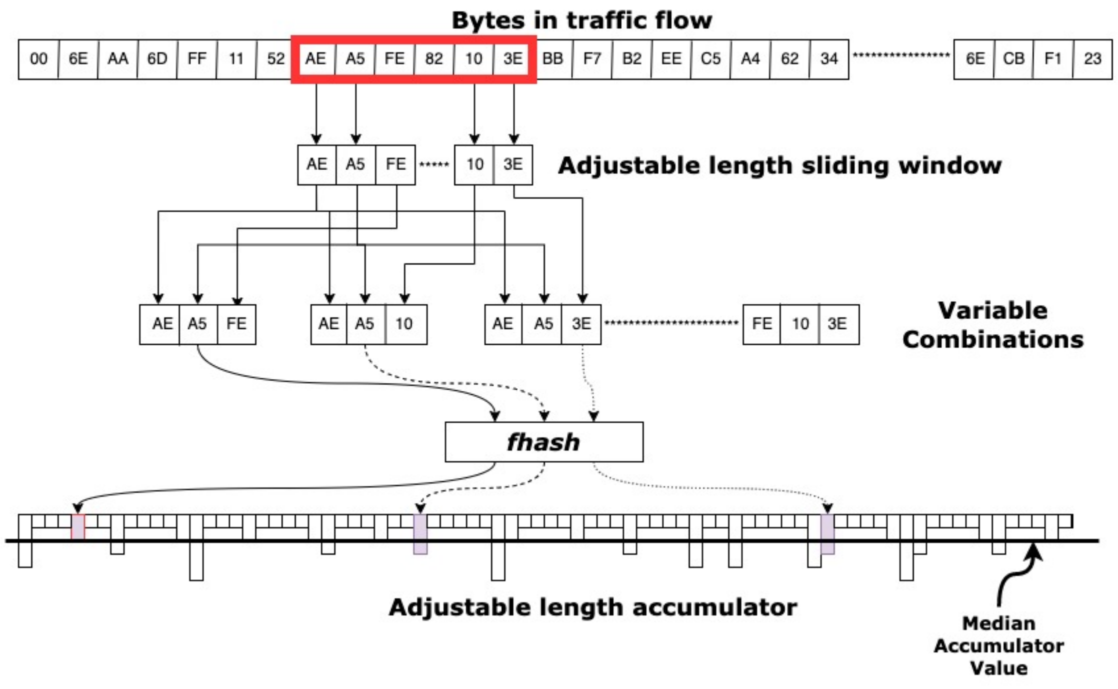


FIGURE 8.1: FlexHash functionality with tunable parameters

## 8.2 Hybrid LSH

To achieve identification of identical devices we develop a novel version of locality sensitive hashing based on the approach taken by other n-gram locality sensitive hashing schemes. It is novel in that it does not seek to measure similarity, but uses the same approach to reduce high dimensional data, namely *.pcap* files using similarity hashing, but then converts the resultant digest into a base-10 feature vector for use with a machine learning algorithm. In addition, the static parameters seen in examples such as *Nilsimsa* are made to be adjustable with the goal of amplifying the distinguishing characteristics of the hashes to learn individual behaviors being carried out by otherwise identical devices as they produce network traffic. This hashed traffic becomes a unique fingerprint, and is identified through machine learning such that a device can reliably be identified using just a single packet. We write this hybrid LSH scheme in Python and share it publicly at: *https://github.com/UNR-IoT-Fingerprinting/FlexHash*

### 8.2.1 Tunable Parameters

There are implementations of the Nilsimsa hashing algorithm in various languages, but most function in the same way that the original algorithm was designed, and these parameters were optimized for its originally intended use as a spam email detection tool. There are three main components of the algorithm that we have found useful to adjust in order to bring out more detail in Iot device traffic hashes; the sliding window, the process of producing n-grams from the data found in the window at each step, and the length of the accumulator that the data is mapped to. Figure 8.1 gives an overview of the entire process. In the following we consider each of these parameters in detail.

#### 8.2.1.1 Sliding Window

As with most n-gram based hashing algorithms, FlexHash utilizes a shingling approach wherein data is read in bytes based on a sliding window of length $n$. For Nilsimsa, $n$=5, a parameter that was chosen for efficiency in identifying spam email. FlexHash, which is implemented in Python, allows for a variable length to be set for $n$, allowing different characteristics to be captured based on the traffic patterns and behavior of the device in question. Which window length is best varies, as devices repetitively send unique bursts of traffic in different packet formats. By increasing or decreasing the window size, the margins between packets, or between segments of a packet can be blended, or generalized, to a greater or lesser degree which allows the hash to capture unique behavior by the device.

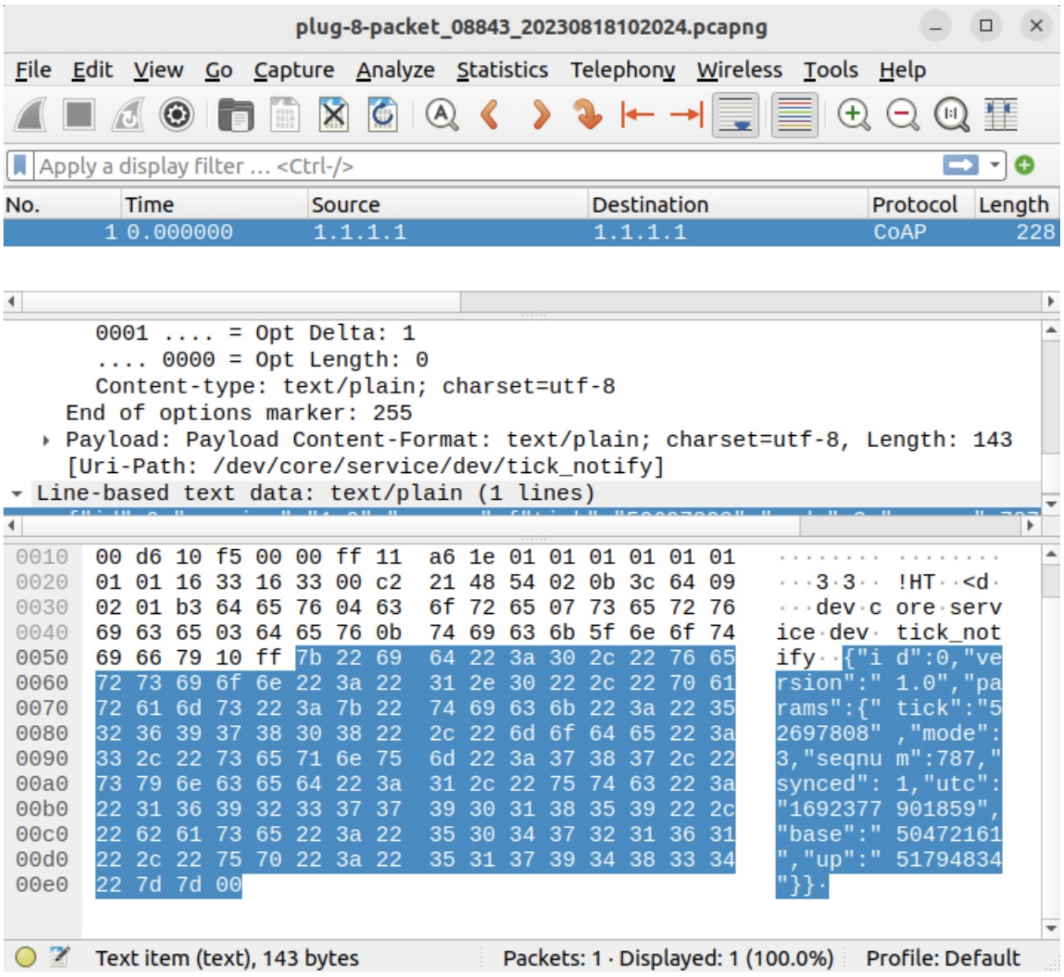
Figure 8.2 is a Wireshark representation of a random packet generated by a smart plug. We know the packet contains information such as protocols used at each layer (application, transport, IP, and network), header information, ect. and finally payload content. Each of these is accompanied with a variety of metadata specific to each layer. This information is seen as bytes in the figure, and the bytes are generalized by the sliding window. Repetitions or uniqueness of these bytes can be amplified or diminished by changing the window size. Which direction (longer or shorter) is optimal depends on the device. To determine this and the other parameters that provide the best results for a given device, we randomly choose several settings then measure performance of the classifier to find the settings that work best. Adjustable parameters and the pre-chosen values for each can be seen in Figure 8.1.

It can also be seen in Figure 8.2 that all source and destination IP addresses and MAC addresses have been generalized to either `1.1.1.1` or `11.11.11.11.11.11` respectively. Once these have been altered, header checksums are recalculated to

TABLE 8.1: Tunable Parameters of FlexHash

| | |
|---|---|
| Accumulator Size (in bits) | 128, 256, 512, 1024 |
| Window Size | 4, 5, 6 |
| Combination Size | [2: Window Size] |

avoid unexpected bias based on these unique and easily spoofed characteristics. However, the highlighted portion of the hexadecimal and ASCII representations of the packet, the packet payload, still contain some device specific information. We believe this may be key to uniquely identifying packets as belonging to a specific device. As devices talk to one another in a network, or as they send traffic back to the manufacturer for various reasons, they include specific information about


plug-8-packet_08843_20230818102024.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.   Time       Source    Destination   Protocol  Length
    1 0.000000   1.1.1.1   1.1.1.1       CoAP      228
```

```
      0001 .... = Opt Delta: 1
      .... 0000 = Opt Length: 0
      Content-type: text/plain; charset=utf-8
    End of options marker: 255
  ▸ Payload: Payload Content-Format: text/plain; charset=utf-8, Length: 143
    [Uri-Path: /dev/core/service/dev/tick_notify]
▾ Line-based text data: text/plain (1 lines)
```

```
0010  00 d6 10 f5 00 00 ff 11  a6 1e 01 01 01 01 01 01   ········ ········
0020  01 01 16 33 16 33 00 c2  21 48 54 02 0b 3c 64 09   ···3·3·· !HT··<d·
0030  02 01 b3 64 65 76 04 63  6f 72 65 07 73 65 72 76   ···dev·c ore·serv
0040  69 63 65 03 64 65 76 0b  74 69 63 6b 5f 6e 6f 74   ice·dev· tick_not
0050  69 66 79 10 ff 7b 22 69  64 22 3a 30 2c 22 76 65   ify··{"i d":0,"ve
0060  72 73 69 6f 6e 22 3a 22  31 2e 30 22 2c 22 70 61   rsion":" 1.0","pa
0070  72 61 6d 73 22 3a 7b 22  74 69 63 6b 22 3a 22 35   rams":{" tick":"5
0080  32 36 39 37 38 30 38 22  2c 22 6d 6f 64 65 22 3a   2697808" ,"mode":
0090  33 2c 22 73 65 71 6e 75  6d 22 3a 37 38 37 2c 22   3,"seqnu m":787,"
00a0  73 79 6e 63 65 64 22 3a  31 2c 22 75 74 63 22 3a   synced": 1,"utc":
00b0  22 31 36 39 32 33 37 37  39 30 31 38 35 39 22 2c   "1692377 901859",
00c0  22 62 61 73 65 22 3a 22  35 30 34 37 32 31 36 31   "base":" 50472161
00d0  22 2c 22 75 70 22 3a 22  35 31 37 39 34 38 33 34   ","up":" 51794834
00e0  22 7d 7d 00                                        "}}·
```

```
Text item (text), 143 bytes    Packets: 1 · Displayed: 1 (100.0%)    Profile: Default
```

FIGURE 8.2: Wireshark representation of a single *cleaned* packet (.pcap) including the hexadecimal representation of the packet contents which are processed by FlexHash.

themselves, their firmware, their current location, ect. that cannot be erased or spoofed in the packet, and that is sent consistently. Even if this data is encrypted, it will still produce a unique hash based on the packet structure and may be one of the reasons a classifier is able to single out individual devices based on their network activity. This is significant as most previous studies focus on packet structure, header details, ect. but generally igonore the actual payload of the packet. This is also important for window size, in that based on how it is set, it will capture more or less of this information as *shingles* that affect the final hash.

#### 8.2.1.2 n-gram Size

As the sliding window progresses byte by byte across the input data file, $n$ characters are represented in the window at each step. Nilsimsa, using a window of size 5, calculates all possible tri-grams (more specifically, all possible combinations) from these characters before advancing. So for 5 bytes, there would be 10 possible combinations of these bytes. To expand the possibilities, FlexHash allows for a variable number of combinations from 2 to the max size of the window; so for a window of 7, there could be combinations of 2-7. The fhash algorithm (see *Algorithm 2*) is applied to each combination mapping it to a variable length accumulator. By applying all possible combination sizes, the degree to which the bytes in the window are further shingled is affected again capturing different details in the input data to help identify traffic as being unique to an individual device.

#### 8.2.1.3 Accumulator Length

Finally, FlexHash allows for a variable length accumulator. As combinations are mapped to the accumulator space, the cell are initially set to a value of 0, and as items are mapped to cells they are incremented by 1. Once all data is processed, the median value of all cells is calculated, then for each cell if the final value is less than or equal to this median the cell is given a value of 0, otherwise it is given a value of 1. This byte string is then converted to hexadecimal, then to binary values between 0-255 to be used as an input feature vector for a machine learning algorithm. Nilsimsa uses a set accumulator length of 256, but FlexHash allows for a variable length to capture unique characteristics in the input data. We note here that unlike the LSH approaches covered in Chapter 7, FlexHash does not provide a similarity score. Instead it uses the hashing function of LSH to produce a feature vector which is then passed to a classifier to produce a model for device identification.

#### 8.2.1.4 Visualization of FlexHash tunable parameters

In this section we demonstrate the strength of the FlexHash algorithm; hashing parameter tuning. As discussed above, our novel algorithm allows for the adjustment of variables that are pertinent to the hashing process (i.e. accumulator size, sliding window size, and combination size). The adjustment of these values allows FlexHash to capture more subtle aspects contained in the device network activity, producing hashes which are more effectively identified by the machine learning process when creating device traffic fingerprints.

To aid the reader in understanding differences in parameter settings we utilize two types of visual aids; confusion matrices and UMAP representations [122]. A confusion matrix is a tabular representation of data that illustrates the performance of a model by displaying the counts of true positive, true negative, false positive, and false negative predictions. This enables assessment of the model's accuracy and error rates across classes, and helps determine misidentification. UMAP is a nonlinear dimensionality reduction technique used for visualizing high-dimensional data in lower-dimensional space while preserving local and global structure. It focuses on finding a low-dimensional representation of data points while maintaining their high-dimensional relationships with one another.

To demonstrate the effect of parameter tuning, we analyze the samples produced with the best and worst hashing parameters in a set of 8 identical web cameras. Visualizations of these samples are found in Figure 8.3, 8.5. Note that the UMAP plots are generated with only the hashed data and have no exposure to the machine learning model. We begin by inspecting Figure 8.3 which reflects hashes generated from sub-optimal parameters. While cam-1 achieves high performance, the clusters of data points representing cameras 2-8 are overlapping, thus the machine learning models struggle to accurately predict to which device a sample belongs. In Figure 8.4 we see the result of these same parameters represented in a confusion matrix. While cam-1 is correctly identified 99.71% of the time, cams 2-8 are frequently misidentified as one of the other cameras.

In Figure 8.5 we see that by optimizing the parameter set used, FlexHash is able to hash devices into a representation which reliably separates data points originating from different devices into distinct spatial regions. The downstream effect of these parameters is shown in Figure 8.6, with all devices being classified correctly 96% or more of the time. Overall, locality-sensitive hashing removes the need for feature engineering in device fingerprinting, while tunable hash parameters enable adaptation of the hashing algorithm to various sets of IoT devices, including identical devices.

In the previous research on IoT device identification by network fingerprinting there are two key areas where there has been little work; sample data that includes the presence of background noise, and the identification of homogeneous (identical) devices. For a system to be useful in a live environment the inclusion of random and IoT-specific noise is essential as these will always be present. In the same way, network administrators will typically be required to manage groups of devices that are of the same manufacturer and even the same model, as in multiple sensors or cameras in a system. Similar to the works cited in our background study, our previous research [121] was effective in identifying heterogeneous devices, but was of limited use when facing homogeneous traffic. Through the tun-able parameters found in FlexHash characteristics in traffic produced by identical devices can be isolated and amplified by adjusting parameters such as accumulator length, ngram length, and combinations when hashing traffic. We believe this addresses a significant gap in the previous research.

### 8.2.2 fhash

At each position of the sliding window all combinations of the length set by FlexHash are generated. The function (fhash) multiplies the cumulative integer value of each combination and then adds a prime number to the result to facilitate greater hash distribution to the accumulator. The result is divided by the length of the accumulator and the remainder (modulo) becomes the current target index. For more detail please refer to Algorithm 1. To perform optimal parameter selection for a particular device, we run pre-chosen combinations of parameters and test which set provides the best results. Optimal parameters vary based on the device type, making the adjustable property of FlexHash vital to improving accuracy and other performance metrics, especially when identifying identical devices. Optimal parameters for devices found in our evaluations are *1024, 6, and 2* for accumulator, window, and combination size for smart plug and light bulb, and *1024, 4, and 2* for web cameras. As a future work, an automated process to predict a range of optimal parameters for a given device type would be desirable.

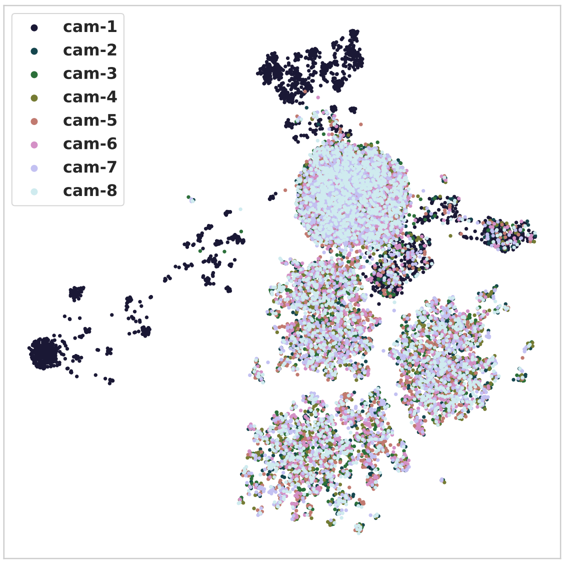
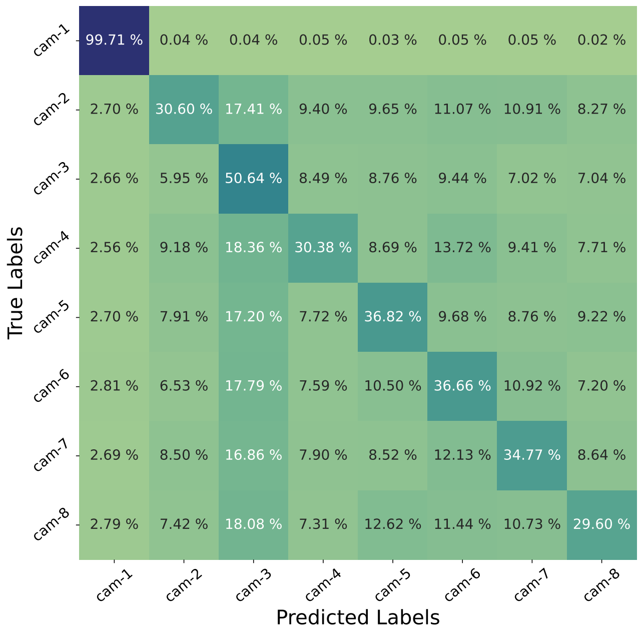


FIGURE 8.3: UMAP representation of cameras 1-8 to correlate with 8.4. We see well defined clusters only in cam-1.



| | cam-1 | cam-2 | cam-3 | cam-4 | cam-5 | cam-6 | cam-7 | cam-8 |
|---|---|---|---|---|---|---|---|---|
| cam-1 | 99.71 % | 0.04 % | 0.04 % | 0.05 % | 0.03 % | 0.05 % | 0.05 % | 0.02 % |
| cam-2 | 2.70 % | 30.60 % | 17.41 % | 9.40 % | 9.65 % | 11.07 % | 10.91 % | 8.27 % |
| cam-3 | 2.66 % | 5.95 % | 50.64 % | 8.49 % | 8.76 % | 9.44 % | 7.02 % | 7.04 % |
| cam-4 | 2.56 % | 9.18 % | 18.36 % | 30.38 % | 8.69 % | 13.72 % | 9.41 % | 7.71 % |
| cam-5 | 2.70 % | 7.91 % | 17.20 % | 7.72 % | 36.82 % | 9.68 % | 8.76 % | 9.22 % |
| cam-6 | 2.81 % | 6.53 % | 17.79 % | 7.59 % | 10.50 % | 36.66 % | 10.92 % | 7.20 % |
| cam-7 | 2.69 % | 8.50 % | 16.86 % | 7.90 % | 8.52 % | 12.13 % | 34.77 % | 8.64 % |
| cam-8 | 2.79 % | 7.42 % | 18.08 % | 7.31 % | 12.62 % | 11.44 % | 10.73 % | 29.60 % |

FIGURE 8.4: With window, combination, and accumulator of size 128, 6, 6 respectively, we see that cam-1 is correctly identified, but cam-2 through cam-8 are frequently misidentified.

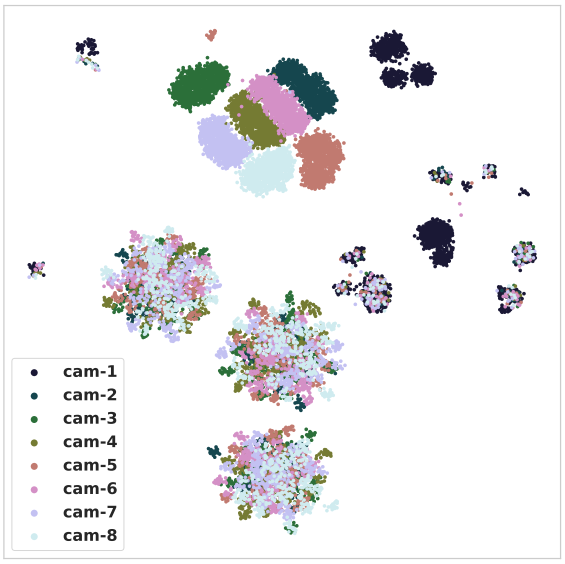
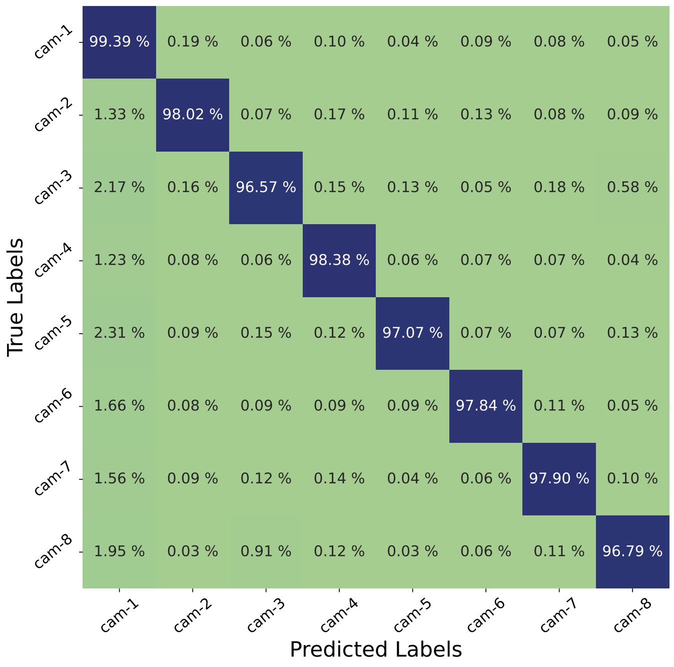


FIGURE 8.5: UMAP representation of cameras 1-8 to correlate with 8.6. In this image we see with optimal parameters all cameras are well clustered.



| | cam-1 | cam-2 | cam-3 | cam-4 | cam-5 | cam-6 | cam-7 | cam-8 |
|---|---|---|---|---|---|---|---|---|
| cam-1 | 99.39 % | 0.19 % | 0.06 % | 0.10 % | 0.04 % | 0.09 % | 0.08 % | 0.05 % |
| cam-2 | 1.33 % | 98.02 % | 0.07 % | 0.17 % | 0.11 % | 0.13 % | 0.08 % | 0.09 % |
| cam-3 | 2.17 % | 0.16 % | 96.57 % | 0.15 % | 0.13 % | 0.05 % | 0.18 % | 0.58 % |
| cam-4 | 1.23 % | 0.08 % | 0.06 % | 98.38 % | 0.06 % | 0.07 % | 0.07 % | 0.04 % |
| cam-5 | 2.31 % | 0.09 % | 0.15 % | 0.12 % | 97.07 % | 0.07 % | 0.07 % | 0.13 % |
| cam-6 | 1.66 % | 0.08 % | 0.09 % | 0.09 % | 0.09 % | 97.84 % | 0.11 % | 0.05 % |
| cam-7 | 1.56 % | 0.09 % | 0.12 % | 0.14 % | 0.04 % | 0.06 % | 97.90 % | 0.10 % |
| cam-8 | 1.95 % | 0.03 % | 0.91 % | 0.12 % | 0.03 % | 0.06 % | 0.11 % | 96.79 % |

FIGURE 8.6: With window, combination, and accumulator of size 1224, 4, 2 respectively, we see that all cams are correctly identified with a high degree of accuracy.

**Algorithm 1** fhash

```
1: $ngram\ list \leftarrow$ current ngrams
2: $total \leftarrow 1$
3: for each ngram do
4:     $total \leftarrow total * ngram$
5: end for
6: $result \leftarrow total + 3$
7: $result \leftarrow result\ \%$ accumulator length
8: return $result$
```

### 8.2.3 Hashing traffic data

As mentioned earlier, FlexHash avoids the complexity of feature selection and instead applies the entirety of a traffic data in hashed form to produce a fingerprint. Earlier studies such as [6] [64] [79] [65] often work to capture network features from packet headers such as protocol, packet size, packet inter-arrival time, etc. to find distinct qualities with which to produce a recognizable uniqueness. What is overlooked in this approach is what lies below the surface, in the message body of the packet. We find that each packet produces traffic in which it identifies itself in various ways as it communicates with the device manufacturer, other devices, or other unknown IP addresses. During this activity devices often share their MAC address, current IP address, or other details that may specifically identify an individual device. Even if such identifying features are removed from the packet headers during processing to avoid bias in the machine learning algorithm (as they are in our study), information to a specific device may still appear in the encapsulated message body and can be captured by a hash regardless of encryption, etc. This is helpful when fingerprinting with LSH because the hash seeks to identify not specific information, but similarity to other packets produced by the same device.

**Algorithm 2** FlexHash

```
$x \leftarrow$ window size
$y \leftarrow$ ngram size
$z \leftarrow$ accumulator size
procedure MAKEHASH($x, y, z$)
    for .pcap in file do
        $counter \leftarrow 0$
        for bytes in .pcap do
            $current \leftarrow$ window[counter:counter+x]
            $ngramlist \leftarrow current$
            procedure FHASH($ngramlist$)
                return $index$
                accumulator[index += 1
            end procedure
        end for
    end for
    $total \leftarrow 0$
    while index < z do
        $total \leftarrow total +$ accumulator[index]
        $median \leftarrow total / z$
        for accumulator[x]=0; x < z; x++ do
            if accumulator[index] ¡ median then
                accumulator[index] $\leftarrow 0$
            else if accumulator[index] ¿ median then
                accumulator[index] $\leftarrow 1$
            end if
        end for
    end while
    $vector = []$
    $counter \leftarrow 0$
    for bits in accumulator do
        $current \leftarrow$ accumulator[counter:counter+8]
        $current \leftarrow$ int(current)
        vector.append $\leftarrow current$
        $counter \leftarrow counter + 8$
    end for
end procedure
```

## 8.3 Methodology

For evaluation we focus on three categories of devices; smart plugs (*Ghome Smart Plug*), smart light bulbs (*General Electric CYNC Full-Color Smart Bulb*), and web cameras (*YI 1080p Home Camera*). Each category of device contains ***8 identical devices*** for a total of 24 devices as seen in Figure 8.7. Devices are connected to the network and allowed to complete their initial setup phase and then traffic data is collected on a resting state for 24 hours. Data is then sanitized and header checksums are recalculated using *editcap*, replacing all MAC and IP addresses of the devices with 11:11:11:11:11:11 and 1.1.1.1 respectively to avoid bias introduced by unique addresses. Digests of each packet for each parameter combination are generated and converted to feature vectors as described in prior sections. For evaluation, the data was split into 80% for training and validation, and 20% for testing. Results are measured in terms of precision (i.e., $TP/(TP+FP)$), recall (i.e., $TP/(TP+FN)$), f1-score (i.e., $2/(1/precision+1/recall)$), and accuracy (i.e., $(TP+TN)/(TP+FP+TN+FN)$) where TP, TN, FP, and FN stand for true positive, true negative, false positive and false negative.

### 8.3.1 Identification of IoT Devices

The overview of the IoT device identification system is presented in Figure 8.8. Traffic data is captured from IoT devices in a network and hashes of the traffic data are generated with FlexHash. These hashes are converted to feature vectors by converting each byte value in the resultant digest to base-10 numerical values ranging from 0-255. For instance, the byte value "FB" would be converted to the integer value 251. These feature vectors are then passed to our device identification system to train the underlying machine-learning model. Once a model is trained

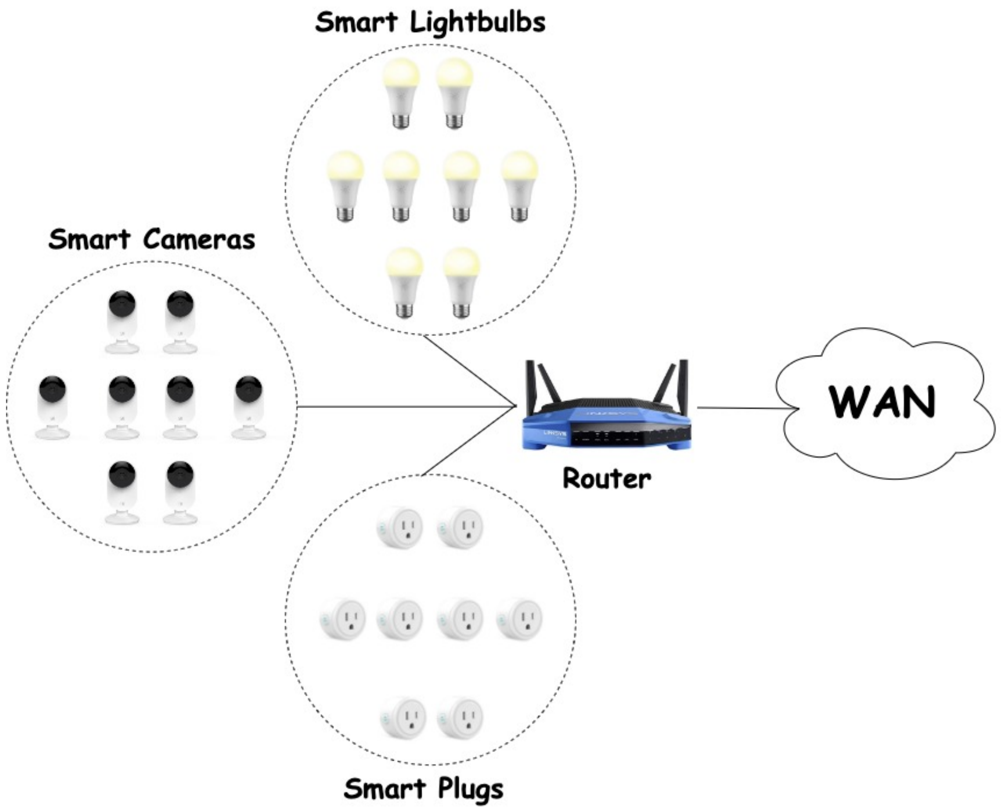


FIGURE 8.7: Traffic data was collected from 3 categories of devices, representing simple to complex set of devices. Each set contains 8 identical devices. Data is collected for 24 hours.

the identification system is ready for deployment on a router or micro-controller to serve as a gateway to a network. The device identification system will continuously monitor traffic data by capturing periodic samples to identify known devices, new devices joining the network, and to identify changes indicating anomalous behavior. This is done by generating the digest of randomly captured traffic data in the form of packets, converting them to numeric feature vectors, and passing them to a pre-trained machine learning classifier.

### 8.3.2 Ensemble Learning

As a machine learning framework we use Autogluon-Tabular (AGT) [123] from Autogluon version 1.0.0. AGT is an automatic machine learning framework designed specifically for tabular datasets, such as spreadsheets. AGT aims to simplify the application of machine learning techniques for practitioners and researchers. We elect to use this framework because it enables streamlining the use of various best practices and machine learning strategies that lead to superior performance. The components of AGT that make it an effective framework for our data are the use of diverse base model types and carefully designed ensembling strategies.

AGT makes use of various well-known methods as base models. Base models are the underlying learning strategies used by AGT for modeling the input data. Each of these models has the capacity to perform well independently, but differences in their behavior lead to variance in the patterns recognized. Ensembling predictions from models with different perceptions of the data space leads to robust

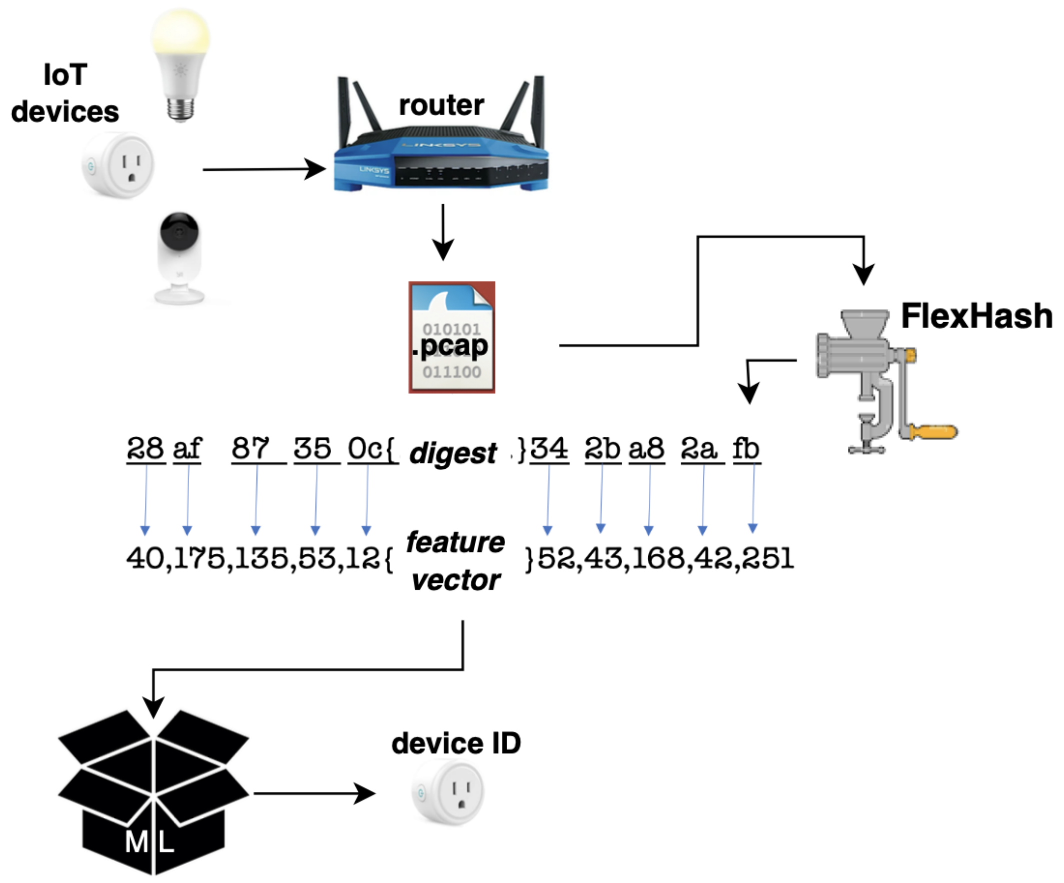

IoT devices
router
010101
.pcap
011100
FlexHash
28 af 87 35 0c{ digest }34 2b a8 2a fb
40,175,135,53,12{ feature vector }52,43,168,42,251
ML
device ID

FIGURE 8.8: The network traffic is continuously monitored by the device identification system. The traffic data is processed by FlexHash and converted into feature vectors. These feature vectors are given as input to the pre-trained ML model to identify the device that generated the traffic.

final predictions. Base models available are Random Forest, Extra Trees [124], CatBoost [125], LightGBM [126], XGBoost [127], Logistic Regression, K-Nearest Neighbors (KNN), and Neural Network. We select three base models to use in our experiments. These are Extra Trees, LightGBM, and XGBoost. These models were selected based on processing time and performance in preliminary experiments. Extra Trees, an ensemble learning method with additional randomness, builds multiple decision trees for predictions, potentially enhancing robustness at the cost of increased computational expense. LightGBM, a gradient boosting framework, prioritizes speed and efficiency with a tree-based learning approach, particularly suitable for large datasets but potentially more sensitive to overfitting. XGBoost, an optimized gradient boosting library, achieves exceptional performance through regularized learning and parallel computing, necessitating careful hyperparameter tuning.

Ensembling the predictions of multiple models leads to a reduction in the variance of a machine learning system and improved prediction accuracy [128]. AGT provides the implementation for two important forms of ensembling: repeated k-fold bagging and multi-layer stacking [129, 130]. Multi-layer stack ensembling is a strategy in which the output predictions of a preliminary layer of models are used as features or inputs to a subsequent layer. AGT's implementation also utilizes a form of skip connection which concatenates the original model features onto the preliminary layer's predictions. This technique allows the subsequent layers of models to have an understanding of the original data space in addition to the previous layers' predictions. Repeated k-fold bagging is a method that randomly splits the data into chunks and trains multiple models on different random chunks of the available data. When making a prediction, the input data is passed through each model and the outputs are voted on. Whichever output is most frequent is used as the output for the group. The final model produced by AGT is a combination of 68 total models in 2 layers. All bags contain 8 copies of the base model trained on different portions of the available data. The first layer (models trained on just input data) contains 2 Extra Trees models, 3 bags of LightGBM models with varying hyperparameters, and 1 bag of XGBoost models. The second layer is identical to the first, except models are trained with the output predictions of the first layer models concatenated to the original input data. AGT utilizes a method called ensemble selection [131], which takes predictions from all available models into account and produces a single output.

## 8.4 Experimental Analysis and Results

In this section, we analyze the performance of our device identification system using FlexHash with the following experiments.

- To demonstrate the system's ability to operate in a realistic network setting we consider the effect of adding background noise to each data set. To achieve this we combine device data with two types of noise: *IoT noise* (network traffic of random IoT devices), and *network noise* (random traffic from a live network).
- We generalize all devices into three categories: *smart plug*, *smart bulb*, or *web camera*. A model is built and packets from each device are classified by *genre*.
- We determine the source device of a network packet from a pool of 8 of the same model devices. We train a model, then identify packets from that group as belonging to a distinct individual.
- We compare individual device results to those from another popular LSH method, namely *Nilsimsa*.

TABLE 8.2: Average performance in identifying device genre in the presence of noise and without noise.

| | Accuracy | | F1 Score | |
|---|---|---|---|---|
| Device | Without Noise | With Noise | Without Noise | With Noise |
| Smart Plugs | 99.98 | 99.89 | 99.97 | 99.90 |
| Smart Lights | 99.88 | 99.41 | 99.92 | 99.57 |
| Smart Cameras | 99.99 | 99.98 | 99.99 | 99.98 |

### 8.4.1 Identify devices in the presence of background noise

In a live network traffic noise is inevitable, therefore it is crucial to explore system performance in a noisy environment. This demonstrates the ability to apply device identification techniques in a realistic network setting when monitoring for IoT device membership or when searching for unknown devices by category. While offering good results, many of the previous studies in this field have relied on artificially sanitary environments made up of heterogeneous devices to perform experimental analysis, but fail to show these systems could function in a live network. To achieve realistic results, we introduce two types of background noise: *IoT noise* (random IoT traffic from [68]) and *network noise* (random network traffic from a large set of heterogeneous devices. We create hashes of the noise data and add it to device data set labeled as either *network-noise* or *iot-noise*. Even tested in adverse conditions, results show noise has a minimal impact on performance demonstrating the FlexaHash systems ability to function in a live network environment.

We begin with results for experiments with and without noise which are presented in Tables 8.2, 8.3, and 8.4.

Table 8.3: Average performance in identifying identical devices from the same category in the presence of noise and without noise.

| | Accuracy | | F1 Score | |
|---|---|---|---|---|
| Device | Without Noise | With Noise | Without Noise | With Noise |
| Smart Plugs | 85.79 | 85.02 | 85.77 | 84.91 |
| Smart Bulbs | 93.63 | 89.09 | 93.60 | 84.75 |
| Web Cameras | 97.89 | 98.61 | 97.78 | 98.55 |

### 8.4.2 Identify devices by genre

The identification of devices by genre enables inferring the device type for traffic captured in real-time via a single packet. By monitoring a network and testing random packets we can predict the likelihood that a particular packet belongs to a genre, i.e. the packet is *probably* a camera, or *probably* a smart bulb, etc. To do this, all 24 devices are re-labeled as either smart bulb, smart plug, or web camera, and a multi-class classifier is built. The implication here is that in a network scenario, randomly captured packets identified as being generated from some IoT device can be further categorized as a specific device type. This is useful for investigating unknown or rogue devices on a network that could potentially compromise security. Results for this experiment are presented in Table 8.2. Results for identification by genre are above 99% for all devices both with and without background noise. We are also able to differentiate the noise type from known device genres, achieving an accuracy for *iot-noise* above 98% and for *network-noise* above 97%.

Table 8.4: Comparison of average performance, FlexHash vs Nilsimsa in identifying identical devices from the same category.

| | Accuracy | | F1 Score | |
|---|---|---|---|---|
| Device | FlexHash | Nilsimsa | FlexHash | Nilsimsa |
| Smart Plugs | **85.79** | 72.97 | **85.77** | 73.27 |
| Smart Bulbs | **93.63** | 78.04 | **93.60** | 80.48 |
| Web Cameras | **97.89** | 84.74 | **97.78** | 84.66 |

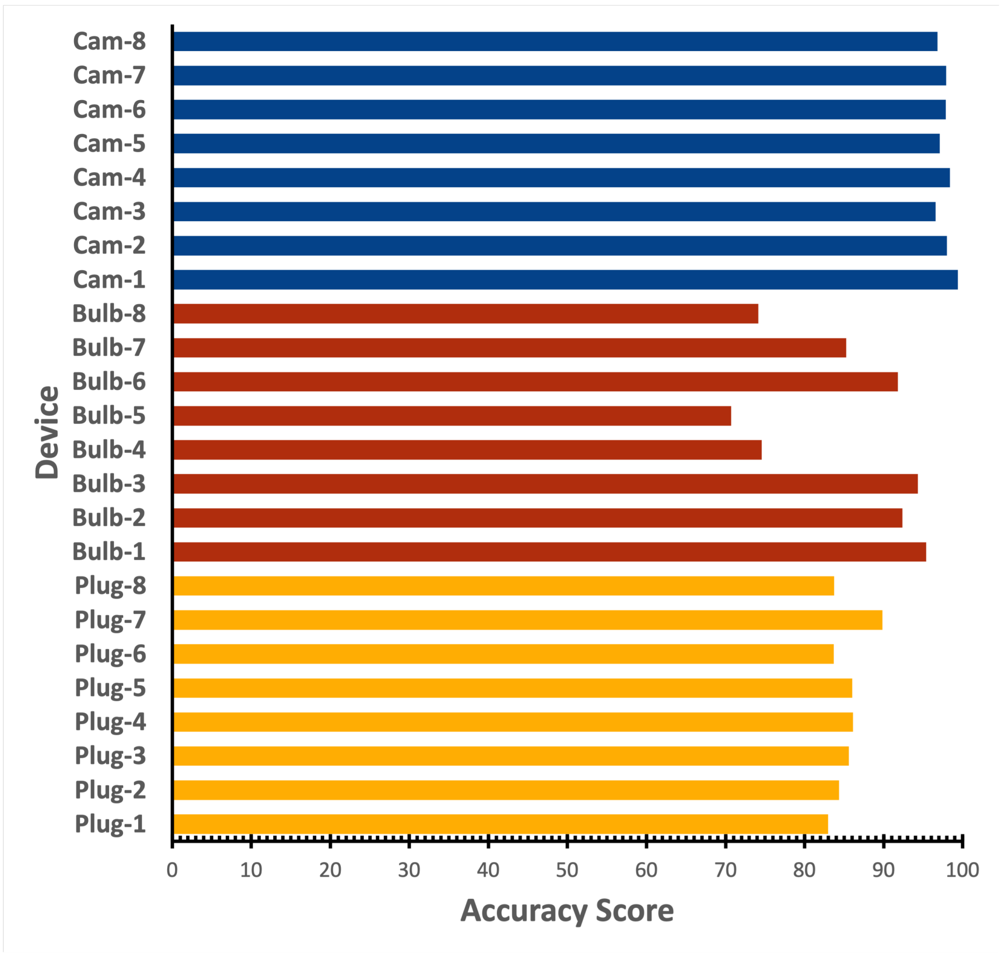


FIGURE 8.9: Results for identification of individual devices from a group of identical peers (plugs, lights, and cameras).

### 8.4.3 Identification of individual devices from identical peers

In this subsection, we work to identify an individual device from a group of identical peer devices. Instead of attempting to identify a heterogeneous set of unique devices, we address the more challenging task of identifying clusters of identical devices, a task under-explored in the literature. This type of identification is critical in tracking device behavior and membership as there are often multiple individuals of the same device appearing in a network. Experiments are performed both with and without background noise and results compared.

Average results in terms of both accuracy and F1 score for this experiment are shown in Table 8.3. We see that without background noise, web cameras achieve

Table 8.5: Comparison of average performance, FlexHash vs Nilsimsa in identifying identical devices from the same category.

| | Accuracy | | F1 Score | |
|---|---|---|---|---|
| Device | FlexHash | Nilsimsa | FlexHash | Nilsimsa |
| Smart Plugs | **85.79** | 72.97 | **85.77** | 73.27 |
| Smart Bulbs | **93.63** | 78.04 | **93.60** | 80.48 |
| Web Cameras | **97.89** | 84.74 | **97.78** | 84.66 |

an accuracy above 97% on average while smart bulbs and smart plugs achieve results above 85% and 93% respectively. We note that in the case of smart plugs, some imbalance was found in the data samples for a 24 hour period, with three of the eight plugs generating considerably more traffic than the other five, possibly accounting for this change in performance. Interestingly, devices with greater complexity appear to perform better than simpler devices when identifying an individual from identical peers. We note here that FlexHash and our device identification system achieve accuracy results in this more difficult scenario that are either competitive or superior to results offered by other approaches in the literature while using only a single packet sample. Results per device are shown in Figure 8.9. When running the same experiment with background noise, web cameras, smart bulbs, and smart plugs achieve an average accuracy of 98%, 89%, and 85% respectively. With noise added, we only see a slight degradation of performance in the smart bulbs, with web cameras and smart plugs performing at nearly an equal accuracy.

In Table 8.5 we compare the performance of FlexHash with another n-gram based LSH method, *Nilsimsa*. In our previous study, we observed that Nilsimsa outperforms other similar methods such as ssdeep [108], sdhash [109], and tlsh [110]. Thus, we can assume that better results with FlexHash over Nilsimsa will imply better results over other hashing techniques as well. Experiments are performed on network traffic data without background noise. FlexHash achieves an average accuracy of 97.74%, 93.63%, and 85.79% for web cameras, smart bulbs, and smart plugs respectively. On the same data, Nilsimsa achieves a lower performance of 84.74%, 78.04%, and 72.97%. This represents a percent increase of 13.00%, 15.59%, and 12.82%. We see here that in every case FlexHash achieves a significant increase in accuracy over Nilsimsa hashing, an indication of the effectiveness of FlexHash's tunable parameters.

# Chapter 9

# Conclusions and Future Work

## 9.1 Conclusion

There has been rapid growth around the concept of the Internet of Things with devices such as cameras, low-powered sensors, wearable technologies, and household and industrial sensors and actuators becoming commonplace on both public and private networks. These devices are often inter-connected to one another and to source sites and command-control servers on the Internet. With the average enterprise network consisting of 48% IoT devices and the average American household network hosting upwards of 20 devices, methods for tracking membership and detecting anomalous behavior are essential for providing adequate security. In this study we offer insights on current security issues and provide a novel approach for tracking IoT device activity based on network traffic fingerprinting. Our method demonstrates increased performance, provides a robust and sustainable method for monitoring, and requires only a single packet of network data to identify devices. In addition our method works in both heterogeneous and homogeneous device clusters, can identify unknown device by genre, and performs well in the presence of realistic background noise.

In Chapter 1 an introduction to the current landscape in IoT technology is provided, and we formulate a problem statement and detail our research approach. This is followed by Chapter 2 where a history of bots on the Internet is provided with particular attention given to recent issues with the Mirai botnet and its many variants. An interest in this type of malicious network activity provides a foundation for our research, and has lead to the various experiments and analyses we perform.

In Chapter 4 we begin by constructing a Smart City testbed utilizing production grade Iot in the form of a network connected programmable logic controller and several tangible systems to allow for both education and research. The Smart City provides hands-on access to the networking components and protocols commonly found in industrial IoT systems and proved to be highly useful in both training students in their use, as well as providing a vulnerable target on which to perform simulated attacks. This lead to further research on how best to protect modbus protocols and PLC-based systems, and perimeter security techniques were employed to prevent future attacks. This experiment was also useful in that it highlighted the weaknesses often found in industrial control systems and IoT in general.

In Chapter 5 a follow-up experiment in networking testbeds was performed with the implementation of an extensive software defined network for research and education. The SDN testbed provides an environment for students and new researchers to gain valuable hands-on experience with these cutting edge technologies, and provided a proving ground for experiments designed to aid with other studies relating to edge computing and sensor management. The testbed was made up of a complex network of both physical and virtual devices, including an SDN controller, edge devices and several honeypots. Chapter 6 discusses the collection and analysis of malicious traffic using a global network of honeypots for bot traffic analysis. A network of honeypots using the Cowrie honeypot software was established in Digital Ocean data centers around the globe, and a large body of attack traffic logs were amassed, providing material for analyzing and correlating attacks globally. It was determined from this study that a majority of malicious bot traffic occurring during the time of the study could be traced to Mirai and Mirai-variant botnets. This experiment lead to a closer analysis of IoT device security and monitoring techniques through device traffic fingerprinting.

In Chapter 7 we develop a method for creating a unique fingerprints for a group of heterogeneous IoT devices by combining locality sensitive hashing with a simple neural network. This approach is able to identify known devices based on a classifier model using a single packet of network traffic with very high accuracy. This novel approach opens the door for fingerprinting without the need for feature extraction or engineering, and can be used to identify network devices in real time by sampling single packets and comparing them to know devices. In addition, changes to known devices will indicate possible compromise providing a powerful tool for device monitoring. This work lead to several experiments in homogeneous device identification, genre identification, and performance in the presence of background noise which are taken up in Chapter 8. In this final work, a novel hybrid locality sensitive hashing algorithm is developed and programmed in Python, allowing for the adjustment of several parameters of the hashing algorithm. This allows for tuning of device hashes that lead to the ability to identify one device among several identical peers, an approach not yet done in the current literature. In addition, we used this method to identify both known and unknown devices by genre, and also added various types of background noise to demonstrate the method would be effective as a tool in a realistic network scenario where there would inevitably be a variety of interfering factors such as random, unidentified, and unknown traffic from IoT and non-IoT devices. This approach then becomes the foundation of a framework for device and network modeling for secure management.

## 9.2 Future Research Directions

There are several future study directions available in this work that were not pursued due to time constraints. We list these here for those interested in taking this study further, and happily invite them to do so.

### 9.2.1 Questions Regarding Tunable Parameters

In Section 8 we discuss the implementation of *tunable parameters* in the Flex-Hash algorithm. By adjusting the sliding window length, n-gram length, and the length of the accumulator we are able to optimize the resultant hashes for specific devices to increase the accuracy of our model when performing identification. We randomly select a set of parameters as outlined in the chapter and test every possible combination to see which work best. More work should be done here to explain why certain parameters work better than others, and what part of the hashed packets might be more or less affected by these adjustments. This would provide for a means to predict which parameters should be chosen, and would offer explain-ability for the improvement we see with this adjustment.

### 9.2.2 Further Explanation of the *fhash* Hashing Algorithm

In Section 8 pseudo-code for the *fhash* algorithm is given. This hashing algorithm maps each of the n-grams from a given position of the sliding window to an address in the accumulator. While effective in our case, we have given little explanation for the mathematical underpinnings of this algorithm which would strengthen this dissertation. In addition, more work could be done on this algorithm which may possibly improve performance further.

### 9.2.3 Device Anomaly Detection

To show the usefulness of FlexHash for anomaly detection it would be beneficial to take a sample of similar devices and infect some of them with a known malware, then hash traffic from devices and develop a binary classifier to determine if a device on the network was *healthy* or *infected*. We hypothesize that this would work very well with our approach as even the smallest change to the operating system on a device (i.e. a firmware update) has a significant effect on the accuracy when identifying devices. While this was a goal for our project, it proved more difficult than anticipated to infect a device with malware, and time constraints prevented the completion of the experiment.

### 9.2.4 Questions Regarding Importance of Packet Elements

To create feature vectors for our method, network packets are saved as individual .pcap files, and are then hashed. The question is, which part of the packet has the greatest effect on the outcome of the hash? Is the whole packet required for good accuracy, or is accuracy dependent on just certain parts of the packet, i.e. just the TCP headers, or just the payload, or some combination of the various elements available. To pursue this, packets could be broken down into their constituent parts and experiments could be run to determine which parts have the greatest impact.

### 9.2.5 Distribution Represented in UMAP

In Section 8 a UMAP representation of the spacial distribution of packets from identical devices is given before and after parameter tuning to demonstrate the effect of changing FlexHash parameters. The graph is made up of 8 identical web cameras, and it can be seen that as parameters are optimized, groupings of the cameras become more distinct and the associated confusion matrices show an increase in accuracy when predicting a device. While the groupings are more distinct, there is some question about why certain parts of the UMAP representation are still quite mixed while others are more segregated. Further explanation is needed to explain this behavior.

### 9.2.6 Machine Learning Algorithm Optimization

We use AutoGluon Tabular (AGT), an ensemble learning tool for building a classifier. AGT uses several different machine learning algorithms in succession and combination by using K-fold bagging, an approach where the outcome of one algorithm becomes the input for another. In this way the optimal result is found. Unfortunately it is essentially a *black box* in that we learn little about which algorithms work best and why. Further research could be done here to explain why some algorithms might work better than others, and if different algorithms may work better for certain devices.

### 9.2.7 Implementation of FlexHash as a Framework

Finally, the implementation of FlexHash as a working framework that could be applied to a live network for testing would do much to verify this work. We hypothesize that with this tool network administrators could monitor known devices on their networks, monitor for the appearance of unknown devices and identify them by genre, and discover when devices exhibit anomalous behavior owing to an infection by botnet code.

# Appendix A

# Publications

## A.1 Published

**Comparative analysis of internet topology data sets** by M. Abdullah Canbaz, Jay Thom, and Mehmet Hadi Gunes. In 2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 635-640. IEEE, 2017.

**Analysis and prevention of security vulnerabilities in a smart city** by Ben Lupton, Mackenzie Zappe, Jay Thom, Shamik Sengupta, and Dave Feil-Seifer. In 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), pp. 0702-0708. IEEE, 2022.

**Correlation of cyber threat intelligence data across global honeypots** by Jay Thom, Yash Shah, and Shamik Sengupta. In 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC), pp. 0766-0772. IEEE, 2021.

**Casting a wide net: An internet of things testbed for cybersecurity education and research** by Jay Thom, Tapadhir Das, Bibek Shrestha, Shamik Sengupta, and Engin Arslan. In 2021 International Symposium on Performance

Evaluation of Computer and Telecommunication Systems (SPECTS), pp. 1-8. IEEE, 2021.

**Smart recon: Network traffic fingerprinting for IoT device identification** by Jay Thom, Nathan Thom, Shamik Sengupta, and Emily Hand. In 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), pp. 0072-0079. IEEE, 2022.

**Locality Sensitive Hashing for Network Traffic Fingerprinting** by Nowfel Mashnoor, Jay Thom, Abdur Rouf, Shamik Sengupta, and Batyr Charyyev. In 2023 IEEE 29th International Symposium on Local and Metropolitan Area Networks (LANMAN), pp. 1-6. IEEE, 2023.

**FlexHash-Hybrid Locality Sensitive Hashing for IoT Device Identification** by Nathan Thom, Jay Thom, Batyr Charyyev, Emily Hand, and Shamik Sengupta. In 2024 IEEE 21st Consumer Communications  Networking Conference (CCNC), pp. 368-371. IEEE, 2024.

## A.2 Submitted for Review

**Flexhash: IoT Device Identification with Hybrid Locality Sensitive Hashing** by Jay Thom, Nathan Thom, Batyr Charyyev, Emily Hand, Shamik Sengupta. Submitted to IEEE Transactions on Networking (February 2024).

# Bibliography

[1] T. Alves, T. Morris, and S.-M. Yoo, "Securing scada applications using openplc with end-to-end encryption," in *3RD ANNUAL INDUSTRIAL CONTROL SYSTEM SECURITY WORKSHOP (ICSS 2017)*, (NEW YORK), pp. 1–6, ACM, Assoc Computing Machinery, 2017.

[2] K. Tanaka and E. Kondo, "A scalable algorithm for monte carlo localization using an incremental (elsh)-l-2-database of high dimensional features," in *2008 IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, VOLS 1-9*, (NEW YORK), pp. 2784–2791, IEEE, IEEE, 2008.

[3] D. Turnbull, "A pure python lsh nearest neighbors implementation." https://softwaredoug.com/blog/2023/08/21/implementing-random-projections, August 2023. Accessed: April 12, 2024.

[4] N. Pang, J. Zhang, C. Zhang, and X. Qin, "Parallel hierarchical subspace clustering of categorical data," *IEEE transactions on computers*, vol. 68, no. 4, pp. 542–555, 2019.

[5] C. Weinschenk and J. Engebretson, "Report finds drop in number of smart home devices per home." https://www.telecompetitor.com/report-finds-drop-in-number-of-smart-home-devices-per-home, August 2022. Accessed: April 12, 2024.

[6] Q. H. Mahmoud, "Network traffic flow based machine learning technique for iot device identification," in *The Institute of Electrical and Electronics Engineers, Inc. (IEEE) Conference Proceedings*, (Piscataway), pp. 1–, The Institute of Electrical and Electronics Engineers, Inc. (IEEE), 2021.

[7] G. Author, "Inside the infamous mirai iot botnet: A restrospective analysis." https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis, September 2021. Accessed: April 12, 2022.

[8] O. klaba, "Ovh hosting suffers from record 1tbps ddos attack driven by 150k devices." https://it.slashdot.org/story/16/09/27/2042246/ovh-hosting-suffers-from-record-1tbps-ddos-attack-driven-by-150k-devices, September 2016. Accessed: April 12, 2022.

[9] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, *et al.*, "Understanding the mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, pp. 1093–1110, 2017.

[10] B. Kovacs, "Over 500,000 iot devices vulnerable to mirai botnet." https://www.securityweek.com/over-500000-iot-devices-vulnerable-mirai-botnet, Oct 2016. Accessed: April 15, 2024.

[11] P. Paganini, "Mirai botnet evolution since its source code is available online." https://resources.infosecinstitute.com/topic/mirai-botnet-evolution-since-its-source-code-is-available-online/, June 2019. Accessed: April 12, 2022.

[12] B. Lingenfelter, I. Vakilinia, and S. Sengupta, "Analyzing variation among iot botnets using medium interaction honeypots," in *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0761–0767, IEEE, 2020.

[13] R. Kolcun, D. A. Popescu, V. Safronov, P. Yadav, A. M. Mandalari, Y. Xie, R. Mortier, and H. Haddadi, "The case for retraining of ml models for iot device identification at the edge," *arXiv.org*, 2020.

[14] R. Kolcun, D. A. Popescu, V. Safronov, P. Yadav, A. M. Mandalari, R. Mortier, and H. Haddadi, "Revisiting iot device identification," *arXiv.org*, 2021.

[15] B. Charyyev and M. H. Gunes, "Iot traffic flow identification using locality sensitive hashes," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2020.

[16] K. Ding, C. Huo, B. Fan, S. Xiang, and C. Pan, "In defense of locality-sensitive hashing," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 1, pp. 87–103, 2016.

[17] V. Satuluri and S. Parthasarathy, "Bayesian locality sensitive hashing for fast similarity search," *arXiv preprint arXiv:1110.1328*, 2011.

[18] staff, "What is the history of bots?." https://www.fastly.com/learning/what-is-the-history-of-bots, Jul 2022. Accessed: April 20, 2024.

[19] J. Weizenbaum, "Eliza—a computer program for the study of natural language communication between man and machine," *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966.

[20] E. Spafford, "The usenet," in *The User's Directory of Computer Networks*, pp. 386–390, Elsevier, 1990.

[21] S. Dhenakaran and K. T. Sambanthan, "Web crawler-an overview," *International Journal of Computer Science and Communication*, vol. 2, no. 1, pp. 265–267, 2011.

[22] T. Prakash, B. K. Tripathy, and K. Sharmila Banu, “Alice: A natural language question answering system using dynamic attention and memory,” in *Soft Computing Systems: Second International Conference, ICSCS 2018, Kollam, India, April 19–20, 2018, Revised Selected Papers 2*, pp. 274–282, Springer, 2018.

[23] P. Shukla, “The compromised devices of the carna botnet: As used for the internet,” *Proceedings of the DeepSec Conferences*, vol. Special Edition: In Depth Security, 2015.

[24] J. Nazario and T. Holz, “As the net churns: Fast-flux botnet observations,” in *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 24–31, IEEE, 2008.

[25] M. Baezner and P. Robin, “Stuxnet,” tech. rep., ETH Zurich, 2017.

[26] C. Cerrudo, “An emerging us (and world) threat: Cities wide open to cyber attacks,” *Securing Smart Cities*, vol. 17, pp. 137–151, 2015.

[27] A. Gomez, H. Shahriar, V. Clincy, and A. Shalan, “Hands-on lab on smart city vulnerability exploitation,” in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1777–1782, IEEE, 2020.

[28] R. Khatoun and S. Zeadally, “Cybersecurity and privacy solutions in smart cities,” *IEEE Communications Magazine*, vol. 55, no. 3, pp. 51–59, 2017.

[29] R. Kitchin and M. Dodge, “The (in) security of smart cities: Vulnerabilities, risks, mitigation, and prevention,” *Journal of Urban Technology*, vol. 26, no. 2, pp. 47–65, 2019.

[30] M. M. Yamin, B. Katt, E. Torseth, V. Gkioulos, and S. J. Kowalski, “Make it and break it: An iot smart home testbed case study,” in *Proceedings of the*

*2nd International Symposium on Computer Science and Intelligent Control*, pp. 1–6, 2018.

[31] T. Nguyen, B. Lakshmanan, C. Lin, W. Sheng, Y. Gu, M. Liu, and S. Zhang, "A miniature smart home testbed for research and education," in *2017 IEEE 7th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*, pp. 1637–1642, IEEE, 2017.

[32] I. S. Alsukayti, "A multidimensional internet of things testbed system: Development and evaluation," *Wireless Communications and Mobile Computing*, vol. 2020, pp. 1–17, 2020.

[33] M. AbdelHafeez and M. AbdelRaheem, "Assiut iot: A remotely accessible testbed for internet of things," in *2018 IEEE Global Conference on Internet of Things (GCIoT)*, pp. 1–6, IEEE, 2018.

[34] O. A. Waraga, M. Bettayeb, Q. Nasir, and M. A. Talib, "Design and implementation of automated iot security testbed," *Computers & security*, vol. 88, p. 101648, 2020.

[35] I. A. Oyewumi, A. A. Jillepalli, P. Richardson, M. Ashrafuzzaman, B. K. Johnson, Y. Chakhchoukh, M. A. Haney, F. T. Sheldon, and D. C. de Leon, "Isaac: The idaho cps smart grid cybersecurity testbed," in *2019 IEEE Texas Power and Energy Conference (TPEC)*, pp. 1–6, IEEE, 2019.

[36] S. Siboni, V. Sachidananda, Y. Meidan, M. Bohadana, Y. Mathov, S. Bhairav, A. Shabtai, and Y. Elovici, "Security testbed for internet-of-things devices," *IEEE transactions on reliability*, vol. 68, no. 1, pp. 23–44, 2018.

[37] Y. Li, X. Su, J. Riekki, T. Kanter, and R. Rahmani, "A sdn-based architecture for horizontal internet of things services," in *2016 IEEE international conference on communications (ICC)*, pp. 1–7, IEEE, 2016.

[38] Z. Guo, Y. Hu, G. Shou, and Z. Guo, “An implementation of multi-domain software defined networking,” in *IET Conference Proceedings*, (Stevenage), The Institution of Engineering  Technology, 2015.

[39] M. AbdelHafeez, A. H. Ahmed, and M. AbdelRaheem, “Design and operation of a lightweight educational testbed for internet-of-things applications,” *IEEE Internet of Things Journal*, vol. 7, no. 12, pp. 11446–11459, 2020.

[40] T. Guo, D. Khoo, M. Coultis, M. Pazos-Revilla, and A. Siraj, “Iot platform for engineering education and research (iot peer)–applications in secure and smart manufacturing,” in *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 277–278, IEEE, 2018.

[41] P. Čeleda, J. Vykopal, V. Švábenský, and K. Slavíček, “Kypo4industry: A testbed for teaching cybersecurity of industrial control systems,” in *Proceedings of the 51st acm technical symposium on computer science education*, pp. 1026–1032, 2020.

[42] F. Sauer, M. Niedermaier, S. Kießling, and D. Merli, “Licster–a low-cost ics security testbed for education and research,” *arXiv preprint arXiv:1910.00303*, 2019.

[43] J. Munoz, F. Rincon, T. Chang, X. Vilajosana, B. Vermeulen, T. Walcarius, W. Van de Meerssche, and T. Watteyne, “Opentestbed: Poor man’s iot testbed,” in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 467–471, IEEE, 2019.

[44] A. J. Raglin, D. Huang, H. Liu, and J. McCabe, “Smart ccr iot: Internet of things testbed,” in *2019 IEEE 5th International Conference on Collaboration and Internet Computing (CIC)*, pp. 232–235, IEEE, 2019.

[45] I. Koniaris, G. Papadimitriou, and P. Nicopolitidis, "Analysis and visualization of ssh attacks using honeypots," in *Eurocon 2013*, pp. 65–72, IEEE, 2013.

[46] S. Bistarelli, E. Bosimini, and F. Santini, "A report on the security of home connections with iot and docker honeypots.," in *ITASEC*, pp. 60–70, 2020.

[47] K. Finley, "Linux took over the web, now, it's taking over the world," *Wired Magazine*, Oct 2016.

[48] J. Bennett, "The iot landscape and what it empirically looks like." https://ubuntu.com/blog/eclipse-2018-survey-the-iot-landscape-what-it-empirically-looks-like, April 2018. Accessed: April 12, 2022.

[49] G. Kambourakis, C. Kolias, and A. Stavrou, "The mirai botnet and the iot zombie armies," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, pp. 267–272, IEEE, 2017.

[50] S. Kumar, B. Janet, and R. Eswari, "Multi platform honeypot for generation of cyber threat intelligence," in *2019 IEEE 9th International Conference on Advanced Computing (IACC)*, pp. 25–29, IEEE, 2019.

[51] A. Kyriakou and N. Sklavos, "Container-based honeypot deployment for the analysis of malicious activity," in *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, pp. 1–4, IEEE, 2018.

[52] N. Memari, S. Hashim, and K. Samsudin, "Container based virtual honeynet for increased network security," in *2015 5th National Symposium on Information Technology: Towards New Smart World (NSITNSW)*, pp. 1–6, IEEE, 2015.

[53] W. Cabral, C. Valli, L. Sikos, and S. Wakeling, "Review and analysis of cowrie artefacts and their potential to be used deceptively," in *2019 International Conference on computational science and computational intelligence (CSCI)*, pp. 166–171, IEEE, 2019.

[54] Z. Zhang, H. Esaki, and H. Ochiai, "Unveiling malicious activities in lan with honeypot," in *2019 4th International Conference on Information Technology (InCIT)*, pp. 179–183, IEEE, 2019.

[55] A. Vetterl and R. Clayton, "Bitter harvest: Systematically fingerprinting low-and medium-interaction honeypots at internet scale," in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[56] M. Nawrocki, M. Wählisch, T. C. Schmidt, C. Keil, and J. Schönfelder, "A survey on honeypot software and data analysis," *arXiv preprint arXiv:1608.06249*, 2016.

[57] D. Fraunholz, D. Krohmer, H. D. Schotten, and C. Nogueira, "Introducing falcom: A multifunctional high-interaction honeypot framework for industrial and embedded applications," in *2018 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pp. 1–8, IEEE, 2018.

[58] I. Vakilinia, S. Cheung, and S. Sengupta, "Sharing susceptible passwords as cyber threat intelligence feed," in *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pp. 1–6, IEEE, 2018.

[59] L. Fan, S. Zhang, Y. Wu, Z. Wang, C. Duan, J. Li, and J. Yang, "An iot device identification method based on semi-supervised learning," in *2020 16th International Conference on Network and Service Management (CNSM)*, pp. 1–7, IEEE, 2020.

[60] D. Fraunholz, M. Zimmermann, and H. D. Schotten, “An adaptive honeypot configuration, deployment and maintenance strategy,” in *2017 19th International Conference on Advanced Communication Technology (ICACT)*, pp. 53–57, IEEE, 2017.

[61] R. J. McCaughey, “Deception using an ssh honeypot,” tech. rep., Naval Postgraduate School Monterey United States, 2017.

[62] J. M. Pittman, K. Hoffpauir, and N. Markle, “Primer–a tool for testing honeypot measures of effectiveness,” *arXiv preprint arXiv:2011.00582*, 2020.

[63] H. Noguchi, T. Demizu, N. Hoshikawa, M. Kataoka, and Y. Yamato, “Autonomous device identification architecture for internet of things,” in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pp. 407–411, IEEE, 2018.

[64] M. H. Mazhar and Z. Shafiq, “Characterizing smart home iot traffic in the wild,” in *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pp. 203–215, IEEE, 2020.

[65] S. Aneja, N. Aneja, and M. S. Islam, “Iot device fingerprint using deep learning,” in *2018 IEEE International Conference on Internet of Things and Intelligence System (IOTAIS)*, pp. 174–179, IEEE, 2018.

[66] Y. Meidan, M. Bohadana, A. Shabtai, J. D. Guarnizo, M. Ochoa, N. O. Tippenhauer, and Y. Elovici, “Profiliot: A machine learning approach for iot device identification based on network traffic analysis,” in *Proceedings of the symposium on applied computing*, pp. 506–509, 2017.

[67] R. R. Chowdhury, S. Aneja, N. Aneja, and E. Abas, “Network traffic analysis based iot device identification,” in *Proceedings of the 2020 the 4th International Conference on Big Data and Internet of Things*, pp. 79–89, 2020.

[68] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma, "Iot sentinel: Automated device-type identification for security enforcement in iot," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2177–2184, IEEE, 2017.

[69] S. Marchal, M. Miettinen, T. D. Nguyen, A.-R. Sadeghi, and N. Asokan, "Audi: Toward autonomous iot device-type identification using periodic communication," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1402–1412, 2019.

[70] O. Salman, I. H. Elhajj, A. Chehab, and A. Kayssi, "A machine learning based framework for iot device identification and abnormal traffic detection," *Transactions on Emerging Telecommunications Technologies*, p. e3743, 2019.

[71] A. Sivanathan, H. H. Gharakheili, and V. Sivaraman, "Inferring iot device types from network behavior using unsupervised clustering," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*, pp. 230–233, IEEE, 2019.

[72] J. Bao, B. Hamdaoui, and W.-K. Wong, "Iot device type identification using hybrid deep learning approach for increased iot security," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*, pp. 565–570, IEEE, 2020.

[73] S. A. Hamad, W. E. Zhang, Q. Z. Sheng, and S. Nepal, "Iot device identification via network-flow based fingerprinting and learning," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 103–111, IEEE, 2019.

[74] B. Bezawada, M. Bachani, J. Peterson, H. Shirazi, I. Ray, and I. Ray, "Iotsense: Behavioral fingerprinting of iot devices," *arXiv preprint arXiv:1804.03852*, 2018.

[75] M. S. Gill, D. Lindskog, and P. Zavarsky, "Profiling network traffic behavior for the purpose of anomaly-based intrusion detection," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 885–890, IEEE, 2018.

[76] N. Yousefnezhad, A. Malhi, and K. Främling, "Automated iot device identification based on full packet information using real-time network traffic," *Sensors*, vol. 21, no. 8, p. 2660, 2021.

[77] A. Aksoy and M. H. Gunes, "Automated iot device identification using network traffic," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pp. 1–7, IEEE, 2019.

[78] N. Ammar, L. Noirie, and S. Tixeuil, "Network-protocol-based iot device identification," in *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 204–209, IEEE, 2019.

[79] X. Feng, Q. Li, H. Wang, and L. Sun, "Acquisitional rule-based engine for discovering internet-of-things devices," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 327–341, 2018.

[80] N. Ammar, L. Noirie, and S. Tixeuil, "Autonomous identification of iot device types based on a supervised classification," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2020.

[81] J. Kotak and Y. Elovici, "Iot device identification using deep learning," in *Computational Intelligence in Security for Information Systems Conference*, pp. 76–86, Springer, 2019.

[82] B. A. Desai, D. M. Divakaran, I. Nevat, G. W. Peter, and M. Gurusamy, "A feature-ranking framework for iot device classification," in *2019 11th International conference on communication systems & networks (COMSNETS)*, pp. 64–71, IEEE, 2019.

[83] D. Hadden, "Do smart cities improve citizen well-being," vol. 4, pp. 389–413, 2018.

[84] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, *et al.*, "Smartsantander: Iot experimentation over a smart city testbed," *Computer Networks*, vol. 61, pp. 217–238, 2014.

[85] S. Latre, P. Leroux, T. Coenen, B. Braem, P. Ballon, and P. Demeester, "City of things: An integrated and multi-technology testbed for iot smart city experiments," in *2016 IEEE international smart cities conference (ISC2)*, pp. 1–8, IEEE, 2016.

[86] J. Thom, T. Das, B. Shrestha, S. Sengupta, and E. Arslan, "Casting a wide net: An internet of things testbed for cybersecurity education and research," in *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), 2021*, 2021.

[87] G. F. Lyon, *Nmap network scanning: The official Nmap project guide to network discovery and security scanning.* Insecure, 2009.

[88] epsilonRT and Pascal, Jean, "mbpoll." https://github.com/epsilonrt/mbpoll. Accessed: April 21, 2024.

[89] Marty Roesch, "Snort ids." https://www.snort.org/. Accessed: March 23, 2024.

[90] Gerald Combs, "Wireshark." https://www.wireshark.org/. Accessed: March 21, 2024.

[91] Infosec and A. Yadav, "Network design: Firewall, ids/ips." https://resources.infosecinstitute.com/topic/network-design-firewall-idsips/, 2020. Accessed: April 10, 2024.

[92] U. DOE, "21 steps to improve cyber security of scada networks," 2002.

[93] U. CISA, "Ics advisory (icsa-12-102-02) koyo ecom modules vulnerabilities," 2018.

[94] T. Alves and T. Morris, "Openplc: An iec 61,131-3 compliant open source industrial controller for cyber security research," vol. 78, pp. 364–379, 2018.

[95] C. Benitez, "21+ internet of things statistics, facts  trends for 2024," Feb 2023.

[96] K. Benzekki, A. El Fergougui, and A. Elbelrhiti Elalaoui, "Software-defined networking (sdn): a survey," *Security and communication networks*, vol. 9, no. 18, pp. 5803–5833, 2016.

[97] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

[98] S. S. Bhunia and M. Gurusamy, "Dynamic attack detection and mitigation in iot using sdn," in *2017 27th International telecommunication networks and applications conference (ITNAC)*, pp. 1–6, IEEE, 2017.

[99] R. Rayson, "Rayson/cowrie." https://hub.docker.com/r/rayson/cowrie/, January 2016. Accessed: April 12, 2022.

[100] T. D. Community, "The apache httpd server project." https://hub.docker.com/$_/httpd, April 2022. Accessed: April 12, 2022.$

[101] S. Stillard, “Stilliard/docker-pure-ftpd: Docker pure-ftpd server.” https://github.com/stilliard/docker-pure-ftpd, April 2022. Accessed: April 12, 2022.

[102] W. Priesnitz Filho, C. Ribeiro, and T. Zefferer, “An ontology-based interoperability solution for electronic-identity systems,” in *2016 IEEE International Conference on Services Computing (SCC)*, pp. 17–24, IEEE, 2016.

[103] M. Pirker, P. Kochberger, and S. Schwandter, “Behavioural comparison of systems for anomaly detection,” in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pp. 1–10, 2018.

[104] H. M. Kim, H. M. Song, J. W. Seo, and H. K. Kim, “Andro-simnet: Android malware family classification using social network analysis,” in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pp. 1–8, IEEE, 2018.

[105] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati, “An open digest-based technique for spam detection.,” in *PDCS*, pp. 559–564, Citeseer, 2004.

[106] M. N. Marsono, “Packet-level open-digest fingerprinting for spam detection on middleboxes,” *International Journal of Network Management*, vol. 22, no. 1, pp. 12–26, 2012.

[107] B. Charyyev and M. H. Gunes, “Locality-sensitive iot network traffic fingerprinting for device identification,” *IEEE Internet of Things Journal*, vol. 8, no. 3, pp. 1272–1281, 2020.

[108] N. Sarantinos, C. Benzaïd, O. Arabiat, and A. Al-Nemrat, “Forensic malware analysis: The value of fuzzy hashing algorithms in identifying similarities,” in *2016 IEEE Trustcom/BigDataSE/ISPA*, pp. 1782–1787, IEEE, 2016.

[109] F. Breitinger and H. Baier, “Properties of a similarity preserving hash function and their realization in sdhash,” in *2012 Information Security for South Africa*, pp. 1–8, IEEE, 2012.

[110] J. Oliver, C. Cheng, and Y. Chen, "Tlsh–a locality sensitive hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pp. 7–13, IEEE, 2013.

[111] A. Andoni and P. Indyk. https://www.mit.edu/ andoni/LSH/manual.pdf, Jun 2005. Accessed: Feb. 10, 2024.

[112] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pp. 21–29, IEEE, 1997.

[113] A. Gionis, P. Indyk, R. Motwani, *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, pp. 518–529, 1999.

[114] W. J. Buchanan, "Nilsimsa similarity hash." https://asecuritysite.com/encryption/nil, 2022. Accessed: April 12, 2022.

[115] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital investigation*, vol. 3, pp. 91–97, 2006.

[116] V. Roussev, "An evaluation of forensic similarity hashes," *digital investigation*, vol. 8, pp. S34–S41, 2011.

[117] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[118] F. Pagani, M. Dell'Amico, and D. Balzarotti, "Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pp. 354–365, 2018.

[119] F. Rosenblatt, "Principles of neurodynamics. perceptrons and the theory of brain mechanisms," *American Journal of Psychology*, vol. 76, p. 705, 1963.

[120] E. Becht, L. McInnes, J. Healy, C.-A. Dutertre, I. W. Kwok, L. G. Ng, F. Ginhoux, and E. W. Newell, "Dimensionality reduction for visualizing single-cell data using umap," *Nature biotechnology*, vol. 37, no. 1, pp. 38–44, 2019.

[121] J. Thom, N. Thom, S. Sengupta, and E. Hand, "Smart recon: Network traffic fingerprinting for iot device identification," in *2022 IEEE CCWC*, pp. 0072–0079, IEEE, 2022.

[122] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," 2020.

[123] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "Autogluon-tabular: Robust and accurate automl for structured data," *arXiv preprint arXiv:2003.06505*, 2020.

[124] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine Learning*, vol. 63, no. 1, p. 3–42, 2006.

[125] A. V. Dorogush, V. Ershov, and A. Gulin, "Catboost: gradient boosting with categorical features support," *CoRR*, vol. abs/1810.11363, 2018.

[126] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 3149–3157, Curran Associates Inc., 2017.

[127] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), p. 785–794, Association for Computing Machinery, 2016.

[128] T. G. Dietterich, "Ensemble methods in machine learning," in *Multiple Classifier Systems*, (Berlin, Heidelberg), pp. 1–15, Springer Berlin Heidelberg, 2000.

[129] P. W. M. BAMBANG PARMANTO and H. R. DOYLE, “Reducing variance of committee prediction with resampling techniques,” *Connection Science*, vol. 8, no. 3-4, pp. 405–426, 1996.

[130] K. M. Ting and I. H. Witten, “Stacking bagged and dagged models,” in *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML ’97, (San Francisco, CA, USA), p. 367–375, Morgan Kaufmann Publishers Inc., 1997.

[131] R. Caruana, A. Niculescu-Mizil, G. Crew, and A. Ksikes, “Ensemble selection from libraries of models,” in *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML ’04, (New York, NY, USA), p. 18, Association for Computing Machinery, 2004.