ABSTRACT

CROSS-DEVICE FEDERATED INTRUSION DETECTOR FOR EARLY STAGE BOTNET PROPAGATION

by Angela Grace Famera

A botnet is an army of zombified computers infected with malware and controlled by malicious actors to carry out tasks such as Distributed Denial of Service (DDoS) attacks. Billions of Internet of Things (IoT) devices are primarily targeted to be infected as bots since they are configured with weak credentials or contain common vulnerabilities. Detecting botnet propagation by monitoring the network traffic is difficult as they easily blend in with regular network traffic. The traditional machine learning (ML) based Intrusion Detection System (IDS) requires the raw data to be captured and sent to the ML processor to detect intrusion. In this research, we examine the viability of a cross-device federated intrusion detection mechanism where each device runs the ML model on its data and updates the model weights to the central coordinator. This mechanism ensures the client's data is not shared with any third party, terminating privacy leakage. The model examines each data packet separately and predicts anomalies. We evaluate our proposed mechanism on a real botnet propagation dataset called MedBIoT. In addition, we also examined whether any device taking part in federated learning can employ a poisoning attack on the overall system.

CROSS-DEVICE FEDERATED INTRUSION DETECTOR FOR EARLY STAGE BOTNET PROPAGATION

A Thesis

Submitted to the

Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Science

by

Angela Grace Famera

Miami University

Oxford, Ohio

2022

Advisor: Dr. Suman Bhunia

Reader: Dr. Daniela Inclezan

Reader: Dr. Khodakhast Bibak

This Thesis titled

CROSS-DEVICE FEDERATED INTRUSION DETECTOR FOR EARLY STAGE BOTNET PROPAGATION

by

Angela Grace Famera

has been approved for publication by

The College of Engineering and Computing

and

The Department of Computer Science & Software Engineering

Dr. Suman Bhunia

Dr. Daniela Inclezan

Dr. Khodakhast Bibak

Contents

List of Tables

List of Figures

Acknowledgements

I want to thank all of my committee members for their guidance throughout the thesis process. This includes Dr. Suman Bhunia, my thesis advisor, Dr. Daniela Inclezan, and Dr. Bibak Khodakhast.

I thank Dr. Bhunia for his research guidance. He has demonstrated ample patience and knowledge while directing me in my research, as well as dedicated much of his time leading me in my work not only through the school semesters but through summer and winter breaks as well. I also had the pleasure of serving as his graduate assistant for Network Security and Ethical Hacking over the last year. Through these classes, he has helped me progress in my knowledge as a cybersecurity professional and helped me develop skills that made me stand out in and receive offers from the security industry.

I also want to thank Dr. Raj Mani Shukla as an external advisor for many aspects of the machine learning process used in this research. He has also dedicated time to helping Dr. Bhunia and me overcome hurdles with machine learning that we would have not been able to surpass otherwise.

Last, I want to thank all of my friends, classmates, and family for supporting me through the arduous process that is research. They have helped me push through adversity and showed tremendous support during all my academic endeavors.

Chapter 1

Introduction

Computer security plays an imperative role in our technologically advanced society. Malware of all sorts, ranging from all levels of destruction, is continuously threatening individuals and organizations. On the company level, 65% of business leaders in 2019 felt their cybersecurity risks were increasing, and the average cost of a malware attack on a company is currently \$2.6 million [1]. On the individual level, 64% of Americans have never checked to see if they were affected by a data breach in 2020 [1]. With the advancement of smart technology, attacks on IoT devices also tripled in the first half of 2019, and the Mirai Distributed Denial of Service (DDoS) worm (the Mirai botnet) was the third most common IoT threat in 2018 [1]. By 2025, there will be 41.6 billion connected IoT devices generating 79 zettabytes of data [2], and damage related to cybercrime is projected to hit \$10.5 trillion annually [1].

Security is a catch-up game. The attacker comes up with their strategy and weapon, and the security professionals play defense in order to mitigate the attack. Security is historically passive and defensive. Countermeasures and defenses for new attacks are implemented after the damage is done. Malware detection is often signature-based, resulting in passive monitoring of networks and systems for things that appear to be malicious base on previously collected data and experience. This rat race is the reason why research in computer security can never end.

Effective techniques to actively detect malicious network traffic are firewalls, Intrusion Detection Systems (IDS), and Intrusion Prevention Systems (IPS). Firewalls block and filter network traffic, IDS alert network administrators of anomalies in the network, and IPS take action against anomalies in the network. IDS/IPS are generally installed after a firewall on the edge of the network. These defense techniques require analyzing raw network traffic data, which can cause privacy concerns when multiple devices and data silos are owned by independent individuals and organizations. As effective as they are, firewalls, IDS, and IPS are not built to conserve data privacy or learn independently from the data they are fed. Federated learning can address both privacy and self-learning. This research aims to create an IDS to detect IoT botnets on a packet-by-packet basis using federated learning before an attack can take place. We also examine how poisoning attacks affect model performance.

1.1 Motivation

This research aims to add to our defense strategies against malware; specifically botnets. Botnets were a popular research focus starting in the early 2000s as well as around 2016 with the release of Mirai (see Section 2.1.4). Mirai and other IoT botnet variants have only increased and advanced since then with the surge of IoT devices. The surplus of innovative research is from that 2000-2017 time frame, and limited updated discoveries on botnets outside of regurgitated analysis using standard centralized machine learning techniques and surveys exist compared to research on other forms of malware. This research aims not only to bring back focus to botnets but also intends to use a federated learning-based intrusion detection system to prevent botnet attacks before they occur while conserving confidentiality and data privacy. We aim to answer the following research questions:

1. Is it possible to classify botnet malware while in the propagation and C&C phase on a packet-by-packet basis?

2. Can we create an intrusion detection system using federated learning to predict botnet propagation before an attack occurs by analyzing the content in independent network packets?

3. Can we decrease model performance by simulating a targeted poisoning attack?

1.2 Contributions

Understanding how to detect and prevent a botnet attack before it happens can benefit the security industry in many ways. The main contributions of this research are the following:

- We propose a new intrusion detection system mechanism based on federated learning to preserve data privacy. Each device takes part in federated learning by training the model locally so no data is shared.

- Usage of raw packet data from real and emulated network traffic captures during propagation and C&C communication for three popular IoT botnets.

- We propose an online model analyzing network data on a per packet basis

- We designed a neural network model to identify botnet traffic at its early stages (i.e. pre-attack)

- We examined whether poisoning attacks have an impact on model performance. A poisoning attack occurs when an adversary injects bad data into a model. We simulate this kind of attack using label-flipping based on known malware trends.

In this chapter, we introduced our research problem and its applications to computer security. Chapter 2 will provide some background knowledge on botnets, federated learning, and current research published on federated learning in cybersecurity. Chapter 3 offers a holistic overview of our proposed architecture and how it differs from a standard IDS. In Chapter 4, we discuss how we reviewed various published datasets, selected one, and converted all the pcap files to csv format. Chapter 5 explains our design setup, and Chapter 6 goes over our results and discussion. We conclude with Chapter 7, where we discuss limitations and opportunities for future work.

Chapter 2

Background & Related Work

In this chapter, we provide necessary background knowledge on botnets, their command and control architectures, and a few variants. We also provide an introduction to federated learning and the difference between cross-silo and cross-device learning models. The chapter concludes with some recent related work and how our research differs.

2.1 Botnet Overview and Variants

In this section, we discuss the history of botnets and how they work. We also briefly discuss three impactful IoT botnet variants used in the MedBIot [3] dataset: Bashlite, Mirai, and Torii.

2.1.1 A Brief History of Botnets

Before examining the complexities of botnets, it is important to understand their history and original intended purpose. While botnets often serve the hacker today, they were originally developed to assist with the administration of Internet Relay Chat (IRC) servers [4]. IRC is a text-based protocol developed in 1988 used by connected computers for real-time text messaging [5]. IRC supported group chats in discussion rooms known as "channels," as well as private messages between individual users and data transfers [5]. The IRC consisted of five main components [6]:

1. **Servers:** The point where clients and other servers could connect. The IRC networks replicated a spanning tree, where each server acts as the central node for the rest of the network it sees.

2. **Clients:** Anything connecting to the server that is not another server. Clients are distinguished by unique nicknames, and the server keeps track of the real name of the host the client is running on, the client's username on that host, and the server to which the client is connected.

3. **Operators:** A special class of clients allowed to perform maintenance functions on the network. They keep order within the IRC.

4. **Channels:** A named group of one or more clients which will receive all messages sent to that channel.

5. **Channel Operators:** Commonly referred to as a “chop” or “chanop,” own a channel and are responsible for keeping order and sanity within the channel.

IRC became a popular method used by botmasters to send commands to individual computers in their botnet through specific channels, public IRC networks, or separate IRC servers [5]. Command-and-control (C&C) was coined after the server containing the channels used to control the bots [5]. Originally, IRC administrators created botnets to perform automated functions to assist with the administration of IRC servers [4]. Inevitably, this capability was used for more malignant purposes.

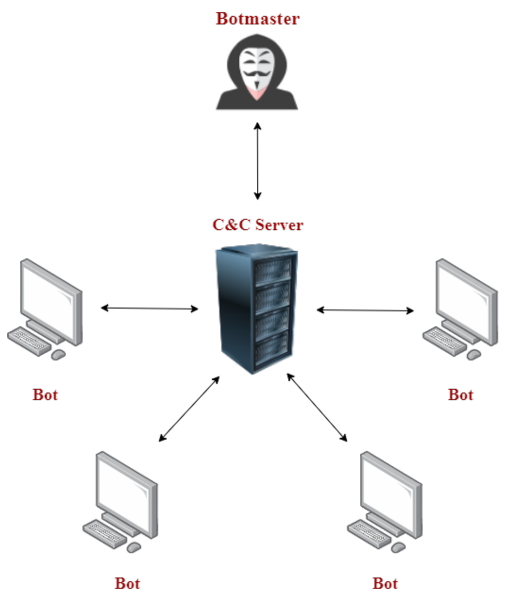
Botnets developed into an attack mechanism used by cybercriminals to perform various malicious actions, most commonly being Distributed Denial of Service (DDoS) attacks, spam distribution, and network scanning, exploration, and exploitation [7]. A botnet is a collection of bots connected to and controlled by a C&C channel [8]. Bots are constituted of host machines, devices, and computers infected by malicious code that enslaves them to the C&C [8]. The C&C updates and guides bots to perform the desired task by acting as the communication link between the bots and an individual known as a botmaster [8]. The botmaster’s primary purpose is to control the botnet by issuing commands through the C&C to perform malicious and illegal activities.

2.1.2 Botnet C&C Architectures

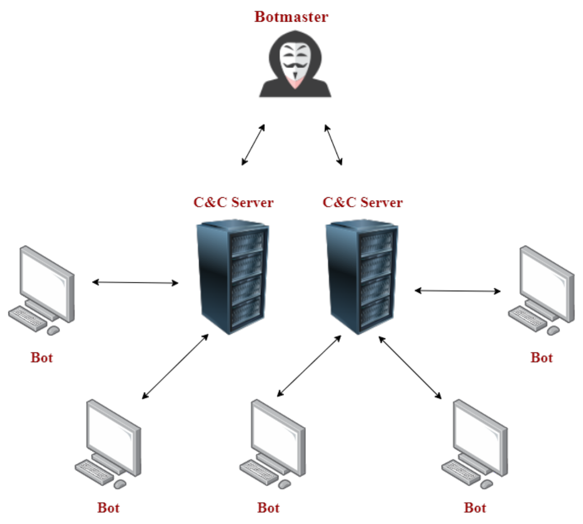
The most critical part of a botnet is the C&C architecture [8]. The C&C is the only way to control the bots within a botnet and is responsible for their smooth and collective operation. Therefore, if the C&C was destroyed, the botnet would no longer be able to carry out its intended purpose. The three most common botnet C&C structures are centralized, decentralized, and hybridized control [8, 9]:

- **Centralized:** Centralized C&C primarily uses HTTP and IRC based protocols. Here, there exists a central C&C giving instructions to the botnet. It is important to note that a botnet structure is still considered centralized even if there exists more than one server, so long as all the bots report to one of the n servers for communication. See Figure 2.1 for better visualization of the differences in these centralized structures. This architecture allows the botmaster to react quickly to events given the ability to receive direct feedback from the bots. The botmaster can also coordinate the botnet more efficiently, given the ability to easily monitor the botnet’s status and distribution. However, if the C&C is taken down, the whole botnet is taken down with it because it is also a central point of failure.

- **Decentralized:** Unlike centralized C&C, decentralized C&C uses Peer-to-Peer (P2P) protocols in which all the bots are connected. These protocols focus on hiding the C&C channels, and botmasters can call different bots for different issues. See Figure 2.2. Decentralized C&C is good for the attacker because the detection of one bot does not jeopardize the whole botnet. It also allows for more flexible and robust botnets.



(a) Centralized One Server



(b) Centralized Multiple Servers

Figure 2.1: Centralized C&C Architecture

- **Hybridized:** Hybridized botnets use a combination of the centralized and decentralized C&C structure, often using encryption to hide botnet traffic. This structure divides bots into two groups: servant and client bots. Servant bots are configured with "static and routable" IP addresses, and are able to act as both clients and servers. Client bots, on the other hand, are configured with "dynamically designated or non-routable IP addresses," and block any incoming connection. Servant bots listen to determined ports for incoming connections and use "self-generated symmetric encryption" to communicate. See Figure 2.2 for more information.

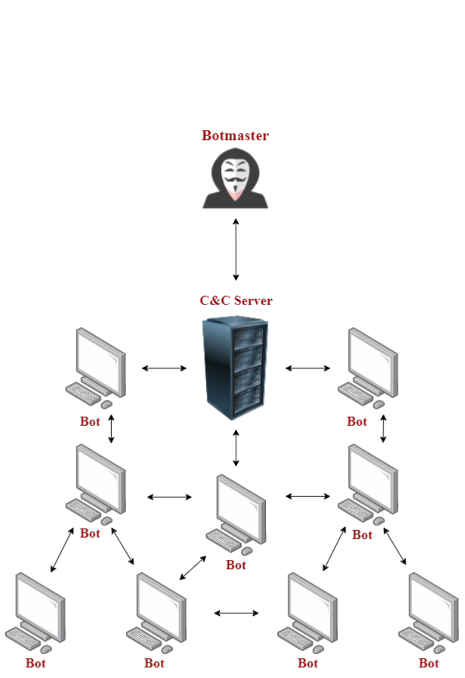
Centralized or decentralized, one of the most serious threats to the internet today is the collection of infected devices controlled by the hand of a singular malicious entity [8].

2.1.3 The Botnet Life Cycle

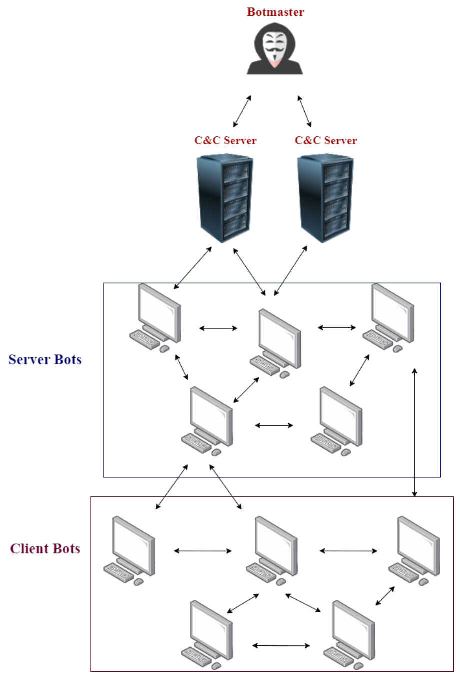
Botnets have a unique life cycle [8, 10]:

- **Initial Injection:** The host device is infected and becomes a potential bot. During this phase, the attacker may use an array of different infection mechanisms, such as infected files and removal disks or forced downloads of malware from various websites.

- **Secondary Injection:** The infected host runs a program that transforms the device into a bot. Scripts are executed by the infected host that fetches the device's binary code via FTP, HTTP, or P2P protocol. This binary contains the addresses of the machines and may be encoded directly as hard-coded IP addresses or domain names. During this phase, the host becomes a bot.



(a) Decentralized C&C



(b) Hybridized C&C

Figure 2.2: Decentralized and Hybridized C&C Architecture

- **Connection:** Through a process known as rallying, the bot establishes a connection with the C&C server and becomes part of the botmaster's botnet. This phase occurs every time the host is restarted to let the botmaster know that the device is still able to receive and act on malicious commands.
- **Performance:** The bot is able to receive commands and perform attacks. The C&C enables the botmaster to monitor and control the botnet however seen fit.
- **Maintenance:** The bot's malware is updated for the botmaster to maintain the botnet. Here binary's are updated to ensure a connection with the C&C remains established.

Methods of propagation include but are not limited to email attachments, infected websites, and previously installed backdoors [8].

2.1.4 Important Botnets Variants

Many different botnet variants circle our internet today. We discuss Bashlite, Mirai, and Torii because 1) they produce the malware traffic captured in the dataset we utilize (see Section 4.1.5), and 2) they have an incredible impact on malware migrating to the IoT space. A great example of this is Mirai, which is considered the first notable botnet to launch a DDoS attack using IoT devices.

Bashlite, 2014

Bashlite, also known as Gafgyt, Lizkebab, Qbot, Torlus, and LizardStresser, was originally seen in 2014 utilizing Shellshock (a software bug) in bash shell to exploit devices running BusyBox [11]. This exploit, coined as *Bashdoor*, is used by the malware to infect Linux systems to launch DDoS attacks [11]. The botnet was created by the Lizard Squad, a black hat hacking group specializing in DDoS attacks against gaming services [12].

In 2014, the botnet launched massive DDoS attacks against banks, telecommunication, and government agencies in Brazil, as well as three large US gaming companies [13]. Lizard Squad launched their DDoS attacks first on League of Legends servers taking them offline, followed by attacks on PlayStation Network and multiple servers run by Blizzard, taking the networks down for nearly a day [12]. Using IoT devices, the botnet has grown large enough to launch a 400 Gbps attack without amplification [13].

The source code was leaked in 2015, and in 2016 around one million devices, most commonly manufactured by Dahua Technology, were reported as being infected by the malware in Brazil, Colombia, and Taiwan [11, 14]. Of the identifiable bots in 2016, 96% of them were reported as IoT devices, 4% home routers, and less than 1% Linux servers [11]. While the malware can support multiple C&C servers, most variants have a single C&C hardcoded IP address [11], making it a centralized C&C architecture.

Mirai, 2016

Mirai was an IoT botnet created by Paras Jha and Josiah White, co-founders of Protraf Solutions [15]. Mirai has created the basis for many botnets that exist today due to the original creators releasing the source code to the world (under the name "Anna-Senpai") back in 2016 on Hackforums.

Mirai's first large-scale attack was on OVH, a French hosting platform. A source of inspiration for Mirai was surrounding Jha's interest in hosting a Minecraft game server. Minecraft is a popular online video game where upwards of $100,000 can be earned by hosting a game server in the summer months [16]. As a result, Jha was interested in performing DDoS attacks against other Minecraft servers to attract business to his server [16, 17]. This resulted in the first major DDoS attack, which occurred on September 19th, 2016, when Mirai was used against OVH, a popular Minecraft hosting service.

During the DDoS attack, Mirai used 145,000 infected devices to send 1.1 Tbps of data traffic to OVH's servers, bringing their services to a halt. This amount of traffic was unprecedented in 2016, being magnitudes larger than vDOS, which maxed out at 50 Gbps [16]. vDOS was a popular DDoS-as-a-service provider and was used in the gaming industry to gain a competitive advantage against opponents. Mirai could target multiple IP addresses at the same time, allowing it to infect an entire network, rather than a specific server, application, or website [16].

Shortly after the attack on OVH, Mirai launched another DDoS attack against Brain Kreb's security website *Krebs On Security* [16]. Jha admitted that this attack was paid for by a customer who rented a bunch of Mirai-infected devices [18]. This DDoS attack peaked at 623 Gbps, forcing Kreb's DDoS mitigation service, Akamai, to drop Kreb's website due to the incurred costs by the attack [16]. It took four days for *Krebs On Security* to go back online. Right after this attack was when Mirai's source code was released to the dark web.

Soon after the attack on *Krebs On Security*, the release of the code resulted in an attack that took down Dyn. Inc's DNS servers, bringing down major websites on the east coast of the United States [17]. Upon the release of Mirai's source code, the hacker was able to launch the attack in October 2016 that left the US east coast without access to Amazon, Netflix, PayPal, and Reddit [16]. To this day, Dyn, Inc. is still unable to assess the full weight of the assault.

The *Krebs On Security* attack brought Mirai to the forefront of the FBI's investigation. The FBI joined with the private industry to figure out the inner workings of Mirai. Akamai created honeypots that allowed the investigators to observe how infected devices communicated with Mirai's C&C servers [16]. After the Dyn, Inc attack, a collaboration between investigators grew, and engineers from around the world came together to discuss the threat Mirai poses. The engineers see the attack on Dyn, Inc. as a proof of concept that it can affect the entire Internet if it is not mitigated. While studying the Mirai servers, the investigators noticed a trend of the botnet targeting gaming servers, and more specifically, Minecraft servers. This led investigators to discover the original intention of Mirai, which was to target Minecraft game servers in order to gain a competitive edge [16].

Torii, 2017

Torii is a botnet that surfaced in 2017 and gained notoriety for more advanced techniques used when compared to Mirai and Bashlite [19]. It is able to infect a wide range of device architectures, such as MIPS, ARM, x86, x64, PowerPC, SuperH, and others [19]. The malware prioritizes stealth and persistence, using encryption when communicating with its C&C server and tunneling communication through the Tor network [20]. Infecting systems that have been Telnet exposed and protected by weak credentials, the malware uses commands like "wget" and "busybox wget" to deliver the binary payloads, and is special in the sense that it can survive device system reboots [20]. As of now, the purpose of Torii is assumed to be DDoS attacks for mining cryptocurrencies, but this is speculation as the intentions are still unclear [20].

2.2 Applications of Machine Learning in Cybersecurity

Machine learning is the study of algorithms used for building applications that learn from data and are self-programmed to improve their accuracy over time [21]. Algorithms in machine learning are trained to recognize patterns in large quantities of data to make predictions and accurate decisions with new data [21]. The final algorithm used after all adjustments are made to produce the desired result is known as a model [21]. Similar to how humans learn from experience, machine learning attempts to teach computers to do the same, making it a sub-field of Artificial Intelligence (AI) and a plausible active approach to malware detection. There are four basic steps performed by data scientists when building machine learning applications:

1. Selecting and preparing training data [21]. This data will be representative of the input ingested by the algorithm used to solve the desired problem [21].
2. Choosing an algorithm to run on the training data [21]. The type of algorithm chosen will be dependent on the data available and the type of problem that needs to be solved [21].
3. Training the algorithm to construct a model [21]. This involves repeatedly running variables through the algorithm and adjusting weights and biases in order to produce an accurate and consistent output or result [21].
4. Using the model with the new data and ensuring it improves in accuracy and effectiveness over time [21].

The following subsections go into further detail about the types of common machine learning methods and how they differ from federated learning.

2.2.1 Types of Machine Learning

Machine learning uses many different methods to learn from data and build models. The five most common methods are supervised, unsupervised, semi-supervised, reinforcement, and deep learning methods [21, 22].

Supervised Learning

Supervised learning is one of the most widely used machine learning methods and utilizes labeled datasets [22]. Labeled datasets are designed to train or "supervise" algorithms into accurately classifying data or prediction outcomes [23]. Supervised learning can be separated into two types of problems when data mining: classification and regression [23].

Classification is the problem of assigning a category to each item in the dataset [24]. An example would be classifying a dog as a German Shepherd or a Movie as a comedy. The number of classification categories for a given dataset is usually under a few hundred, but can be unbounded when it comes to tasks like text or speech classifications [24]. Common types of classification algorithms are linear classifiers, support vector machines, decision trees, and random forests [24].

Regression is the problem of predicting a real value for each item [24]. It is a learning method used to understand the relationship between dependent and independent variables [23]. Some examples of regression are predicting stock values in a given market, or predicting sales revenues for a given business [23, 24]. Popular regression algorithms are linear regression, logistic regression, and polynomial regression [23].

Unsupervised Learning

Unsupervised learning involves analyzing unlabeled data to extract meaningful features necessary to sort, label, and classify the data in real time without human intervention [21]. These algorithms are used to identify patterns humans would miss [21]. Unsupervised learning is used for three main tasks: clustering, association, and dimensionality reduction [23]. Clustering is the problem of partitioning a set of items into homogeneous subsets and is often used with very large datasets [24]. It is used to group unlabeled data based on their similarities or differences [23]. Association is used to find relationships between variables in a given dataset [24]. These patterns are in the form of "if-then" relationships called association rules [21]. Dimensionality reduction is used to transform an initial representation of items into a lower-dimensional representation while preserving some of the original properties [24]. In other words, it reduces the number of data inputs in a dataset with a large number of features to a manageable size while still preserving the integrity of the data [23].

Semi-Supervised Learning

Semi-supervised learning is a combination of supervised and unsupervised learning. It is a happy medium that uses labeled and unlabeled training data and is useful when relevant features are difficult to extract from data [23].

Reinforcement Learning

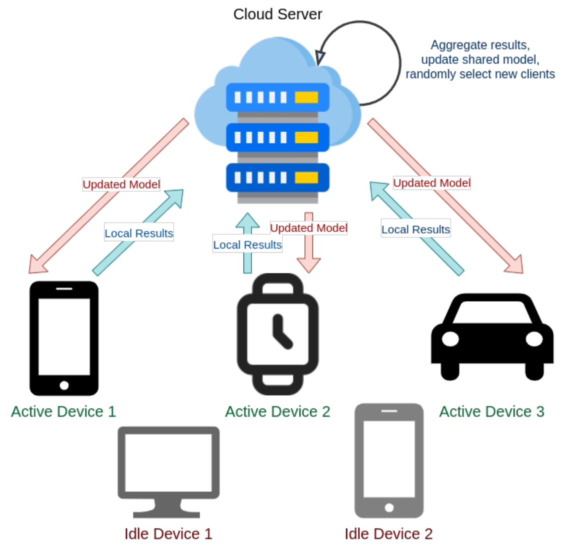
Reinforcement machine learning, rather than being trained through sample data, learns as it goes using trial and error [21]. Training data used in reinforcement learning is assumed to provide only an indication as to whether an action is correct or not [22]. Incorrect actions denote that there still exist problems in finding the correct action and that the model still needs to improve [22]. On the other hand, a sequence of successful outcomes is reinforced to develop the best recommendations for a given problem, [21].

Deep Learning

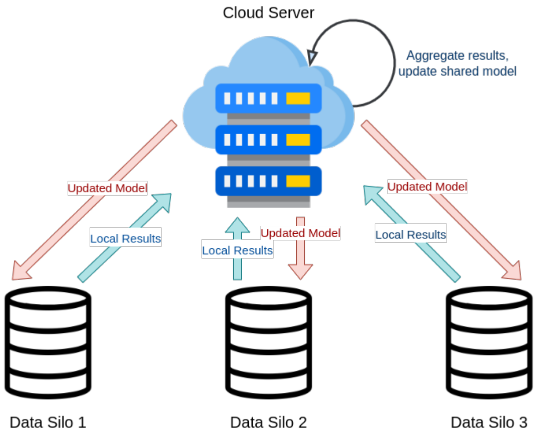
Deep learning is a subset of machine learning where the algorithms define an artificial neural network designed to learn the way a human brain learns [21]. They require large amounts of data to pass through multiple layers of calculations and make use of gradient-based optimization algorithms to adjust and improve outcomes [22, 21]. Current types of deep learning models are convolutional neural networks (CNNs) and recurrent neural networks (RNNs).

2.2.2 Federated Learning Process

The federated learning process used in this research encompasses a combination of supervised and deep learning. Federated learning is a machine learning method where multiple clients collectively train a model using their own data in a decentralized manner under the guidance of a central server [25]. In 2016, the term "Federated Learning" was coined by McMahan



(a) Cross-Device



(b) Cross-Silo

Figure 2.3: Federated Learning Architectures

et al. in [26] as a learning task is solved by a loose federation of participating devices. Federated learning differs from distributed learning in the sense that a traditional distributed system encompasses distributed computation and storage [27]. The primary advantage of the federated learning approach is to remove a model's need for direct access to raw training data, resulting in enhanced privacy [26]. Its advantages are often described for the medical setting, where medical organizations can train models on patient data without violating privacy laws or exposing clients' medical history. Because of the emphasis on privacy, this makes the federated learning framework attractive for cybersecurity, given it will inherently protect data security and confidentiality [28]. There are two notable types of federated learning, 1) Cross-device and 2) Cross-silo (see Figure 2.3). The following section will discuss each one more in-depth.

Crossed Device Federated Learning

The clients are a very large number (i.e. thousands or millions) of mobile, edge, or IoT devices [25]. Here each client stores its own data and is unable to see the data produced by another client, and the data is not independently or identically distributed [25]. There is a central server that supervises training but does not have access to the raw data, only the model parameters returned by the clients. In this type of learning style, only a fraction of the clients are available via the internet or similar connections. This produces three main concerns with the Cross-device approach: 1) communication (i.e. slow connections between clients and server), 2) clients failing or dropping out during training due to bandwidth and battery or processing power, and 3) potential bias in devices selected given clients cannot be indexed directly [25].

We use the Cross-device approach in our research. We use a labeled dataset and feed it through distributed clients sharing a simple neural network model for training. We aim to reduce the complexity of our network given the low processing or computing power IoT devices may exhibit. In our simulation, each client acts as an IoT device and stores network packet data.

Cross-Silo Federated Learning

A data silo is a repository of data stored in a standalone system controlled by a single department or business unit within an organization [29]. As the name suggests, Cross-silo federated learning trains a model on siloed data, making the clients different organizations or geographically distributed datacenters (typically 2 - 100 clients) [25]. Similar to Cross-device, data is generated locally and remains decentralized while a central server organizes training [25]. The main concerns with this method surround communication or computation, but each client has an ID for specified access and all the clients can participate in the federation rounds (unlike Cross-device, where clients are randomly selected and should not appear more than once for training).

The Federated Learning Lifecycle

Similar to the steps mentioned at the beginning of the section for building a machine learning model, federated learning has its own lifecycle. The steps in the federated learning process are as follows [25]:

1. Identifying the problem to be solved with federated learning.
2. Training data is distributed among clients (simulation environment).
3. Federated model training begins.
   (a) The server samples a set of clients.
   (b) Selected clients download current model weights and a training program.
   (c) Clients locally compute and update the model.
   (d) The server collects an aggregate of the device updates.
   (e) The server locally updates its model based on the aggregated data computed from the clients.
4. The federated model is evaluated after sufficient training.
5. The model is deployed within the data center or network using a staged rollout.

In our research, we completed steps 1-4. We identified our problem domain as the malware prediction pre-attack phase. Our training data was distributed randomly among our non-ID clients. However, in our work, all the clients participated and were selected. They could have been randomly selected, but we used a smaller number of clients with larger distributions of data. We trained our model 10 times with and without a poising attack as described in Section 6.1. The only step we did not complete is Step 5, given we did not deploy our model anywhere.

2.3 Related Work

Malware detection and federated learning are widely researched topics. Ghimire et al. compile a comprehensive survey on federated learning for cybersecurity in relation to IoT devices [28]. The survey discusses a variety of topics, primarily focusing on applications of federated learning in cybersecurity, performance issues associated with federated learning, current research, and available datasets. They present a detailed study on federated learning models used for cybersecurity and their associated performance metrics and challenges. They discuss the limitations of federated learning in the IoT space, such as limited device memory, battery power, and computing power. In summary, it appears to be the most recent survey of all things federated learning in relation to cybersecurity and IoT devices.

Rey et al. use federated learning to detect malware in IoT devices [30]. The researchers perform both supervised and unsupervised federated learning using N-BaIoT, a dataset compiled in 2018 containing attack data from nine IoT devices infected by Mirai and Bashlite (see Section 4.1.2). They achieve accuracies, True Positive Rates (TPR), and True Negative Rates (TNR) of +99% in their Multi-Epoch and Mini-Batch supervised models. They achieve similar results for their unsupervised models, obtaining a TPR of +99% and TNR between 91%-96% using Multi-Epoch and Mini-Batch averaging. They also perform benign label flipping, attack label flipping, all label flipping, gradient factor, and model canceling attacks and show that while averaging proves to be the best aggregation function in these scenarios, they are not entirely resilient.

This is the most closely related paper to our research, performing similar classifications and attacks while achieving impressive results. However, there are a few key differences between Rey et al. and our research. Two of the most important are first, we are using an entirely different dataset. Rey et al. use N-BaIoT, which contains attack data from two botnets. We are using MedBIoT, a more recent dataset that contains propagation data from three botnets. Second, we are examining our instances on a packet-by-packet basis similar to how an intrusion detection system would. We do not use statistics collected from the packet streams, but rather examine each packet and its contents independently. The dataset Rey et al. contains statistics such as the weight of a packet stream, the magnitude between two streams, the covariance between two streams, etc. We look at the raw data such as IP header length, time to live, and other features described in Table 5.1.

Galvez et al. present a malware classifier leveraging federated learning for Android applications called LiM ('Less is More') [31]. LiM uses a safe semi-supervised learning ensemble (SSL) to maximize accuracy with respect to a baseline classifier on the cloud by ensuring unlabeled data does not worsen the performance of a fully supervised classifier. Using AndroZoo dataset, the researchers show that the cloud server is able to achieve an f1 score of 95% and that clients produce a perfect recall with only one false positive. In addition, the researchers also test poisoning and inference attacks on their model and show that it is resistant to such malicious behavior.

Prior to [31], Hsu et al. present a privacy-preserving federated learning (PPFL) android malware detection system [32]. Implementing the PPFL using support vector machines, the researchers demonstrate its feasibility using an Android malware dataset by the National Institute of Information and Communication Technology (NCIT) containing over 87,000 APK files from the Opera Mobile Store. Using secure multi-party computation (SMPC), which lets multiple participants combine private inputs, the mobile devices can utilize edge devices to collaboratively train a global model without leaking information. The researchers show that their federated model can achieve a higher accuracy than their localized model, as well as being comparable to their centralized model. They also show that accuracy increases as the number of clients increases.

Zhao et al. propose a multi-task deep neural network in federated learning (MT-DNN-FL) to perform network anomaly detection tasks, VPN (Tor) traffic recognition tasks, and traffic classification tasks simultaneously [33]. They use three datasets: CICIDS2017, ISCXVPN2016, and ISCXTor2016. When compared to baseline centralized methods like deep neural networks, logistic regression, KNNs, Decision Trees, and Random forests, MT-DNN-FL performs better by achieving precision and recall scores between 94%-99%. The multi-task methods also reduce training time.

Chapter 3

Proposed Cross-Device FL IDS Architecture

The following sections discuss a holistic overview of the thesis project. We provide a broad overview of the components of a standard IDS, followed by how botnets could normally be detected on a by-packet basis. We then briefly summarize our federated model IDS, as well as the poisoning attack we plan to launch against it.

3.1 Components of the IDS

An intrusion detection system is a software application used to monitor networks for malicious traffic. A standard IDS can be broken up into various types, such as network-based, host-based, signature-based, and anomaly-based [34]. Network-based IDS analyze network traffic, host-based analyze operating system files, signature-based analyze patterns/signatures of malware, and anomaly-based analyze unknown attacks using machine learning. Figure 3.1 displays a general IDS as it is most widely used in the market. When a router connects to the internet, a firewall is commonly put up to block specified malicious traffic. An IDS can be placed before or after a firewall, but it is optimal to put it after so the firewall takes the brunt of the action and the IDS can catch malicious stragglers the firewall may miss. Then whatever traffic passes through the IDS is allowed to enter the private network.

Anomaly-based IDS solutions model "normal" behavior within the system, labeling packets as potential threats if they present anomalous behavior. This is beneficial for a closed

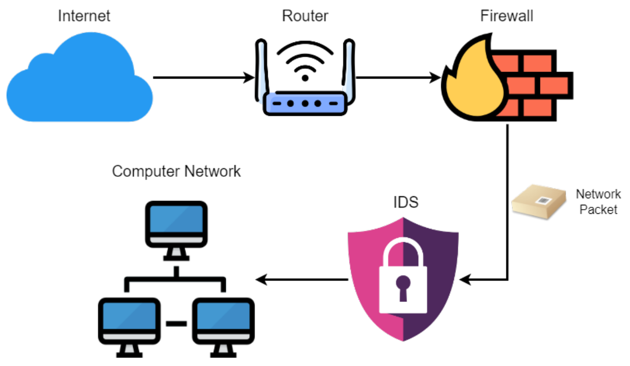


Figure 3.1: Standard IDS Setup

or private organizational network where there could be such a thing as "normal" traffic behavior, but this becomes difficult to pinpoint as open networks and IoT devices grow in size. We propose a slightly different architecture.

3.2 Detecting Botnet Propagation

As discussed in Section 2.1.3, the phases of the botnet lifecycle are initial injection, secondary injection, connection, performance, and maintenance. During the initial and secondary injection, the host device is infected by either weak credentials or system vulnerabilities. For example, Mirai would infect devices by performing a brute force attack at guessing common or insecure credentials, such as "admin" and "admin" for username and password. Mirai had a list of 60 common usernames and passwords, and would randomly to chose 10 from that list. Bashlite would infect devices via Shellshock, a bash vulnerability that allowed the execution of arbitrary commands to gain unauthorized access when concatenated to the end of function definitions. Torii is more sophisticated in the sense it will download different binary payloads based on the architecture of the targeted device. More sophisticated botnets are sneaky when infecting devices, and it is difficult to tell when a device has been infected. After infection, the device is then connected to the C&C, and it awaits instruction. Once it receives the instruction to do so, it can then carry out attacks.

The ability to detect the early botnet phases before receiving instruction from the C&C to attack is important. We believe analyzing the individual network packets may be one of the best ways to do this. If we can identify communication traffic, we can more likely identify the C&C and put up defenses against it. To analyze the individual packets, we examine the information stored in the packet headers. The packet attributes we initially analyzed using Wireshark are shown in Table 3.1. Section 4.3 and Table 5.1 discuss the packets we ended up using as predictors given their data type and easy ability to encode. We believe a system that analyzes on a per-packet basis is a beneficial way to detect discrepancies between normal and malicious traffic.

3.3 Federated Machine Learning Approach

Figure 3.2 displays a holistic overview of our proposed architecture. The components of the architecture are listed below:

- **Federated IDS:** The intrusion detection systems that utilizes federated learning to differentiate between anomalous and normal network data on a per-packet basis.
- **Devices:** Unlike the standard IDS system, our IDS would be able to connect to multiple devices on multiple networks. The devices would train our model, then report the model weights back to the IDS without having to share data.
- **Internet Routers:** What allows the devices to connect to the internet.

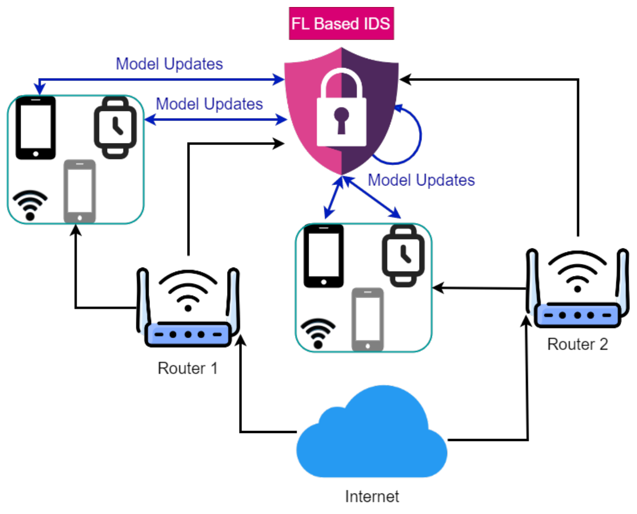


Figure 3.2: Federated IDS Setup

We want our intrusion detection system to use federated learning to detect anomalous traffic. There are a few important key points to this. First, we want to use federated learning because it conserves data privacy. In theory, our IDS would be connected to different routers, therefore being connected to different networks. Within each network, the IDS should be able to randomly select any device connected to that network for model training. The central IDS or server will distribute its model to the selected devices, and each device will run the model on its own network data. Devices would not have to share their network data with other devices or the IDS. The devices do not interact with each other, but rather send their updated model weights back to the IDS for it to average and distribute to a new randomly selected set of devices.

As mentioned briefly in Section 2.2.2, federated learning is most popularly explained in the medical setting given its ability to keep patient data confidential between different organizations. But why would we want to keep network data private to the device? Packets hold an array of information that could be used to cause harm if it falls into the wrong hands. A good example is IP addresses. Anyone can use your IP address to track and follow you around if not kept private. Based on the destination address you send information to, anyone could also exploit you via your browsing history and internet activity. People can also sniff traffic if the communication channels are not secure. As a result, we don't want to pass or transport any raw data past the device, just the model weights to conserve privacy and confidentiality.

Second, we want to use federated learning to aggregate more information from multiple networks. A typical IDS sees all the network traffic coming into a network. Our approach allows us to connect to multiple networks, allowing us to process more data without actually seeing the data. More data from different networks allows us to build better models because they will have more content to learn from. Third, the nature of anomaly detection will be more robust. Rather than building a model to recognize "normal" traffic and flag everything that does not fit that standard, we also want our model to recognize hard-to-detect malicious traffic. For example, during botnet propagation and C&C communication, it is very easy for that traffic to blend in with normal traffic. While it is good to be able to predict attack traffic, ideally we would like to take a more proactive approach to recognize anomalies before harm can be done. Especially since the main attack mechanism of botnets is DDoS attacks, there is not much someone can do to mitigate that circumstance in the heat of the moment. That is why to start small, we stuck with a dataset that only contains propagation and C&C communication data, rather than using a more popular dataset that contains physical attack data from botnets and other forms of malware.

In summary, we propose an IDS that should be able to randomly select devices from multiple networks to learn how to detect anomalies. To avoid exposing private information across devices or through the internet, all data remains local to the device, and only the model weights are updated and adjusted. We take the approach of trying to differentiate between "normal" and "anomalous" traffic since normal traffic may be hard to define in larger networks that are open or have many devices connected to them. We propose training our model on pre-attack data to help it focus on learning the discrepancies between malicious and benign network traffic when they are hard to tell apart.

3.4 Poisoning Attack by Participant Node

Poisoning attacks happen when a hacker tries to inject fake training data into a model to reduce or hinder its performance. There are four main categories of poisoning attacks: 1) logic corruption, 2) data manipulation, 3) data injection, and 4) Domain Name System (DNS) cache poisoning [35]. Logic corruption is where the attacker changes the logic of the systems to disrupt how the system learns [35]. Data manipulation is where the attacker manipulates the data rather than the logic [35]. Data injection is when fake data is inserted into the actual dataset to skew model results and weaken outcomes [35]. Last, DNS cache poisoning is when the attacker corrupts DNS data, causing the name server to return incorrect results [35]. The type of poisoning attack we try to replicate is a data manipulation attack. Since we are using supervised learning (see Section 2.2.1), we emulate an attack as if a malicious user switched the malware labels to benign to cover up bad traffic. We tried to be sophisticated with our approach using label-flipping. We researched the most popular source port among the malicious data in the dataset and found port 23 to be on top. As a result, we flipped all the labels from 1 to 0 for packets with a source port equal to 23 in the first client.

Table 3.1: Original Wireshark Keys

Key	Description	Type
frame.encap_type	Frame Encapsulation Type	Signed Integer (2 Bytes)
frame.time	Frame Arrival Time	Date and Time
frame.time_epoch	Epoch Time	Time Offset
frame.offset_shift	Time Shift for This Packet	Time Offset
frame.time_delta	Time Delta from Previous Captured Frame	Time Offset
frame.time_delta_displayed	Time Delta from Previous Displayed Frame	Time Offset
frame.time_relative	Time Since Reference of First Frame	Time Offset
frame.number	Frame Number	Unsigned Integer (4 Bytes)
frame.len	Frame Length on The Wire	Unsigned Integer (4 Bytes)
frame.cap_len	Frame Length Stored in Capture File	Unsigned Integer (4 Bytes)
frame.marked	Frame is Marked	Boolean
frame.ignored	Frame is Ignored	Boolean
frame.protocols	Protocols in Frame	Character String
frame.coloring_rule.name	Coloring Rule Name	Character String
eth.dst	Destination Address	Ethernet or Other MAC Address
eth.src	Source Address	Ethernet or Other MAC Address
eth.type	Protocol Field	Unsigned Integer (2 Bytes)
ip.dsfield	Differentiated Service Field	Unsigned Integer (1 Byte)
ip.len	IP Total Length	Unsigned Integer (2 Bytes)
ip.id	IP Identification	Unsigned Integer (2 Bytes)
ip.flags	IP Flags	Unsigned Integer (1 Byte)
ip.ttl	Time to Live	Unsigned Integer (1 Byte)
ip.proto	IP Protocol	Unsigned Integer (1 Byte)
ip.checksum	Header Checksum	Unsigned Integer (2 Bytes)
ip.checksum.status	Header Checksum Status	Unsigned Integer (1 Byte)
ip.src	Source Address	IPv4 Address
ip.dst	Destination Address	IPv4 Address
tcp.srcport	Source Port	Unsigned Integer (2 Bytes)
tcp.dstport	Destination Port	Unsigned Integer (2 Bytes)
tcp.stream	Stream Index	Unsigned Integer (4 Bytes)
tcp.len	TCP Segment Length	Unsigned Integer (4 Bytes)
tcp.seq	Sequence Number	Unsigned Integer (4 Bytes)
tcp.seq_raw	Raw Sequence Number	Unsigned Integer (4 Bytes)
tcp.nxtseq	Next Sequence Number	Unsigned Integer (4 Bytes)
tcp.ack	Acknowledgement Number	Unsigned Integer (4 Bytes)
tcp.ack_raw	Raw Acknowledgement Number	Unsigned Integer (4 Bytes)
tcp.hdr_len	Header Length	Unsigned Integer (1 Byte)
tcp.flags	Flags	Unsigned Integer (2 Bytes)
tcp.window_size_value	Window	Unsigned Integer (2 Bytes)
tcp.window_size	Calculated Window Size	Unsigned Integer (4 Bytes)
tcp.window_size_scalefactor	Window Size Scaling Factor	Unsigned Integer (4 Bytes)
tcp.checksum	Checksum	Unsigned Integer (2 Bytes)
tcp.checksum.status	Checksum Status	Unsigned Integer (1 Byte)
tcp.urgent_pointer	Urgent Pointer	Unsigned Integer (2 Bytes)
tcp.time_relative	Time Since First Frame in This TCP Stream	Time Offset
tcp.time_delta	Time Since Previous Frame in This TCP Stream	Time Offset
tcp.analysis.bytes_in_flight	Bytes in Flight	Unsigned Integer (4 Bytes)
tcp.analysis.push_bytes_in_flight	Bytes Sent Since Last PSH Flag	Unsigned Integer (4 Bytes)
udp.srcport	Source Port	Unsigned Integer (2 Bytes)
udp.dstport	Destination Port	Unsigned Integer (2 Bytes)
udp.length	Length	Unsigned Integer (2 Bytes)
udp.checksum	Checksum	Unsigned Integer (2 Bytes)
udp.time_relative	Time Since First Frame	Time Offset
udp.time_delta	Time Since Previous Frame	Time Offset

Chapter 4

Processing Packet Captures to Dataset for Machine Learning

This chapter goes over the different publicly available datasets commonly used by researchers in the cybersecurity field. We talk about why we chose MedBIoT, and our process of converting all the pcap files to csv format to generate a single dataset.

4.1 Exploring Available Datasets

The following subsections discuss the most common published datasets relating to malware network traffic. We summarize each dataset before explaining why we chose to move forward with MedBIoT.

4.1.1 CTU-13

CTU-13 consists of real botnet traffic mixed with normal/background traffic [36]. This traffic can be separated into 13 captures (i.e. scenarios) of different botnet samples. These botnet samples include Neris, Rbot, Virut, Menti, Sogou, Murlo, and NSIS.ay. Since it was created in 2014, it does not contain IoT device data or IoT-specific botnets such as Mirai. The data is stored in pcap files. In summary, CTU-13 is a dataset consisting of 13 scenarios covering botnet data predating Mirai and other IoT botnets.

4.1.2 N-BaIoT

N-BaIoT contains real traffic data gathered from nine IoT devices infected by Mirai and Bashlite [37]. These devices are a Danmini Doorbell, Ecobee Thermostat, Ennio Doorbell, Philips B120N10 Baby Monitor, Provision PT-737E Security Camera, Provision PT-838 Security Camera, Samsung SNH 1011N Webcam, SimpleHome XCS7-1002-WHT Security Camera, and SimpleHome XCS7-1003-WHT Security Camera. The malicious portion of the data is divided into 10 different attacks carried out by the two botnets. Containing 7,062,606 instances and 115 features in total, the attribute information can be grouped by stream aggregation statistics, time frame (The decay factor Lambda used in the damped window), and statistics extracted from the packet stream (i.e. weight, mean, standard deviation, radius, magnitude, covariance, and Pearson correlation). In summary, this dataset contains statistics in csv format from 10 attacks by two botnets on nine IoT devices.

4.1.3 Kitsune

Kitsune includes nine different attacks on a commercial IP-based surveillance system and IoT network [36]. These attacks are OS Scan, Fuzzing, Video Injection, ARP MitM, Active Wiretap, SSDP Flood, SYN DoS, SSL Renegotiation, and Mirai exploitation and scanning. The dataset contains the preprocessed csv with 115 features extracted using their AfterImage feature extractor, providing a statistical snapshot of the network. The dataset also contains the raw pcap files. The botnet data makes up a small portion of the dataset, consisting of 764,137 out of 27,170,754 instances. In summary, Kitsune contains the statistical data and network captures associated with 9 different attacks, one relating to botnets (Mirai).

4.1.4 IoT-23

IoT-23 is a labeled dataset with malicious and benign IoT traffic, 20 of which are malware captures, and three of which are benign captures [38]. The 23 captures are known as scenarios and are divided into 20 pcap files from the infected devices. Each malicious scenario executes a malware sample in a Raspberry Pi (simulated). The three benign scenarios were obtained using a Philips HUE smart LED lamp, an Amazon Echo home intelligent personal assistant, and a Somfy smart doorlock (real). It includes various forms of malware, such as Mirai, Torii, Trojan, Gagfyt, Kenjiro, Okiru, Hakai, IRCBot, Hajime, Muhstik, and Hide and Seek (HNS). In summary, IoT-23 contains a mix of 23 real and simulated network traffic encompassing a variety of different malware in pcap format.

4.1.5 MedBIoT

MedBIoT contains data focusing on the early stages of botnet deployment: propagation and C&C communication [3]. A combination of real and emulated devices are used, making up for a total of 83 devices. The real devices include a Sonoff Tasmota smart switch, TPLink smart switch, and a TPLink light bulb. The Emulated devices include locks, switches, fans, and lights. These devices are infected with the Mirai, Bashlite, and Torii botnet malware. The data is stored in two formats, one being the raw pcap files and the other being the structured statistics in csv format. In summary, this dataset contains both pcaps and statistics in csv format of the propagation and C&C communication of three botnets over 83 devices. We use the raw pcap files in order to analyze the packets independently of one another.

4.1.6 X-IIoTID

X-IIoTID contains data from the Industrial Internet of Things (IIoT), including the behavior of new IIoT connectivity protocols, the activity of recent devices, and diverse attack protocols [39]. The dataset consists of multi-view features such as network traffic, host resources, logs, and alerts from various attacks. These attacks include generic scanning, scanning vulnerabilities, WebSocket fuzzing, discovering CoAP resources, brute force attacks, dictionary attacks, malicious insider, reverse shell and Man-in-the-Middle, MQTT

Table 4.1: Surface-Level Comparison of Available Datasets

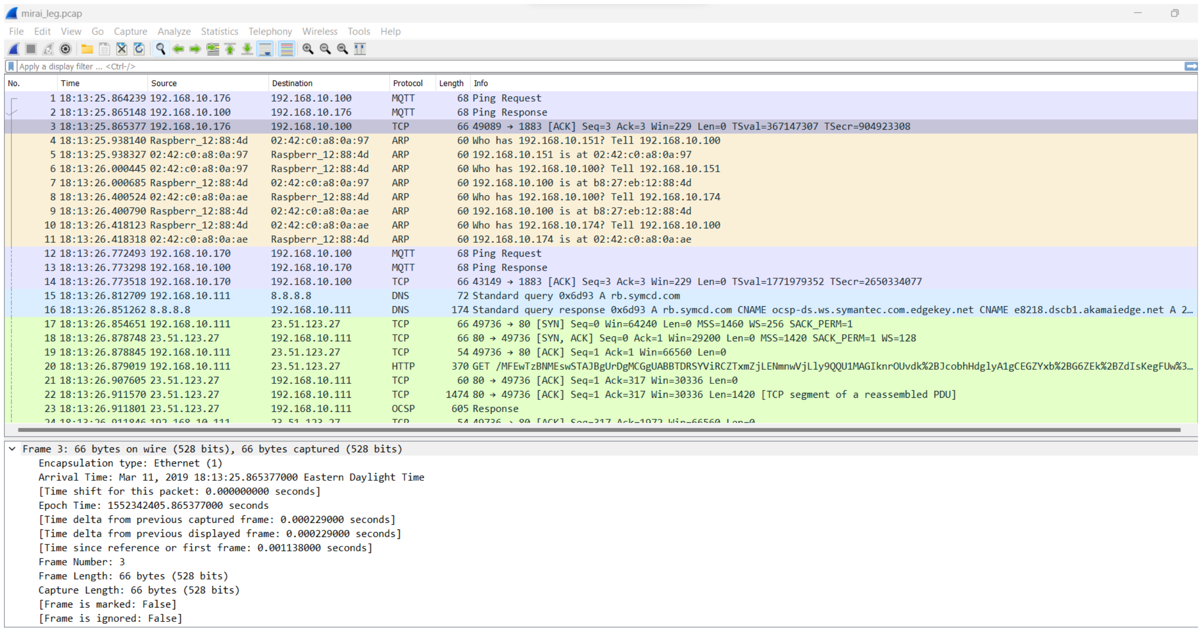
Dataset Name	Month	Year	Description	Format	Publication
X-IIoTID	August	2021	Intrusion dataset for Industrial Internet of Things (IIoT)	CSV	[39]
MedBIot	February	2020	Early stages of botnet deployment: spreading and C&C communication	PCAP	[3]
IoT-23	January	2020	Labeled dataset with malicious and benign IoT network traffic	PCAP	[38]
Kitsune	October	2019	Collection of nine network attack datasets	CSV	[36]
N-BaIoT	May	2018	Traffic data gathered from nine commercial IoT devices	CSV	[37]
CTU-13		2014	Labeled dataset with botnet, normal and background traffic	PCAP	[40]

cloud broker-subscription, Modbus-register reading, TCP Relay attacks, command and control, data exfiltration, poisoning of cloud data (i.e., false data injections), fake notification crypto-ransomware, and ransom denial of service attack. It contains 820,834 instances and 68 features in csv format. In summary, X-IIoTID contains various attack data from Industrial Internet of Things devices.

4.2 Choosing MedBIot

We chose to move forward with MedBIoT for a variety of reasons. Originally we tried to run the Mirai source code on our own using virtual machines to capture real-time network traffic. We faced many issues with this, including issues with running the source code, setting up the virtual machines, incompatibilities between Microsoft and Linux distributions and lack of Microsoft certificates, and operating system version incompatibilities. As a result, we had to switch gears and search for datasets containing Mirai traffic. We liked MedBIoT because not only did it contain real-time traffic data using Mirai, but it also contained data from two other botnets (Bashlite and Torii) that many variants base their source code on. The data is compiled of network traffic from multiple emulated and real IoT devices, does not contain data from any other forms of malware, and only contains propagation and C&C communication traffic. Botnets are deemed to be most vulnerable during communication with the C&C because this stage allows botnet mitigators to identify traffic patterns and identify components of the botnet or the C&C [8]. We were not interested in data containing attack traffic, so MedBIoT was the most applicable to our research area and proved to be a better route than continuing to struggle to emulate a less in-depth network.

MedBIoT provides the pcaps in two main formats: bulk and fine-grained. The bulk data is separated by source type (i.e legitimate, Mirai, Bashlite, and Torii). The fine-grained data is separated by each data source, botnet phase, and device type. The pcaps are in the format SOURCE-TYPE_TRAFFIC-TYPE_PHASE_DEVICE.pcap. For example, mirai_mal_CC_lock.pcap would be the name of one of the files, with “mirai” being the source type, “mal” being the traffic type, “CC” being the phase, and “lock” being the device.


mirai_leg.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.   Time            Source              Destination         Protocol  Length Info
    1 18:13:25.864239 192.168.10.176      192.168.10.100      MQTT        68 Ping Request
    2 18:13:25.865148 192.168.10.100      192.168.10.176      MQTT        68 Ping Response
    3 18:13:25.865377 192.168.10.176      192.168.10.100      TCP         66 49089 → 1883 [ACK] Seq=3 Ack=3 Win=229 Len=0 TSval=367147307 TSecr=904923308
    4 18:13:25.938140 Raspberr_12:88:4d   02:42:c0:a8:0a:97   ARP         60 Who has 192.168.10.151? Tell 192.168.10.100
    5 18:13:25.938327 02:42:c0:a8:0a:97   Raspberr_12:88:4d   ARP         60 192.168.10.151 is at 02:42:c0:a8:0a:97
    6 18:13:26.000445 02:42:c0:a8:0a:97   Raspberr_12:88:4d   ARP         60 Who has 192.168.10.100? Tell 192.168.10.151
    7 18:13:26.000685 Raspberr_12:88:4d   02:42:c0:a8:0a:97   ARP         60 192.168.10.100 is at b8:27:eb:12:88:4d
    8 18:13:26.400524 02:42:c0:a8:0a:ae   Raspberr_12:88:4d   ARP         60 Who has 192.168.10.100? Tell 192.168.10.174
    9 18:13:26.400790 Raspberr_12:88:4d   02:42:c0:a8:0a:ae   ARP         60 192.168.10.100 is at b8:27:eb:12:88:4d
   10 18:13:26.418123 Raspberr_12:88:4d   02:42:c0:a8:0a:ae   ARP         60 Who has 192.168.10.174? Tell 192.168.10.100
   11 18:13:26.418318 02:42:c0:a8:0a:ae   Raspberr_12:88:4d   ARP         60 192.168.10.174 is at 02:42:c0:a8:0a:ae
   12 18:13:26.772493 192.168.10.170      192.168.10.100      MQTT        68 Ping Request
   13 18:13:26.773298 192.168.10.100      192.168.10.170      MQTT        68 Ping Response
   14 18:13:26.773518 192.168.10.170      192.168.10.100      TCP         66 43149 → 1883 [ACK] Seq=3 Ack=3 Win=229 Len=0 TSval=1771979352 TSecr=2650334077
   15 18:13:26.812709 192.168.10.111      8.8.8.8             DNS         72 Standard query 0x6d93 A rb.symcd.com
   16 18:13:26.851262 8.8.8.8             192.168.10.111      DNS        174 Standard query response 0x6d93 A rb.symcd.com CNAME ocsp-ds.ws.symantec.com.edgekey.net CNAME e8218.dscb1.akamaiedge.net A 2…
   17 18:13:26.854651 192.168.10.111      23.51.123.27        TCP         66 49736 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
   18 18:13:26.878748 23.51.123.27        192.168.10.111      TCP         66 80 → 49736 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1420 SACK_PERM=1 WS=128
   19 18:13:26.878845 192.168.10.111      23.51.123.27        TCP         54 49736 → 80 [ACK] Seq=1 Ack=1 Win=66560 Len=0
   20 18:13:26.879019 192.168.10.111      23.51.123.27        HTTP       370 GET /MFEwTzBNMEswSTAJBgUrDgMCGgUABBTDRSYViRCZTxmZjLENmnwVjLly9QQU1MAGIknrOUvdk%2BJcobhHdglyA1gCEGZYxb%2BG6ZEk%2BZdIsKegFUw%3…
   21 18:13:26.907605 23.51.123.27        192.168.10.111      TCP         60 80 → 49736 [ACK] Seq=1 Ack=317 Win=30336 Len=0
   22 18:13:26.911570 23.51.123.27        192.168.10.111      TCP       1474 80 → 49736 [ACK] Seq=1 Ack=317 Win=30336 Len=1420 [TCP segment of a reassembled PDU]
   23 18:13:26.911801 23.51.123.27        192.168.10.111      OCSP       605 Response
   24 18:13:26.911846 192.168.10.111      23.51.123.27        TCP         54 49736 → 80 [ACK] Seq=317 Ack=1972 Win=66560 Len=0
```

```
˅ Frame 3: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
    Encapsulation type: Ethernet (1)
    Arrival Time: Mar 11, 2019 18:13:25.865377000 Eastern Daylight Time
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1552342405.865377000 seconds
    [Time delta from previous captured frame: 0.000229000 seconds]
    [Time delta from previous displayed frame: 0.000229000 seconds]
    [Time since reference or first frame: 0.001138000 seconds]
    Frame Number: 3
    Frame Length: 66 bytes (528 bits)
    Capture Length: 66 bytes (528 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
```

Figure 4.1: Wireshark Screen

4.3 Converting pcap to csv using Wireshark and tshark

We needed to convert the pcap files to csv. To do this, we used Wireshark and tshark. Wireshark is a network protocol analyzer that lets individuals view incoming and outgoing packets on a network. Terminal Wireshark (tshark) is the command line version of Wireshark we used to convert the pcap files to csv. Using Wireshark, we opened a few of the pcap files, selected a few different packets, and manually went through all the packet details (see Figure 4.1) and captured each field name. To do this, we just selected the packet description field (i.e. Epcoh Time) and copied its associated field name (i.e. frame.time_epoch). All the field names we initially selected and viewed are shown in Table 3.1.

We first generated a large, singular dataset from the bulk data containing all the features, then randomly selected a little over 3,700 rows to produce a subset dataset to gain a more holistic view of the values each feature contained. After analyzing these features in the subset, we decided to discard a handful of them. We discarded features for a few reasons. One reason was the large dataset was too big to process with that many features and millions of rows. The other reason is due to the nature of the features. Table 4.2 shows which features were discarded and their associated reason for removal. The reasons for removal include:

- **Lack of Variability:** This feature produced little variability. For example, frame.encap_type only contained the number "1" for each data point.
- **Date/Time:** This feature produced dates and/or times, constituting less-valuable information. For example, frame.time displays the date and time the packet was captured.

Table 4.2: Reason for Removing Features

Key	Reason for Removal
frame.encap_type	Lack of Variability
frame.time	Date/Time
frame.time_epoch	Continuous
frame.offset_shift	Lack of Variability
frame.time_delta	Continuous
frame.time_delta_displayed	Interchangeable (frame.time_delta)
frame.time_relative	Continuous
frame.number	Continuous
frame.cap_len	Interchangeable (frame.len)
frame.marked	Lack of Variability
frame.ignored	Lack of Variability
frame.coloring_rule.name	N/A
eth.dst	Categorical
eth.src	Categorical
eth.type	Lack of Variability
ip.dsfield	Lack of Variability
ip.id	Identification
ip.checksum	Variability
ip.checksum.status	Lack of Variability
tcp.stream	Identification
tcp.seq	Variability
tcp.seq_raw	Variability
tcp.nxtseq	Variability
tcp.ack	Variability
tcp.ack_raw	Variability
tcp.checksum	Variability
tcp.checksum.status	Lack of Variability
tcp.urgent_pointer	Lack of Variability
udp.srcport	Nulls
udp.dstport	Nulls
udp.length	Nulls
udp.checksum	Nulls
udp.time_relative	Nulls
udp.time_delta	Nulls

- **Continuous:** This feature produced continuous variables that would be difficult to encode. An example would be frame.time_epoch.
- **Interchangeable:** This feature had the same values as other features. For example, frame.time_delta_displayed contained the same data as frame.time_delta.
- **Variability:** This feature produced too much variability to recognize a pattern in the data. For example, tcp.seq are randomly generated values identifying packet order.
- **N/A:** The feature is not applicable. For example, frame.coloring_rule.name is only used to differentiate color-coded packets in Wireshark.
- **Identification:** This feature produced the ID of a packet, which is unique to each packet. An example would be ip.id.
- **Nulls:** This feature produced too many null values to be considered worth keeping. An example would be udp.checksum.

Table 4.3: Wireshark Keys Kept

Key	Description	Type
frame.len	Frame Length on The Wire	Unsigned Integer (4 Bytes)
frame.protocols	Protocols in Frame	Character String
ip.len	IP Total Length	Unsigned Integer (2 Bytes)
ip.flags	IP Flags	Unsigned Integer (1 Byte)
ip.ttl	Time to Live	Unsigned Integer (1 Byte)
ip.proto	IP Protocol	Unsigned Integer (1 Byte)
ip.src	Source Address	IPv4 Address
ip.dst	Destination Address	IPv4 Address
tcp.srcport	Source Port	Unsigned Integer (2 Bytes)
tcp.dstport	Destination Port	Unsigned Integer (2 Bytes)
tcp.len	TCP Segment Length	Unsigned Integer (4 Bytes)
tcp.hdr_len	Header Length	Unsigned Integer (1 Byte)
tcp.flags	Flags	Unsigned Integer (2 Bytes)
tcp.window_size_value	Window	Unsigned Integer (2 Bytes)
tcp.window_size	Calculated Window Size	Unsigned Integer (4 Bytes)
tcp.window_size_scalefactor	Window Size Scaling Factor	Unsigned Integer (4 Bytes)
tcp.time_relative	Time Since First Frame in This TCP Stream	Time Offset
tcp.time_delta	Time Since Previous Frame in This TCP Stream	Time Offset
tcp.analysis.bytes_in_flight	Bytes in Flight	Unsigned Integer (4 Bytes)
tcp.analysis.push_bytes_in_flight	Bytes Sent Since Last PSH Flag	Unsigned Integer (4 Bytes)

Once we figured out which features to remove, we then converted the individual pcaps in the fine-grained dataset using the desired features shown in Table 4.3. We used tshark to convert these fine-grained pcaps to csv. The code to do this is shown below.

```python
import os
import csv
import shutil

# loop throught malware and normal directories
dirs = ['malware', 'normal']
# create a new directory to store the csv files
os.mkdir('fine_grained_csv')
# path
path = '.../fine-grained/raw_dataset'
# loop through files in directory
for file in os.listdir():
    # select only pcap files
    if file.endswith(".pcap"):
        # tshark command to convert pcap to csv
        command = f'tshark -r {file} -T fields -E header=y -E separator=, -E quote=d -E occurrence=f -e frame.len -e frame.protocols -e ip.len -e ip.flags -e ip.ttl -e ip.proto -e ip.src  -e ip.dst -e tcp.srcport -e tcp.dstport -e tcp.len -e tcp.hdr_len -e tcp.flags -e tcp.window_size_value -e tcp.window_size -e tcp.window_size_scalefactor -e tcp.time_relative -e tcp.time_delta -e tcp.analysis.bytes_in_flight -e tcp.analysis.push_bytes_sent > {file[:-5]}.csv'
        # run the tshark command
        os.system(command)
        # move newly created csv to new directory
        shutil.move(f'{file[:-5]}.csv', f'{path}/fine_grained_csv')
```

The csv files produced carried the same naming conventions as the pcap files. For example, mirai_mal_CC_lock.pcap became mirai_mal_CC_lock.csv. Now that we had the features we wanted, we then added a few additional features to use as labels.

4.4 Building the Dataset

The main thing we wanted to be able to predict was whether traffic was malicious or not. If we had time, we also wanted to predict the type of botnet malware and the device type. While we were not able to complete these last two predictions, we still created labels for all of them. The first feature we modified was frame.protocols. These were long character strings, so we split up the string and kept the last element. For example, if the data point was "eth:ethertype:ip:tcp" we manipulated it such that only "tcp" was kept.

```
def split_protocol(proto):
    proto = proto.split(':')
    proto = proto[len(proto)-1]
    return proto[:-1]
```

After splitting the protocol, we then created various labels. The first label we created was used to display whether a packet was malicious or not. Based on the file name, we created a feature called is_malware, and labeled it "1" if the file name contained "mal", and "0" if it contained "leg".

```
def is_malware(file_name):
    n = file_name.split('_')
    if n[1] == 'mal':
        return 1
    else:
        return 0
```

After creating is_malware, we then created a label differentiating the botnet's action. To do this, we first determined whether the file was malicious or legitimate. Then, based on the file name, we labeled it as "spread" for propagation traffic, and "cc" for C&C communication traffic. Note that all the Torii files only housed propagation data given the researchers did not want to risk communication with Torii's real C&C server.

```
def define_phase(file_name):
    n = file_name.split('_')
    if n[1] == 'mal':
        if n[0] == 'torii':
            return 'spread'
        elif n[2] == 'spread':
            return 'spread'
        elif n[2] == 'CC':
            return 'cc'
    else:
        return 'leg'
```

Once we created a feature for the botnet phase, we then created a feature for the device type used. The types of devices included a fan, switch, lock, light, and two raspberry pis.

Table 4.4: Number of Row for Each Dataset

Modified Dataset (Individualized)	Number of Rows	Device Dataset	Number of Rows
bashlite_mal_CC_fan_mod.csv	188,853	mal_lock_data.csv	1,068,829
bashlite_mal_CC_light_mod.csv	167,285		
bashlite_mal_CC_lock_mod.csv	157,141		
bashlite_mal_CC_switch_mod.csv	114,223	mal_fan_data.csv	1,775,401
bashlite_mal_spread_fan_mod.csv	1,032,604		
bashlite_mal_spread_light_mod.csv	900,135		
bashlite_mal_spread_lock_mod.csv	761,799	mal_light_data.csv	1,232,180
bashlite_mal_spread_switch_mod.csv	589,858		
mirai_mal_CC_fan_mod.csv	296,643		
mirai_mal_CC_light_mod.csv	93,177	mal_switch_data.csv	1,110,148
mirai_mal_CC_lock_mod.csv	56,785		
mirai_mal_CC_switch_mod.csv	106,023		
mirai_mal_spread_fan_mod.csv	119,828	mal_rasp1_data.csv	46,223
mirai_mal_spread_light_mod.csv	71,236		
mirai_mal_spread_lock_mod.csv	92,950		
mirai_mal_spread_switch_mod.csv	162,634	mal_rasp2_data.csv	2,639
torii_mal_fan_mod.csv	137,477		
torii_mal_light_mod.csv	351		
torii_mal_lock_mod.csv	158	leg_lock_data.csv	12,847,387
torii_mal_raspberry1_mod.csv	46,223		
torii_mal_raspberry2_mod.csv	2,639		
torii_mal_switch_mod.csv	137,414	leg_fan_data.csv	1,449,712
bashlite_leg_fan_mod.csv	1,079,670		
bashlite_leg_light_mod.csv	976,966		
bashlite_leg_lock_mod.csv	12,572,305	leg_light_data.csv	1,262,686
bashlite_leg_switch_mod.csv	1,067,072		
mirai_leg_fan_mod.csv	293,099		
mirai_leg_light_mod.csv	263,845	leg_switch_data.csv	1,319,527
mirai_leg_lock_mod.csv	264,179		
mirai_leg_switch_mod.csv	219,556		
torii_leg_fan_mod.csv	76,945	leg_rasp1_data.csv	129,551
torii_leg_light_mod.csv	21,877		
torii_leg_lock_mod.csv	10,905		
torii_leg_raspberry1_mod.csv	129,551	leg_rasp2_data.csv	4,714
torii_leg_raspberry2_mod.csv	4,714		
torii_leg_switch_mod.csv	32,901		

```python
def define_device(file_name):
    n = file_name.split('_')
    return n[-1][:-4]
```

After creating these additional labels, we transformed each fine-grained csv and added "mod" in the name. For example, mirai_mal_CC_lock.csv became mirai_mal_CC_lock_mod.csv. Table 4.4 displays the number of rows for each of these modified datasets. Once the modified datasets were generated with the additional labels, we then combined these files based on device type, but still differentiated between malicious and legitimate traffic. Table 4.4 also displays the number of rows for the device-specified datasets. After combining the datasets by device and traffic type, we then selected 2,000 random rows from each dataset and generated a singular dataset containing a total of 24,000 rows. By doing all this, we tried to keep an even amount of data as well as ensure the dataset was an appropriate size to process efficiently. We then proceeded to do some minimal data preprocessing.

Table 4.5: Number of Nulls and Unique Values for Each Feature

Attribute	# of NaN	# of Unique Values
frame_len	0	358
frame_protocols	0	13
ip_len	0	360
ip_flags	0	2
ip_ttl	0	19
ip_proto	0	5
ip_src	0	97
ip_dst	0	5024
tcp_srcport	207	5878
tcp_dstport	207	2141
tcp_len	355	337
tcp_hdr_len	354	7
tcp_flags	354	8
tcp_window_size_value	354	901
tcp_window_size	354	899
tcp_window_size_scalefactor	6393	6
tcp_time_relative	354	20942
tcp_time_delta	354	9577
tcp_analysis_bytes_in_flight	14782	1223
tcp_analysis_push_bytes_sent	14782	1339
is_malware	0	2
malware_type	0	3
device	0	6
phase	0	3

4.5 Preprocessing: Dropping More Nulls

Once we had a single dataset, we wanted to verify again how many null values we had. Using the code below, we produced statistics similar to the ones shown in Table 4.5.

```
from tabulate import tabulate
#====================== Display Count of NaN for Columns
headers = data.columns.values
null_val = []
unique_val = []
tabs = []

for h in headers:
 null_val.append(data[h].isnull().sum())
 unique_val.append(data[h].nunique())

for i in range(len(headers)):
    tabs.append([headers[i], null_val[i], unique_val[i]])

print(tabulate(tabs, headers=['Attribute', '# of NaN', '# of Unique Values
    '], tablefmt='orgtbl'))

#====================== Drop Columns
null_columns = ['tcp_len', 'tcp_hdr_len', 'tcp_flags',
            'tcp_window_size_value', 'tcp_window_size',
                'tcp_window_size_scalefactor', 'tcp_window_size',
                'tcp_time_relative', 'tcp_time_delta',
                'tcp_analysis_bytes_in_flight',
                'tcp_analysis_push_bytes_sent']

data.drop(null_columns, axis=1, inplace=True)

#====================== Drop Remaining NaN
data.dropna(inplace=True)
```

We removed all of the features containing null values since it would be difficult to fill this data in. We then performed a final dropna() to get rid of any remaining rows with null values. We ended up with a mildly unbalanced dataset containing 23,793 samples in total out of the original 24,000. Out of the 23,793 samples, 11,942 were labeled as malware, and 11,851 were labeled as benign. While we tried to keep things as balanced as possible, we didn't stress too much about this because we inevitably would have to shuffle the data and distribute it randomly amongst the clients. Each client, therefore, will more likely have unbalanced data when training their local model. In the real world, it is also unlikely each client will be processing balanced network traffic. We also did not spend time analyzing outliers during preprocessing given the nature of our data and how an intrusion detection system operates.

Chapter 5

Experiment Setup of Cross-Device FL IDS

5.1 Experimental Design Setup

Figure 5.1 outlines our final experimental design setup. We initially tried to implement our model using Google's TFF but had to switch gears and implement the model using [41]. We repeated this process for our poisoning attack with minimal changes discussed in Section 5.2.

5.1.1 Attempted Strategy Using TFF

To assist with federated learning research, Google has come up with an open-sourced framework called TensorFlow Federated (TFF). It is used in conjunction with TensorFlow (TF) and Python and is compatible with Keras to upload pre-built models. TFF is offered as a Federated Learning API, which allows developers to use existing TF models, and a Federated Core, which allows developers to create novel federated learning solutions.

Originally, we tried to analyze our data using the API. This, however, proved difficult because working with tensors and reshaping data added extra levels of complexity, converting TF methods to TFF functions was complicated, and little documentation exists on how to actually use the API. For example, most of the tutorials by google use a pre-existing TFF formatted dataset, but the tutorials do not discuss how this dataset is different from the original (e.g MNIST vs tff.simulation.datasets.emnist for image classification). As far as using our own data, there were minimal articles discussing how to manipulate and reshape the data, but even then there were slight discrepancies (i.e. using a dictionary vs an ordered dictionary) between them and the google tutorials. TFF also only works with certain versions of TF. After extensive work, trial, and error, we were able to eventually produce some output, exhibiting the same fluctuations discussed in Section 6.1 but with similar accuracy values. The metrics produced, however, were only accuracy and loss. We could not see precision, recall, and f1 scores, and the only place where these metrics were defined was in the Keras model using the metric tf.keras.metrics.BinaryAccuracy(). The way the API is set up and the data is shaped and modified, we could not figure out how to call standard python functions such as precision_score(). As a result, we changed the metrics=[tf.keras.metrics.Accuracy(), tf.keras.metrics.FalseNegatives, tf.keras.metrics.FalsePositives] to get a more holistic view of the model. Changing the metrics without adjusting anything else, however, resulted in

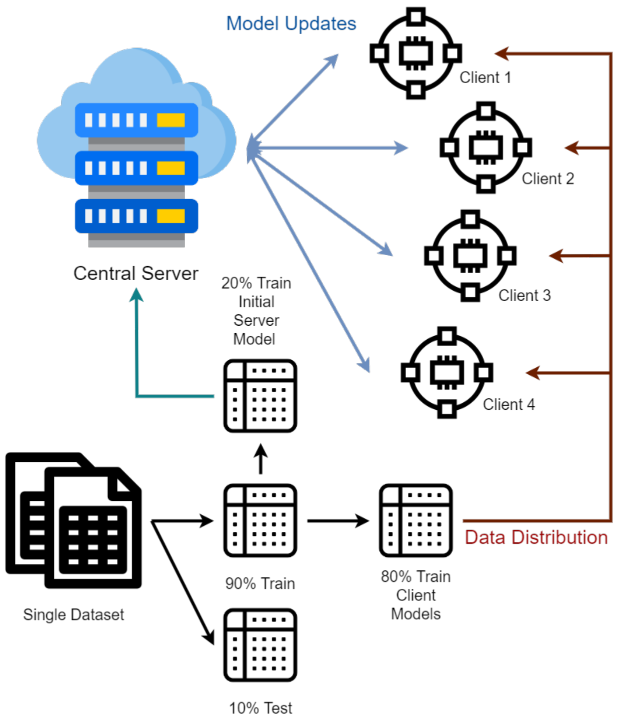


Figure 5.1: Experiment Overview

worst model performance. For example, in one experiment the model would produce 50%, and in one experiment the model would produce 3%. The model also never learned, meaning the model would guess one class for everything, resulting in one metric being zero, and one metric being however much data was processed. As a result, due to our inability to 1) get the model to produce metrics other than accuracy and loss, 2) understand why model performance was so poorly affected due to a change in metric outputs, 3) TFF being a black box method to us, and 4) time constraints, we had to quickly switch gears using another method of implementation. Moving forward, we abandoned the TFF API and proceeded to reference code from [41] used in a recent IEEE federated learning publication [42].

5.1.2 Finalized Strategy of Experiment Setup

After creating the dataset mentioned in Section 4 and removing all the nulls, we were left with the features displayed in Table 5.1. This dataset contained 11,942 malware samples and 11,851 benign samples, totaling 23,793 samples out of the original 24,000 before using dropna(). This contains an imbalance of 91 samples in favor of malware. We decided not to completely balance the data because the data would be shuffled before being distributed amongst the clients anyway. Out of the features shown in Table 5.1, the ones highlighted in green are selected as the predictors to our label is_malware given they are numerical. After defining our inputs, we separated our predictors and label, scaled the data using

Table 5.1: Final Selected Features

Feature	Type
frame_len	int64
frame_protocols	object
ip_len	int64
ip_flags	object
ip_ttl	int64
ip_proto	int64
ip_src	object
ip_dst	object
tcp_srcport	float64
tcp_dstport	float64
is_malware	int64
malware_type	object
device	object
phase	object

MinMaxScaler(), and split the training set using a test_size = 0.10 and a seed of 123 to enforce consistency throughout the experiments.

```
from sklearn import preprocessing

inputx = ['frame_len', 'ip_len', 'ip_ttl', 'ip_proto', 'tcp_srcport', '
    tcp_dstport']
X_full = data.loc[:, inputx].values
Y_full = data['is_malware'].values

scaler = preprocessing.MinMaxScaler()
X_full = scaler.fit_transform(X_full)

xTrain, xTest, yTrain, yTest = train_test_split(X_full, Y_full, test_size
    =0.10, random_state=123)
```

We saved 10% of the data for testing, resulting in a total of 2,380 samples. Out of the 2,380 testing samples, 1,186 of them are benign, and 1,194 are malicious (resulting in an imbalance of 8). This meant that 21,413 samples were used for training. After splitting, we converted the integer class vector is_malware to a binary class matrix using to_categorical(). It is debatable whether this was necessary for simple binary classification, but we implemented it because if we had time we wanted to be able to predict other labels such as botnet cycle phase and malware type. Inevitably, we did not have time to make these additional predictions. After converting the labels to a matrix, we defined the metrics we wanted to track (i.e. accuracy, precision, recall, and f1) using their associated built-in python functions. Then, using a batch size of 64, 200 epochs, ReLu as the activation function (except for the last layer, which uses SoftMax), and Adam as the optimizer, we build the simple neural network model shown below:

```
data = pd.read_csv('mod_combined_2000_processed.csv')
verbose, epochs, batch_size = 0, 200, 64
activationFun='relu'
optimizerName='Adam'
def createDeepModel():
    model = Sequential()
    model.add(Dense(6, input_dim=X_full.shape[1], activation=activationFun
   ))
    model.add(Dropout(0.4))
    model.add(Dense(4, activation=activationFun))
    model.add(Dropout(0.4))
    model.add(Dense(outputClasses, activation='softmax'))
    model.compile(loss='binary_crossentropy',
                  optimizer=optimizerName,
                  metrics=['accuracy'])

    return model
```

Once we defined our model, we initialed it using deepModel=createDeepModel(), set the desired number of iterations to 50, and the number of clients to four. We used 50 iterations because there were some experiments where 30 iterations were not enough for the model output to stabilize. We tested a handful of clients, but to keep things simple and time efficient, we stuck with four of them.

We then had separate functions to 1) update the server model, 2) update the client model, and 3) train the initial server model. Once these methods were defined, we split the data among the clients using intervals (i.e. client one would contain rows 1 through N out of the training data). Since the data was already shuffled during the train/test split, there was no need to shuffle again. We then loaded the model on each of the clients by training the server model with 20% of the data and distributing the server's deep model amongst the clients. Then, for each iteration, each client model was compiled using binary_crossentropy as the loss function, Adam as the optimizer, and accuracy as the metric. Once the client model was updated, we stored the client model weights in a list, then sent them to the server where they were summed and averaged. Once the server updated its model using the averaged values, the model was saved and sent to the clients, and the process repeated.

5.2 Experimental Design Setup (Poisoning Attack)

The setup for the label-flipping was very similar to that of the previous section. This label-flipping is meant to emulate a poisoning attack. Poisoning attacks are when malicious entities purposely inject bad data during the model's training to produce a less effective model. We tried to be sophisticated with our approach to label-flipping. After splitting the data into train and test sets, we looked up the most popular port for the malware data, and found that Telnet (port 23) was at the top of the list (see Figure 5.2. Telecommunications and Networks (Telnet) is an application protocol that allows users to interact with remote systems. As briefly mentioned in Section 2.1.4, Torii takes advantage of devices exposed to Telnet. Most secure devices should have this port closed because credentials submitted through telnet are not encrypted, leaving clients vulnerable to data leaks.

Table 5.2: Top Five Popular Ports Amongst Malicious Data

Port Number	**MinMax Equivalent**	**Description**	**Number of Occurrences**
23	0.000361	Telnet	1441
443	0.007246	HTTPS	1198
32892	0.539223	Unassigned	607
80	0.001295	HTTP	606
60922	0.998754	Unassigned	342

Out of the 23,793 rows of data we had, 1,441 of them (roughly 6%) had a source port equal to 23. This means that rows with a source port of 23 made up 12% of the malicious data. When the data is split amongst the clients, the first client gets the first 4,282 rows of data. In order to keep the attack consolidated within the first client, we looped through the first 4,282 rows of training data and changed all the labels with a source port equal to 23 from 1 to 0. This was also performed after scaling, so instead of checking to see if the port was equal to 23, we checked to see if it was equal to its scaled equivalent of 0.000361 (see code below):

```
count = 0
# loop through dataframe
for i, row in flip.iterrows():
    # if the src port is 23, change label to 0
    if round(row['tcp_srcport'], 6) == 0.000361:
        flip.at[i, 'is_malware'] = 0
        count += 1
    # stop at the end of the first client
    if i > 4282:
        break
```

Pre-label-flipping, the training data had 10,748 instances of malicious data and 10,665 instances of benign data. The code above resulted in 250 flipped labels from 1 to 0 for the first client. So after label flipping, the first client contained 250 incorrect labels, and the total dataset count of “benign” samples went up to 10,915. Outside of this, everything remained the same as far as the experiment design.

Chapter 6

Validation

In this chapter, we present our results and discuss the impact of these results.

6.1 Results

We ran our model using various hyperparameters and achieved varying results. The hyperparameters adjusted are the following:

- **Number of Clients:** The number of individual clients. For simulation purposes, the number of subsets of the data being trained on our model.
- **Number of Epochs:** The number of times to pass the entire dataset forward and backward through the network.
- **Learning Rate:** A parameter that determines the step size at each iteration.
- **Number of Iterations:** The number of cycles to run the federated learning model.

Using these hyperparameters, we tracked the accuracy, precision, recall, f1 score, and loss of our model. Accuracy measures the number of correctly identified cases and is often used when classes share equal importance or distribution [43]. Precision describes how accurate the model is out of the predicted positives, making it a good measure when the cost of false positives is high [44]. In our case, a false positive would be benign traffic being classified as botnet traffic. Recall describes the number of actual positives captured, making it a good measure when the cost of false negatives is high. In our case, a false negative would result in botnet traffic being classified as benign traffic. For our research, the cost of false negatives is higher than that of false positives, so we pay close attention to recall and precision (more so recall) over accuracy. The f1 score is used when seeking a balance between precision and recall in the presence of an uneven class distribution [44]. It places an emphasis on the importance of false positives and false negatives, whereas accuracy places more importance on true positives and true negatives [43]. As a result, we are also more interested in the f1 scores produced by our model than accuracy. These metrics are further broken down into their mathematical formulas below:

- $Accuracy = \dfrac{TP + TN}{TP + TN + FP + FN}$
- $Precision = \dfrac{TP}{TP + FP}$

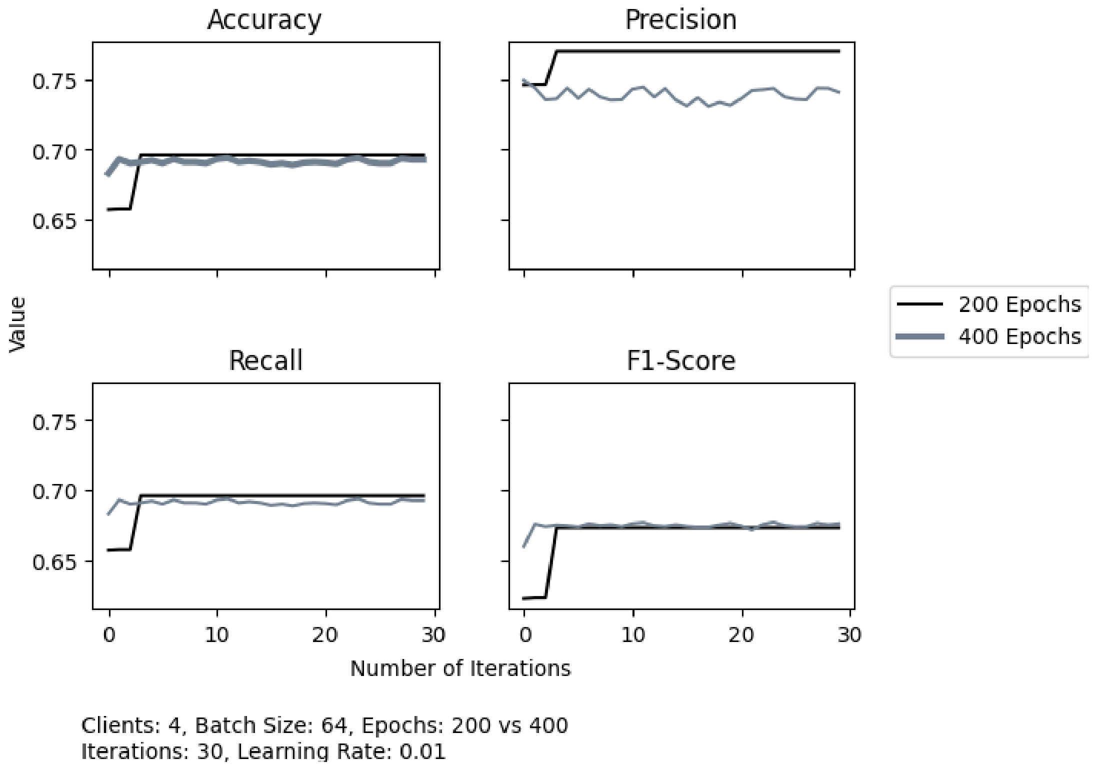


Figure 6.1: Scores Produced While Training 200 vs 400 Epochs (Learning Rate = 0.01)

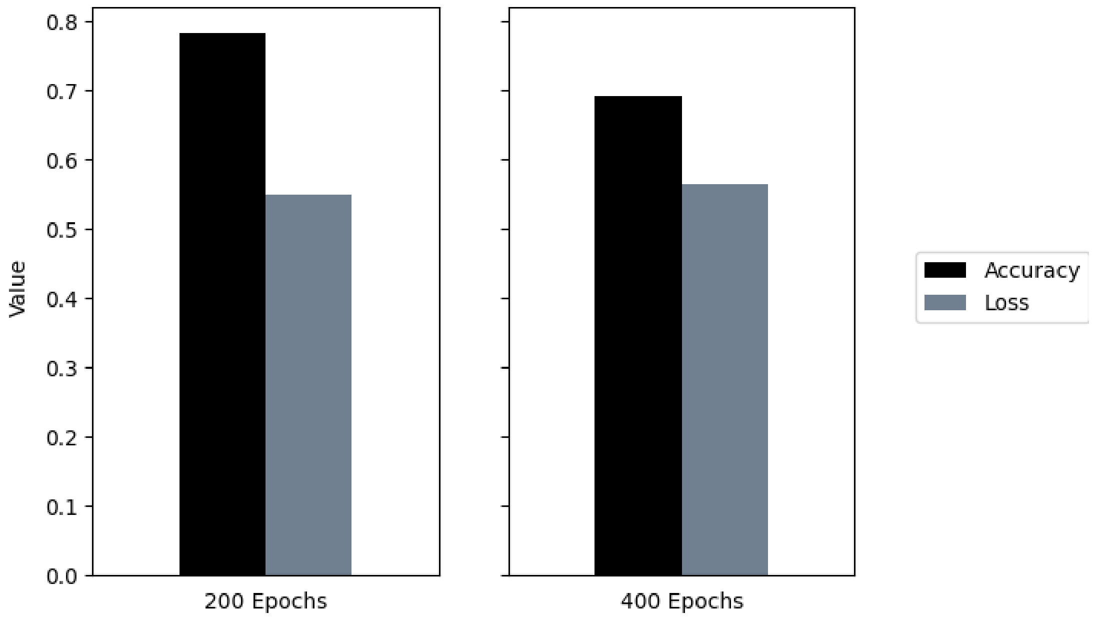
- $Recall = \dfrac{TP + TN}{TP + FN}$
- $F1 = 2 \times \dfrac{Precision \times Recall}{Precision + Recall}$

The following subsections discuss the results obtained using these hyperparameters and metrics.

6.1.1 Hyperparameter: Epochs (200 vs 400)

Using four clients, a batch size of 64, 30 iterations, and a learning rate of 0.01, we found a negligible difference between 200 and 400 epochs. At the end of the last iteration of training, 200 epochs resulted in an accuracy of 70%, precision of 77%, recall of 70%, and f1 score of 67%, while 400 epochs resulted in an accuracy of 69%, precision of 74%, recall of 69%, and an f1 score of 68% (see Figure 6.1). After running our test data through the model, 200 epochs resulted in an accuracy of 78% and a loss of 54%, while 400 epochs resulted in an accuracy of 69% and a loss of 56% (See Figure 6.2).

We then tried comparing 200 vs 400 epochs using a smaller learning rate of 0.001. At the end of the last iteration of training, 200 epochs resulted in an accuracy of 91%, precision of 92%, recall of 91%, and f1 score of 90%, while 400 epochs resulted in an accuracy of 69%,



Clients: 4, Batch Size: 64, Epochs: 200 vs 400
Iterations: 30, Learning Rate: 0.01

Figure 6.2: Accuracy and Loss on Testing 200 vs 400 Epochs (Learning Rate = 0.01)

precision of 77%, recall of 69%, and an f1 score of 67% (see Figure 6.3). After running our test data through the model, 200 epochs resulted in an accuracy of 67% and a loss of 57%, while 400 epochs resulted in an accuracy of 69% and a loss of 62% (see Figure 6.4).

While 400 epochs have a slightly higher accuracy during testing for a learning rate of 0.001, it under-performs in almost all other regards compared to 200 epochs. At a learning rate of 0.001, 200 epochs also seem to fluctuate more for 30 iterations, suggesting we may need to increase this parameter. Since there was no significant improvement in increasing the epoch size past 200 for either learning rate, all further testing was performed using 200 epochs to improve time efficiency.

6.1.2 Model Performance After Ten Runs

Training

Using four clients, a batch size of 64, 50 iterations, and a learning rate of 0.001, we ran our model ten times to account for some randomness and gain an understanding of the model's overall performance. The scores produced by each experiment are shown in Table 6.1.

Figure 6.5 displays the progression of accuracy through 50 iterations for the ten experiments. Based on the graph and Table 6.1, E3 and E10 produce accuracies over 80%, while E7 and E9 produce accuracies below 60%. E1, E2, and E4 produce accuracies between 60%-70%, and E5, E6, and E8 produce accuracies between 70%-80%. When looking at the graph, E4, E7, and E9 all exhibit a sharp decrease in accuracy prior to the 20th iteration and stay

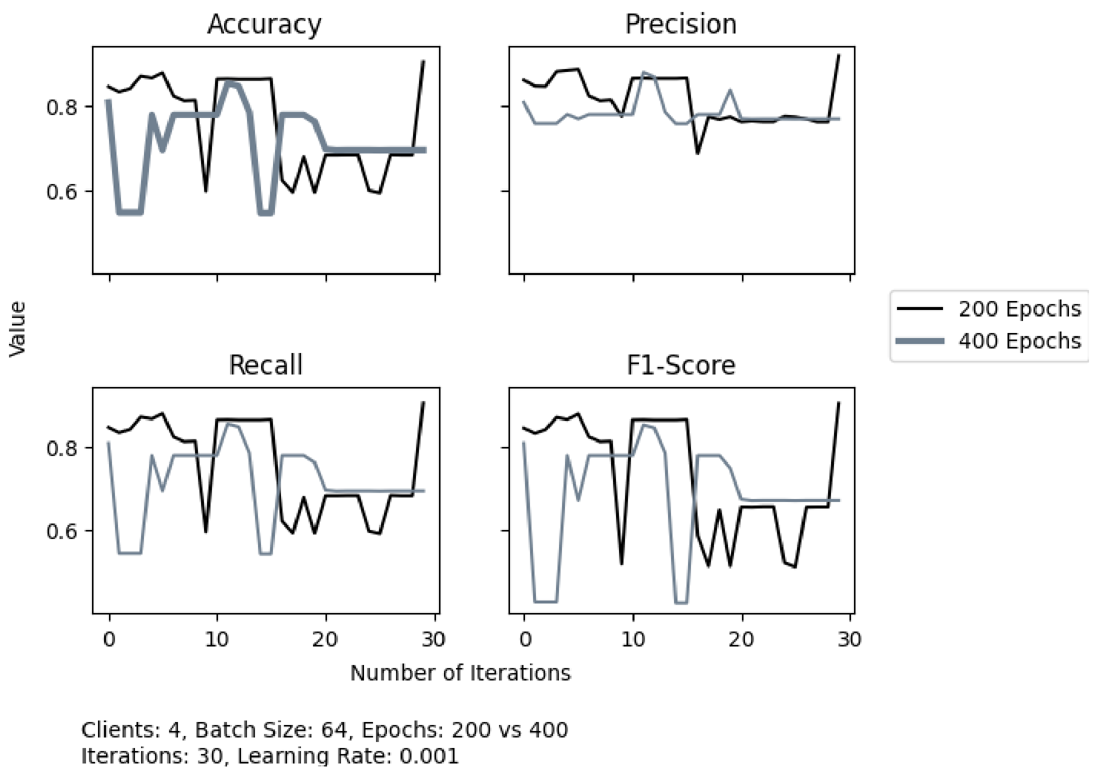


Figure 6.3: Scores Produced While Training 200 vs 400 Epochs (Learning Rate = 0.001)

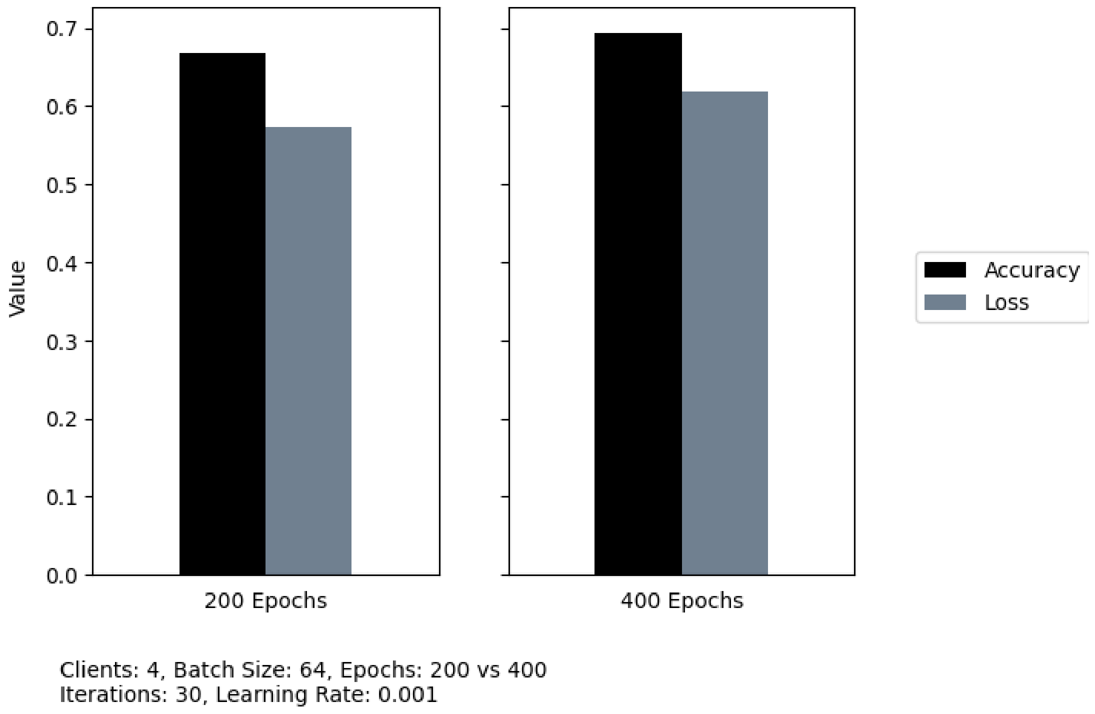


Figure 6.4: Accuracy and Loss on Testing 200 vs 400 Epochs (Learning Rate = 0.001)

Table 6.1: Scores Across 10 Experiments

	Training				Testing	
Experiment	Accuracy	Precision	Recall	F1-Score	Accuracy	Loss
1	67%	79%	67%	63%	67%	58%
2	66%	79%	66%	62%	66%	63%
3	82%	82%	82%	82%	70%	61%
4	65%	74%	65%	62%	64%	60%
5	78%	78%	78%	78%	78%	49%
6	70%	71%	70%	69%	70%	57%
7	54%	76%	54%	43%	64%	63%
8	74%	79%	74%	72%	73%	53%
9	54%	76%	54%	43%	54%	66%
10	81%	81%	81%	81%	81%	51%
Average	**69%**	**78%**	**69%**	**66%**	**69%**	**58%**

stagnant on out. E1, E6, E8, and E10 stay relatively consistent throughout the iterations, and E2, E3, and E5 show an increase by the end of the iterations.

Figure 6.6 displays the progression of precision through 50 iterations for the 10 experiments. Based on the graph and Table 6.1, E3 and E10 produce precision values over 80%, while all other experiments produce precision values between 70%-80%. When looking at the graph, E4, E7, and E9 all exhibit a sharp decrease in precision prior to the 20th iteration and remain stagnant on out. E6, E8, and E10 stay relatively consistent throughout the iterations, and E1, E2, and E5 display a moderate increase in precision. E3 fluctuates, then stops close to where it started during the first iteration.

The graphs for the recall and f1-scores show that these values produce the same shape as the accuracy for each experiment in Figure 6.5, with recall producing the same scores as accuracy by the end of the 50th iteration, and the f1 score being slightly less than accuracy.

Overall, throughout the ten iterations, our model produces an average accuracy of 69%, precision of 78%, recall of 69%, and f1 score of 66%. We see less fluctuation in precision than accuracy, and our precision values are higher than our accuracy values. Our recall is the same as our accuracy, and our f1 score is less than accuracy, precision, and recall. This fluctuation is likely due to randomness. Due to time constraints, we were not able to perform k-fold validation for our model. In the future, however, we believe using k-fold validation may further validate our results and reduce this random fluctuation.

Testing

After running the testing set through our model, Figure 6.7 shows that 60% of the experiments fall between 60%-70% accuracy (E1, E2, E3, E4, E6, and E7). 20% of the experiments fall between 70%-80% accuracy (E5 and E8), and only 10% of the experiments achieved over

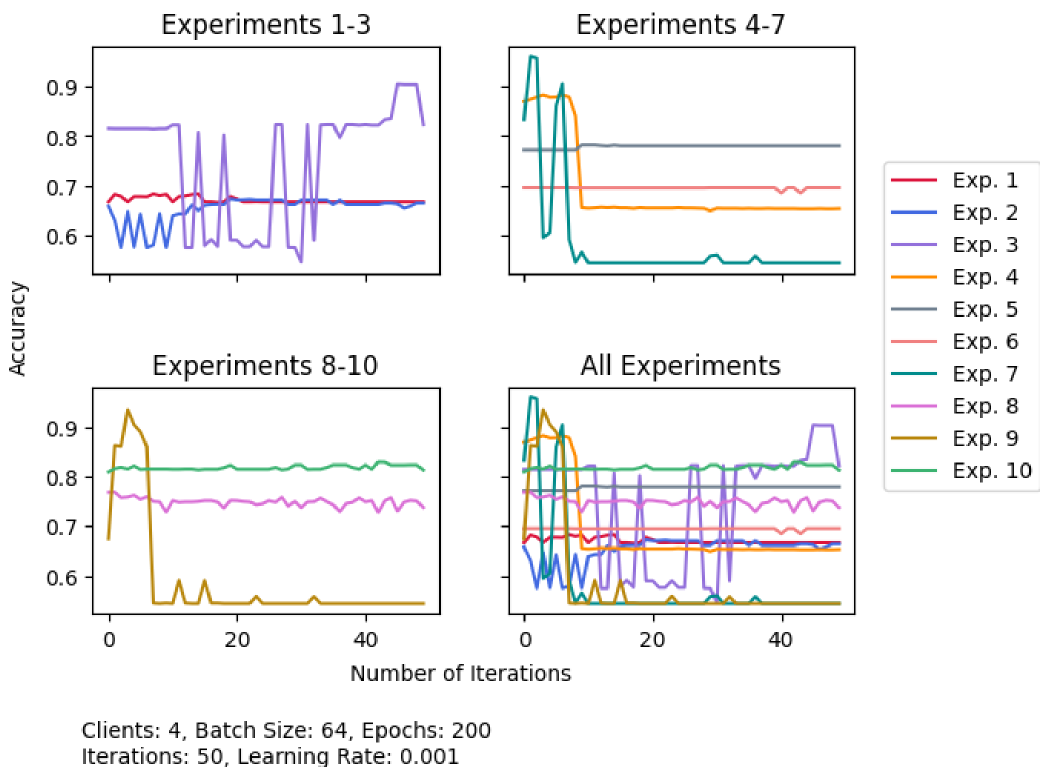


Figure 6.5: Accuracy Produced Through 10 Repetitions of Training

80% accuracy (E10). The remaining 10% (E9) fell between 50%-60%.

Figure 6.7 also shows that while we have relatively high loss values, 90% of the experiments (with the exception of E9) have a loss value less than that of its associated accuracy. Out of the 60% of experiments that fall between 60%-70% accuracy, half of them have a loss value within the same range, and half have a loss value less than 60%. We see the greatest disparity between accuracy and loss E5 and E10 with a difference of 29 and 30 respectively.

On average, we achieve an accuracy of 69% and a loss of 58%. Meaning on average, we achieve the same accuracy during testing as we did during training (see Table 6.1).

6.1.3 Model Performance After Ten Runs (Poisoning Attack)

To label-flip, we took all of the rows in the first client with a source port equal to 23 and changed the label from 1 to 0 (see Section 5.2). Using one corrupted client, three clean clients, a batch size of 64, 50 iterations, and a learning rate of 0.001, we ran our model ten times to account for some randomness and gain an understanding of the model's performance. The scores produced by each experiment are shown in Table 6.2.

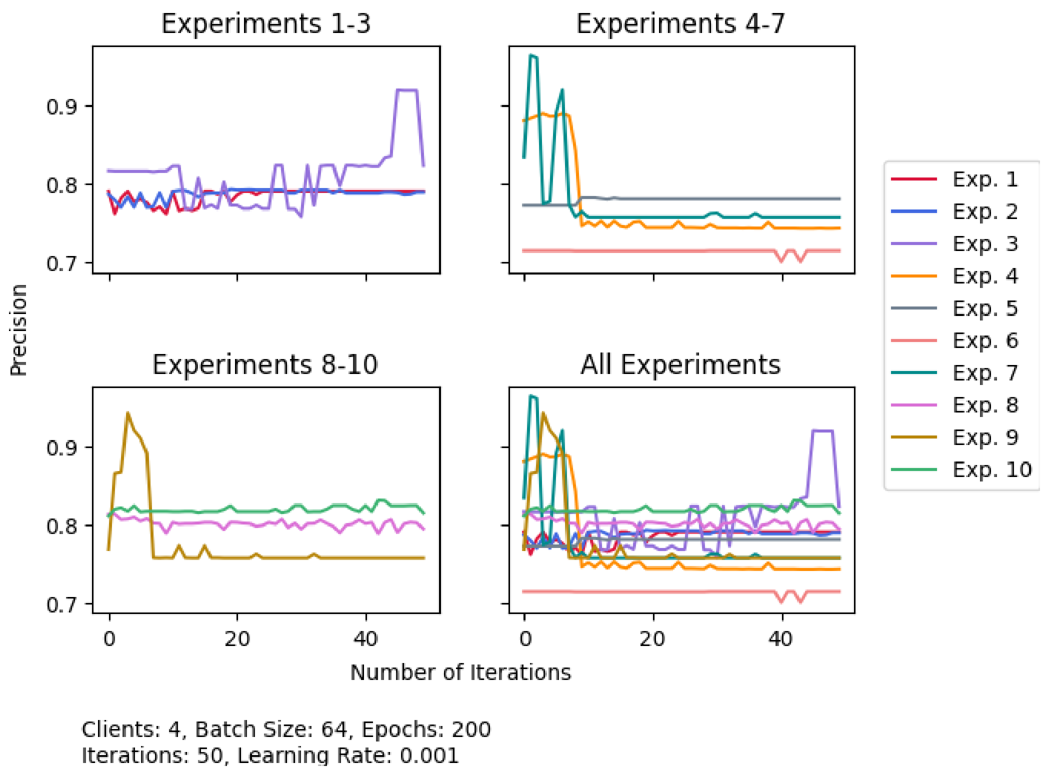


Figure 6.6: Precision Produced Through 10 Repetitions of Training

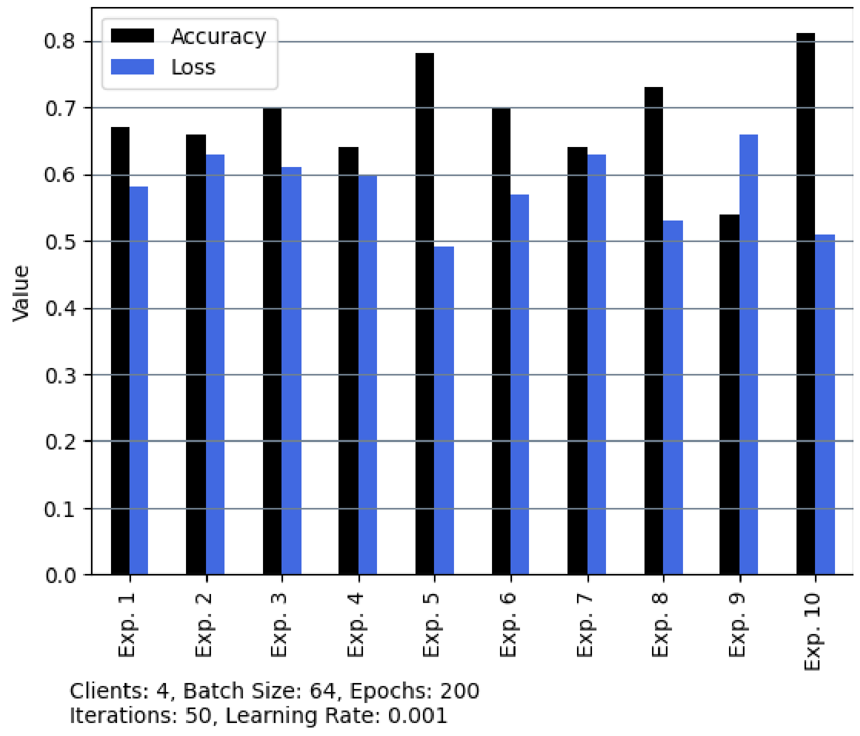


Figure 6.7: Accuracy and Loss Through 10 Repetitions of Testing

Table 6.2: Scores Across 10 Experiments (Poisoning Attack)

	Training				Testing	
Experiment	Accuracy	Precision	Recall	F1-Score	Accuracy	Loss
1	73%	74%	73%	72%	89%	42%
2	69%	77%	69%	67%	69%	62%
3	53%	76%	53%	40%	53%	66%
4	73%	74%	73%	72%	73%	52%
5	69%	72%	69%	68%	69%	64%
6	87%	88%	87%	87%	86%	47%
7	73%	74%	73%	72%	73%	58%
8	92%	93%	92%	92%	65%	60%
9	69%	76%	69%	67%	72%	60%
10	56%	76%	56%	45%	54%	67%
Average	**71%**	**78%**	**71%**	**68%**	**70%**	**58%**

Training

Figure 6.8 displays the progression of accuracy through 50 iterations for the 10 experiments. Based on the graph and Table 6.2, E6 and E8 produce accuracies over 80% and 90% respectively, while E3 and E10 produce accuracies below 60%. E2, E5, and E9 produce accuracies between 60%-70%, and E1, E4, and E7 produce accuracies between 70%-80%. When looking at the graph, E1, E2, E5, and E10 produce a noticeable decrease in accuracy prior to the 20th iteration. E4, and E8 experience rampant fluctuation throughout the iterations, While E3 and E9 increase and then decrease again in a less chaotic manner. E6 and E7 stay relatively consistent throughout the iterations. Unlike the scores produced in Section 6.1.2, while slightly higher accuracies seemed to be achieved, there is noticeably more fluctuation throughout the learning process for the model.

Figure 6.9 displays the progression of precision through the 50 iterations for the 10 experiments. Based on the graph and Table 6.2, E6 and E8 produce precision values over 80% and 90% respectively, while all other experiments produce precision values between 70%-80%. Looking at the graph, E1, E2, E5, E9, and E10 all experience a decrease in precision throughout the iterations. E3 and E8 experience a minor increase in precision while fluctuating more than the other experiments. E4, E6, and E7 experience little to no fluctuation in precision values throughout the iterations.

The recall and f1 scores produce the same shape as the accuracy for each experiment in Figure 6.8, with recall producing the same scores as accuracy by the end of the 50th iteration, and the f1 score being slightly less than accuracy (with the exception of E6 and E8, which produce the same as accuracy and recall).

Overall, throughout the 10 iterations, our model produces an average accuracy of 71%, precision of 78%, recall of 71%, and f1 score of 68%. Compared to our results in Section

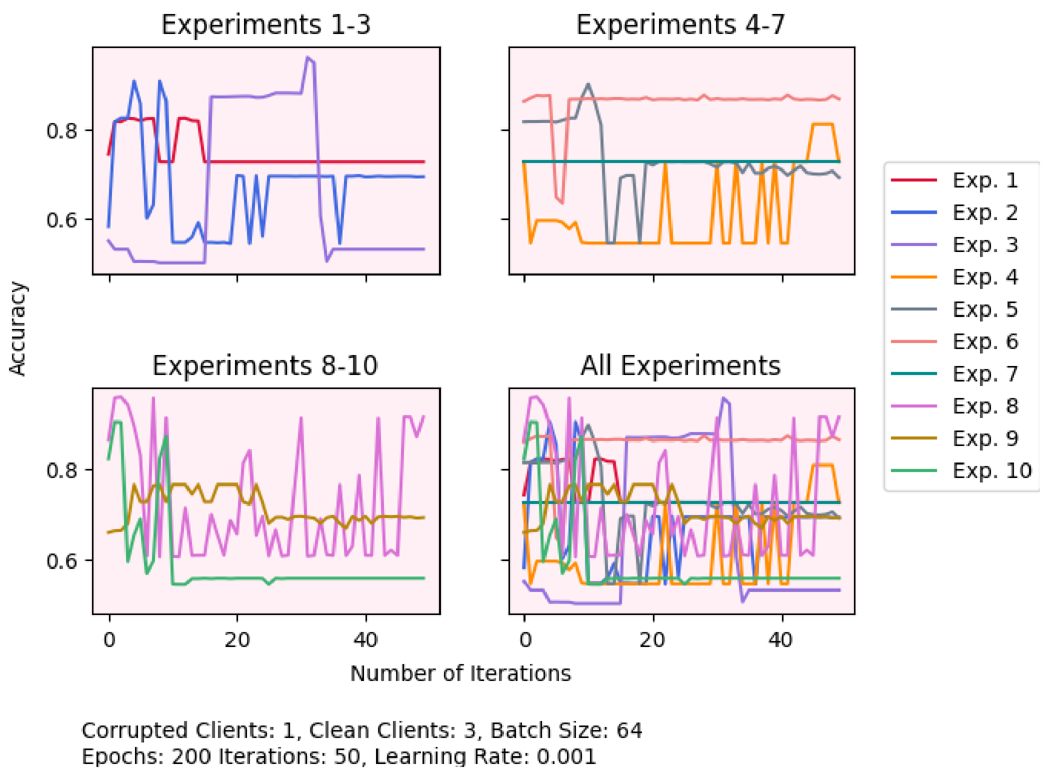


Figure 6.8: Accuracy Produced Through 10 Repetitions of Training (Poisoning Attack)

6.1.2, label-flipping achieves higher accuracy, the same precision, higher recall, and a higher f1 score. We see less fluctuation in precision than accuracy, and our precision values are higher than our accuracy values. Our recall is the same as our accuracy, and our f1 score is mostly less than all accuracy, precision, and recall. While this is strange and unexpected, Section 6.2 goes into detail about why we might be achieving these results.

Testing

After running the testing set through our model, Figure 6.10 shows 30% of the experiments (E2, E5, and E8) produce an accuracy between 60%-70%. 30% of the experiments fall between 70%-80% accuracy (E4, E7, and E9), and 20% of the experiments achieved over 80% accuracy (E1 and E6). The remaining 20% (E3 and E10) fell between 50% and 60%. Compared to testing results in Section 6.1.2, label-flipping seems to produce more sporadic results given 20% as opposed to 10% of the experiments produce accuracies over 80%, while 20%, as opposed to 10%, produce accuracies below 60%.

Figure 6.10 also shows that while we have relatively high loss values, 80% of the experiments (excluding E3 and E10) have a loss value less than that of its associated accuracy. Most of the loss values fall between 60%-70% (E2, E3, E5, E8, E9, and E10). The other 40% of the experiments (E1, E4, E6, and E7) produce loss values below 60%. 20% of the experiments (E3 and E10) produce loss values higher than their associated accuracy, which

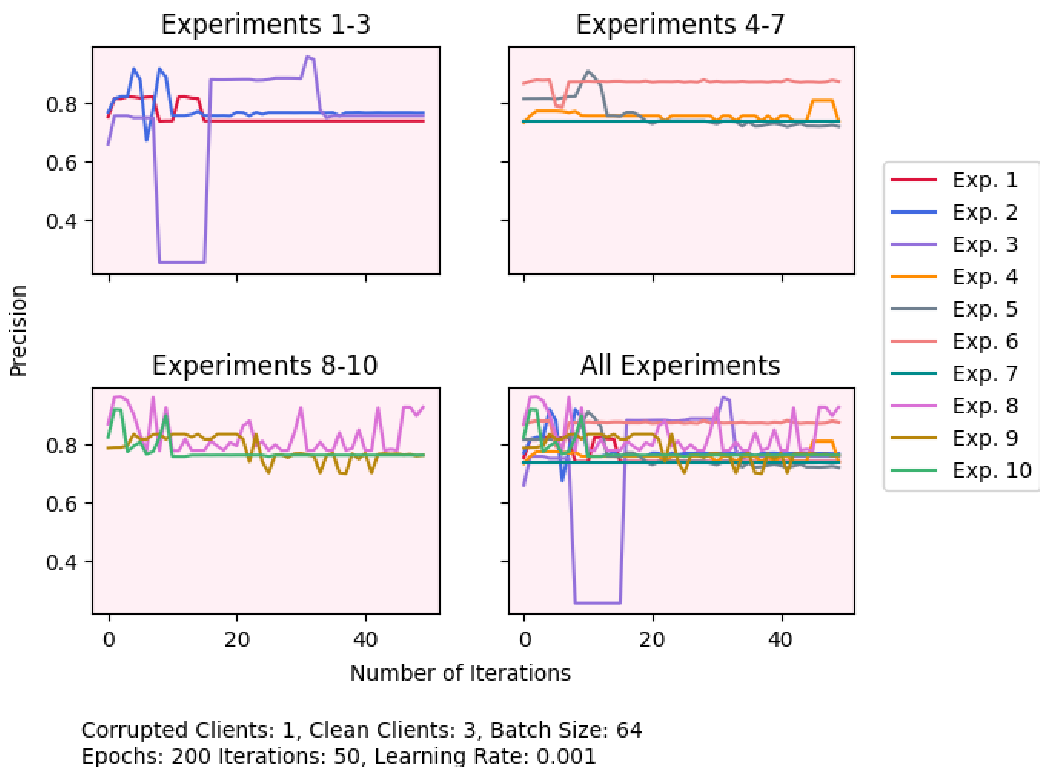


Figure 6.9: Precision Produced Through 10 Repetitions of Training (Poisoning Attack)

is twice as much as the experiments from Section 6.1.2. Last, we see the greatest disparity between accuracy and loss for E1 and E6 with a difference of 47 and 39 respectively.

On average, we achieve an accuracy of 70% and a loss of 58%. Meaning on average, we achieve a slightly lower accuracy during testing than we did during training (see Table 6.2). Compared to Section 6.1.2, we have slightly higher accuracy and the same loss.

6.2 Discussion

Our federated learning model produced on average an accuracy of 69%, precision of 78%, recall of 69%, and f1 score of 66%. After label-flipping for the first client, our model produced on average an accuracy of 71%, precision of 78%, recall of 71%, and f1 score of 68%. Our model post-label-flipping produces a higher accuracy, but as mentioned in Section 6.1, we care more about our recall, precision, and f1 score given the cost of false negatives and false positives is higher than true positives and true negatives. Both models achieve the same precision of 78%, while our model post-label-flipping produces slightly higher recall and f1 scores. This means on average, both models are predicting around the same amount of benign traffic as malicious, but our model post-label-flip predicts less malicious traffic as benign than our model pre-label-flip.

While our precision scores are favorable, our recall and f1 scores are cause for concern

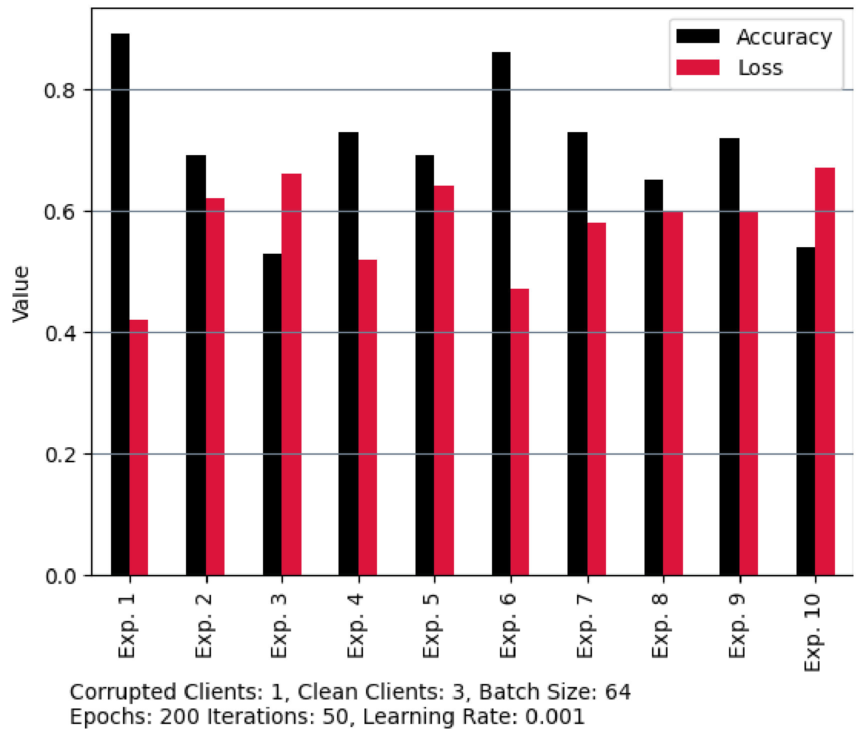
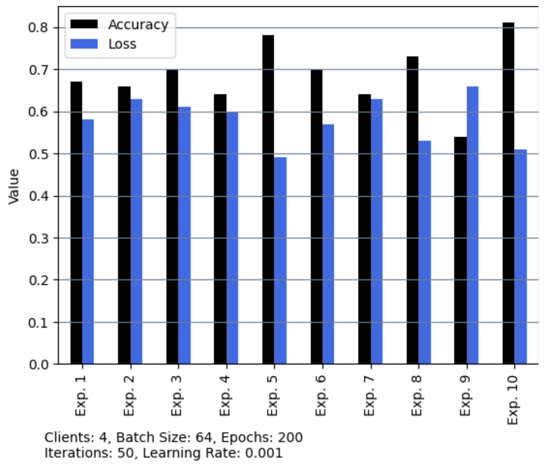


Figure 6.10: Accuracy and Loss Through 10 Repetitions of Testing (Poisoning Attack)

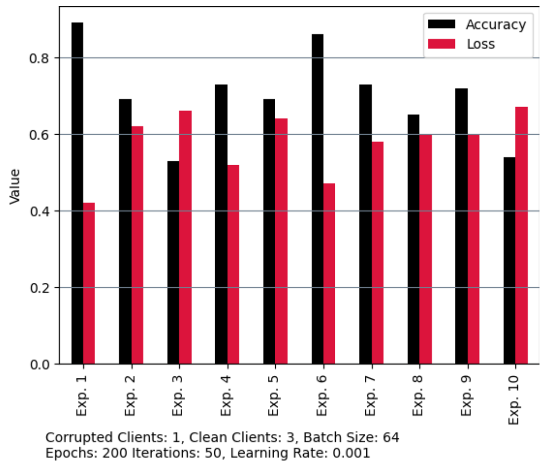
given organizations and individuals will suffer more from malicious traffic being classified as benign than benign traffic being classified as malicious. We also anticipated that our metric results would decrease after label-flipping, but we saw an increase in accuracy, precision, and f1 instead. This could be due to one of three things: 1) Since we only label flipped for the first client and only had four clients in total, a higher accuracy could be produced by the first client due to class imbalance, bringing the average accuracy up in the shared model. This may have less of an impact on a larger number of clients. 2) The model is able to withstand poisoning attacks efficiently. 3) There is something wrong with our model. It is more likely that reasons 1 or 3 are the culprit.

While producing a slightly better metric score, Figure 6.11 shows, however, that the model post-label-flip fluctuates more than the model pre-label-flip. 60% of our model results produce an accuracy between 60%-70%, but only 30% of the results produced after the attack falls within this same range. Post attack, twice the amount of results fall below 60%, and twice the amount fall above 80%. This shows that while the overall results may be higher for label-flipping, a closer look at the model reveals higher fluctuation, making model performance a luck of the draw.

While the results are not as optimal as we would like, [25] mentions how a common challenge with federated learning is often the trade-off between accuracy, privacy, and communication. While it is not too surprising that our model produces a lower accuracy than many published centralized models, the metric results are not optimal enough to consider this federated learning model a sufficient predictor of botnet traffic. Low results could be



(a) Model Performance



(b) Model Performance During Poisoning Attack

Figure 6.11: Difference in Model Performances

caused by a few different factors: the number of neural network layers and their number of inputs and outputs, the features are chosen, the randomness in selected training data, or data imbalance among the clients. In a real-world federated learning scenario though, many of these are things we would not be able to control. The more computationally intensive the neural network, the more likely IoT devices will drop off due to computational power and battery life. To conserve privacy, the model will not be able to see the raw data held on each device, so class balancing is difficult to achieve and excessive data pre-processing may also be difficult or impossible depending on the device. Because we are looking at botnet propagation and C&C communication, the traffic studied is able to blend in easier than if a DDoS attack were taking place. Many of the published federated learning models that achieve high accuracies are fed data consisting of the overall statistics of the network traffic rather than the individual packets themselves. Many of these papers also use attack data, which could be easier to identify than propagation data (i.e. time delta and time relative since first or previous may be shorter during an attack than C&C communication). If we wish to use federated learning as a privacy-protecting, more intelligent form of intrusion detection and prevention pre-attack, further research on the subject is required.

Chapter 7

Conclusion

We proposed an intrusion detection system that utilizes federated learning to preserve data privacy amongst devices. We analyze raw packet data from legitimate botnet traffic. Sampling packets from the propagation and C&C communication phase, we propose an online model to differentiate between malicious and benign traffic on a per-packet basis without allowing the clients to share raw network data. We also examined whether poisoning attacks have an impact on model performance. Using MedBIoT, we converted the fine-grained pcap files to csv format using Wireshark and tshark. We then dropped nulls, scaled, and split our data into training and testing sets. The training data was distributed amongst four clients using a batch size of 64, 200 epochs, and 50 iterations. After running our model 10 times before and after label-flipping, we found that the performance is relatively similar with more fluctuation among the poisoned model throughout the trials. While we believe further improvement could be made to our model, we believe an intrusion detection system using federated learning is a positive step forward to protecting our devices and data privacy.

7.1 Limitations and Drawbacks

To our knowledge, federated learning models only exist in simulated environments and are not utilized publicly in the real world. There are a few reasons for this. One is that these models have to be tailored to fit the data, rather than tailoring the data to fit the model like most centralized approaches. In regards to network data, the reality is many packets could have missing values, outliers, and an unbalanced distribution of malware and normal traffic depending on the security of the system and network. Devices, especially IoT devices, also have different levels of computational power and battery life. This means that when randomly selecting devices, the central server will have to be able to withstand devices dropping randomly due to network bandwidth, battery power, going on or offline, and so on. There is also the issue of how many devices to randomly select for training, and making sure that there isn't a bias as to which devices to select. This method of intrusion detection does require more research regarding the kinds of models devices are able to process and the type of data to analyze from the packets. Then as zero-day malware (malware that hasn't been seen before) is researched and discovered, the model would have to be trained or altered to process new parameters and information much like how intrusion detection systems must be maintained to keep up with new malware. We initiated the process of trying to detect discrepancies on networks using a federated learning-based intrusion detection system, but more has to be done for this idea to be used in real-world applications.

7.2 Future Work

We tried to track botnet propagation on a packet-by-packet basis. This is a difficult task given botnet traffic can easily blend in with regular traffic during propagation and C&C communication pre-attack. This will only become more difficult as newer variants continue to incorporate encryption to evade attacks. This is a research area that requires more attention as we try to prevent attacks and conserve data privacy. There are three things that should be done in the future to potentially produce a better federated learning model. First, more time should be spent on an in-depth analysis of feature selection. Packets contain a lot of data. For ease of training, we used numerical, non-continuous data. It would be interesting to see more features included in pre-processing, such as categorical data and continuous data. Feature selection should also be used to determine feature correlation and impact, and features should be properly encoded based on their type. It is important to find a balance between feature selection and data pre-processing and what devices such as smartphones and IoT can handle computationally though. For federated learning, the model needs to be tailored to the data and clients it processes, rather than tailoring the data to the model. Second, k-fold cross-validation could be tested to see how it impacts model fluctuation. Third, different poising attacks should be used until model performance is visibly decreased. During this research, we saw that some examples of label-flipping decreased accuracy, while other examples increased accuracy. This may be because the label-flipping attack model we have chosen is not perfect. In the future, more sophisticated attacks should be tested to decrease model performance for every test run. This will help researchers understand what types of tailored attacks hackers might use and help them build their model to appropriately defend against such attacks.

References

[1] Rob Sobers. 134 cybersecurity statistics and trends for 2021: Varonis, Mar 2021.

[2] Sectigo. Evolution of iot attacks: An interactive infographicd. `https://sectigo.com/uploads/resources/Evolution-of-IoT-Attacks-Interactive-IG_May2020.pdf`.

[3] Alejandro Guerra-Manzanares, Jorge Medina-Galindo, Hayretdin Bahsi, and Sven Nõmm. Medbiot: Generation of an iot botnet dataset in a medium-sized iot network. 02 2020.

[4] Thomas S. Hyslip and Jason M. Pittman. A survey of botnet detection techniques by command and control infrastructure. *Journal of Digital Forensics, Security and Law*, 10, 2015.

[5] radware. IRC (Internet Relay Chat). `https://www.radware.com/security/ddos-knowledge-center/ddospedia/irc-internet-relay-chat`.

[6] J. Oikarinen and D. Reed. Rfc1459: Internet relay chat protocol, 1993.

[7] Wentao Chang, Aziz Mohaisen, An Wang, and Songqing Chen. Measuring Botnets in the Wild: Some New Trends. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 645–650, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Sérgio S.C. Silva, Rodrigo M.P. Silva, Raquel C.G. Pinto, and Ronaldo M. Salles. Botnets: A survey. *Computer Networks*, 57(2):378–403, 2013. Botnet Activity: Analysis, Detection and Shutdown.

[9] Shivani Gaonkar, Nandini Fal Dessai, Jenny Costa, Ashlesha Borkar, Shailendra Aswale, and Pratiksha Shetgaonkar. A survey on botnet detection techniques. In *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, pages 1–6, 2020.

[10] Syeda Farjana Shetu, Mohd. Saifuzzaman, Nazmun Nessa Moon, and Fernaz Narin Nur. A survey of botnet in cyber security. In *2019 2nd International Conference on Intelligent Communication and Computational Techniques (ICCT)*, pages 174–177, 2019.

[11] The Malware Wiki. Bashlite. `https://malwiki.org/index.php?title=BASHLITE`, May 2021.

[12] Black Hat Ethical Hacking. Lizard squad – the infamous hacking group that brought xbox and playstation networks to their knees. https://www.blackhatethicalhacking.com/articles/hacking-stories/lizard-squad-the-infamous-hacking-group-that-brought-xbox-and-playstation-networks-to-their-knees/, March 2022.

[13] Warwick Ashford. Lizardstresser iot botnet launches 400gbps ddos attack. https://www.computerweekly.com/news/450299445/LizardStresser-IoT-botnet-launches-400Gbps-DDoS-attack, June 2016.

[14] Waqas. Bashlite malware turning millions of linux based iot devices into ddos botnet. https://www.hackread.com/bashlite-malware-linux-iot-ddos-botnet/, September 2016.

[15] CloudFlare. What is the mirai botnet? https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/.

[16] Garrett M. Graff. How a dorm room minecraft scam brought down the internet. https://www.wired.com/story/mirai-botnet-minecraft-scam-brought-down-the-internet/, 2017.

[17] Josh Fruhlinger. The mirai botnet explained: How teen scammers and cctv cameras almost brought down the internet. https://www.csoonline.com/article/3258748/the-mirai-botnet-explained-how-teen-scammers-and-cctv-cameras-almost-brought-down-the-internet.html, 2018.

[18] Brian Krebs. Mirai Botnet Authors Avoid Jail Time. https://krebsonsecurity.com/2020/03/zxyel-flaw-powers-new-mirai-iot-botnet-strain/, 2020.

[19] Jakub Kroustek, Vladislav Iliushi, Anna Shirokova, Jan Neduchal, and Martin Hron. Torii botnet - not another mirai variant. https://www.bleepingcomputer.com/news/security/new-iot-botnet-torii-uses-six-methods-for-persistence-has-no-clear-purpose/, September 2018.

[20] Ionut Ilascus. New iot botnet torii uses six methods for persistence, has no clear purpose. https://www.bleepingcomputer.com/news/security/new-iot-botnet-torii-uses-six-methods-for-persistence-has-no-clear-purpose/, September 2018.

[21] IBM Cloud Education. Machine learning. https://www.ibm.com/cloud/learn/machine-learning, July 2020.

[22] M. I. Jordan and T. M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[23] Julianna Delua. Supervised vs. unsupervised learning: What's the difference?, March 2021.

[24] Mehryar Mohr, Afshin Rostamizadeh, and Ameet Talwalker. *Foundations of Machine Learning*. Adaptive computation and machine learning series. Massachusetts Institute of Technology, 2 edition, 2018.

[25] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista A. Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaïd Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Mariana Raykova, Hang Qi, Daniel Ramage, Ramesh Raskar, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning. *CoRR*, abs/1912.04977, 2019.

[26] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.

[27] Li Li, Yuxi Fan, Mike Tse, and Kuo-Yi Lin. A review of applications in federated learning. *Computers Industrial Engineering*, 149:106854, 2020.

[28] Bimal Ghimire and Danda B. Rawat. Recent advances on federated learning for cybersecurity and cybersecurity for federated learning for internet of things. *IEEE Internet of Things Journal*, 9(11):8229–8249, 2022.

[29] Craig Stedman. data silo. `https://www.techtarget.com/searchdatamanagement/definition/data-silo`.

[30] Valerian Rey, Pedro Miguel Sánchez Sánchez, Alberto Huertas Celdrán, and Gérôme Bovet. Federated learning for malware detection in iot devices. *Computer Networks*, 204:108693, 2022.

[31] Rafa Galvez, Veelasha Moonsamy, and Claudia Díaz. Less is more: A privacy-respecting android malware classifier using federated learning. *CoRR*, abs/2007.08319, 2020.

[32] Ruei-Hau Hsu, Yi-Cheng Wang, Chun-I Fan, Bo Sun, Tao Ban, Takeshi Takahashi, Ting-Wei Wu, and Shang-Wei Kao. A privacy-preserving federated learning system for android malware detection based on edge computing. In *2020 15th Asia Joint Conference on Information Security (AsiaJCIS)*, pages 128–136, 2020.

[33] Ying Zhao, Junjun Chen, Di Wu, Jian Teng, and Shui Yu. Multi-task network anomaly detection using federated learning. In *Proceedings of the Tenth International Symposium on Information and Communication Technology*, SoICT 2019, page 273–279, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Barracuda. Intrusion detection system. `https://www.barracuda.com/support/glossary/intrusion-detection-system`.

[35] Wahaya IT. Protect your organization against cyber poisoning attacks. `https://www.wahaya.com/cyber-poisoning-attack/#:~:text=Types%20of%20Poison%20Attacks&text=Data%20manipulation%20%E2%80%93%20The%20attacker%20manipulates,and%20ultimately%20weaken%20the%20outcome`.

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

[37] Yair Meidan, Michael Bohadana, Yael Mathov, Yisroel Mirsky, Asaf Shabtai, Dominik Breitenbacher, and Yuval Elovici. N-baiot—network-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, 17(3):12–22, 2018.

[38] Sebastian Garcia, Agustin Parmisano, and Maria Jose Erquiaga. Iot-23: A labeled dataset with malicious and benign iot network traffic (version 1.0.0) [data set]. 2020.

[39] Muna Al-Hawawreh, Elena Sitnikova, and Neda Aboutorab. X-iiotid: A connectivity- and device-agnostic intrusion dataset for industrial internet of things. 2021.

[40] Sebastian Garcia, Martin Grill, Jan Stiborek, and Alejandro Zunino. An empirical comparison of botnet detection methods. *Computers and Security Journal*, 45:100–123, 2014.

[41] Sadman Sakib. Federated learning keras. `https://github.com/SadmanSakib93/Federated-Learning-Keras`, 2021.

[42] Sadman Sakib, Mostafa M. Fouda, Zubair Md Fadlullah, and Nidal Nasser. On covid-19 prediction using asynchronous federated learning-based agile radiograph screening booths. In *ICC 2021 - IEEE International Conference on Communications*, pages 1–6, 2021.

[43] Purva Huilgol. Accuracy vs. f1-score. `https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2`, Aug 2019.

[44] Koo Ping Shung. Accuracy, precision, recall or f1? `https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9`, Mar 2018.