University of Zurich[UZH]

MASTER PROJECT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Creation of New Datasets for Decentralized Federated Learning

Jing Han
Zurich, Switzerland
Student ID: 22-738-686
Xi Cheng
Zurich, Switzerland
Student ID: 21-742-945
Zien Zeng
Zurich, Switzerland
Student ID: 21-741-558
Heqing Ren
Zurich, Switzerland
Student ID: 22-736-128

Supervisor: Chao Feng, Dr. Alberto Huertas Celdrán

Date of Submission: January 12, 2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Project
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

We hereby declare that we have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). We are aware that we take full responsibility for the scientific character of the submitted text ourselves, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich,

Signature of students

Zusammenfassung

Das Internet der Dinge (IoT) erlebt einen rasanten Anstieg der Anzahl von verbundenen Geräten, die darauf ausgelegt sind, eine enorme Menge an Daten zu verarbeiten und zu kommunizieren. Diese umfangreiche Datenmenge macht IoT-Geräte jedoch auch anfällig für Cyberangriffe, was die Sicherheit und Integrität der Daten gefährdet. Daher ist es entscheidend, die Sicherheit von IoT-Geräten zu priorisieren, um Datenverletzungen zu verhindern und den Schutz sensibler Informationen zu gewährleisten. Die Verfügbarkeit großer und öffentlich zugänglicher Datensätze für die Erkennung von IoT-Malware ist jedoch begrenzt, was eine Herausforderung für Forscher in diesem Bereich darstellt. Darüber hinaus hat sich die meiste Forschung in diesem Bereich auf traditionelles maschinelles Lernen (ML) und zentrales föderiertes Lernen (CFL) verlassen, was Limitationen in der Anpassung an die dynamischen IoT-Daten und Datenschutzbedenken mit sich bringt. Im Gegensatz dazu ermöglicht dezentrales föderiertes Lernen (DFL) verteiltes Lernen über IoT-Geräte, schützt die Datensicherheit und verbessert die Anpassungsfähigkeit des Modells an die sich entwickelnde Bedrohungslandschaft im IoT-Ökosystem.

Diese Arbeit beinhaltete die Überwachung verschiedener Verhaltensquellen von realen IoT-Geräten. Ziel war es, einen umfassenden Datensatz zu erstellen, der sowohl normales Geräteverhalten als auch anomales Verhalten bei Angriffen durch Malware erfasst. Zusätzlich präsentierte diese Arbeit einen ML-basierten Ansatz, der das DFL-Modell nutzt, um Malware zu erkennen und zu klassifizieren, die auf IoT-Geräte abzielt. Dieses Modell verspricht, Robustheit zu erhöhen und die Datensicherheit zu wahren. Ferner führte diese Arbeit eine gründliche Analyse und Vergleich verschiedener Szenarien durch, um die Wirksamkeit und Zuverlässigkeit des vorgeschlagenen Modells zu demonstrieren.

Abstract

The Internet of Things (IoT) is witnessing a rapid increase in the number of connected devices, which are designed to process and communicate an enormous amount of data. However, this extensive volume of data also makes IoT devices susceptible to cyberattacks, compromising the security and integrity of data. As a result, it is essential to prioritize safeguarding the security of IoT devices to prevent data breaches and ensure the protection of sensitive information. However, the availability of large and publicly accessible datasets for IoT malware detection is limited, posing a challenge for researchers in this field. In addition, most of the research in this area has relied on Traditional Machine Learning (ML) and Centralized Federated Learning (CFL) approaches, which face limitations in adapting to the dynamic IoT data and privacy concerns. In contrast, Decentralized Federated Learning (DFL) enables distributed learning across IoT devices, preserves data privacy, and enhances the model's adaptability to the evolving threat landscape in the IoT ecosystem.

This work involved monitoring various sources of behavior from real IoT devices. The aim was to create a comprehensive dataset capturing both normal device behavior and anomalous device behavior when under attack by malware. Additionally, this work put forward a ML-based approach that utilizes the DFL model to detect and classify malware targeting IoT devices. This model promises to enhance robustness and maintain data privacy. Furthermore, this work conducted a thorough analysis and comparison of various scenarios to demonstrate the effectiveness and reliability of the proposed model.

Acknowledgments

We would like to express our sincere gratitude to our supervisors, Chao Feng and Dr. Alberto Huertas Celdrán for their continuous support and guidance throughout our master project. Their expertise in cybersecurity and data science, coupled with the invaluable suggestions they provided, have been a great help for this project. We are thankful for the knowledge and experience gained under their mentorship.

We would like to thank Prof. Dr. Burkhard Stiller for giving us the opportunity to undertake our master project in an area of our interest. It is a pleasant and rewarding experience to conduct research at the Communication Systems Group.

Contents

Chapter 1

Introduction

In the era of Industry 4.0, the Internet of Things (IoT) is a rapidly evolving technological field. IoT devices, equipped with sensors, microcontrollers, and other digital components, are increasingly interconnected via the Internet. While these devices facilitate human lives, they also increase the likelihood of Radio Frequency (RF) collision. To manage the RF spectrum effectively, Electorsense, an innovative crowdsensing spectrum monitoring platform, has been introduced [1]. This platform utilizes devices with limited resources, like Raspberry Pis, to gather, transmit, and process RF spectrum data. However, the expansion of these spectrum sensors also introduces new security vulnerabilities, particularly susceptibility to malware attacks. These concerns are further compounded by the inherent complexity and diverse nature of IoT devices, making them challenging to secure effectively.

To address these challenges, researchers have been focusing on developing robust security solutions. Notably, some studies have made significant strides in creating effective anomaly detection and malware classification models utilizing Machine Learning (ML) and Deep Learning (DL). However, these models encounter limitations, especially in scenarios where sharing IoT device data is restricted due to privacy concerns. Federated Learning (FL) has emerged as a promising solution to this problem. It offers a way to perform distributed ML in isolated data environments while maintaining data confidentiality [2]. In the FL framework, different clients independently train models on their local data. These local models are then aggregated to form a global model. This approach not only ensures the privacy of each client's data but also contributes to improving the generalization capabilities of the overall model. Through FL, it becomes possible to enhance IoT security without compromising data privacy.

While FL represents a considerable step forward in ensuring data privacy, it does face certain challenges, especially in its Centralized FL (CFL) configuration. The CFL model relies on a central server for the aggregation of the global model, which introduces the risk of a single point of failure. To mitigate this risk, Decentralized FL (DFL) has emerged as a promising area of research. DFL eliminates the need for a central server, adopting a distributed approach for training ML models. This shift from a centralized to a decentralized framework enhances the robustness of the system and further strengthens data privacy safeguards.

In light of these considerations, this project aims to develop a DFL-based method for anomaly detection and malware classification, with the goal of protecting the integrity of IoT spectrum data while preserving privacy. This chapter outlines the motivation for this approach, along with a description of the project's scope and its organizational structure.

1.1 Motivation

Due to the absence of genuine platform data related to device fingerprints for resource-constrained spectrum sensors, there is a need to establish an extensive and comprehensive dataset that encompasses various types of device information.

Traditional ML and DL methods typically require centralized collection and storage of data from all participants, followed by model training on a central server. This can lead to risks of privacy leakage.

To address these issues, this work has collected a large dataset and integrated device fingerprinting with FL. FL employs a decentralized approach, which can mitigate the risk of privacy leakage.

1.2 Description of Work

The main goal of this thesis is to design and implement a framework based on DFL for the detection and classification of eight types of malware.

To achieve this goal, the following tasks were completed:

First, research on IoT security, anomaly detection and malware classification, as well as DFL was conducted. Through these works, necessary background knowledge was obtained, including the existing anomaly detection and malware classification methods, and whether malware attacks would affect the Network, Input/Output (I/O), File System, Resource Usage, System Call, and Kernel Events of a Raspberry Pi. This work helped in choosing the anomaly detection and malware classification methods and the behavioral data to be collected next.

Next, the experimental work of the project began. A Control Module was created to control the operation of six monitoring modules, collecting behavioral data from six Raspberry Pi 3 devices and two Raspberry Pi 4 devices connected to the ElectroSense platform. Each device collected data under eight malware attacks and in normal state for four hours, totaling 288 hours of data and resulting in a dataset of over 50,000 rows. Data cleaning, feature extraction, and feature normalization were performed to obtain a dataset suitable for ML, which then underwent further feature selection to avoid model overfitting.

In the third step, this work designed, developed, trained, and evaluated two FL pipelines, one of which was an Autoencoder, and the other was a Multilayer Perceptron (MLP).

Through the integration of the dataset and models to the Fedstellar framework, decentralized training and testing of the distributed learning model were achieved. The Autoencoder was used to identify the presence of malware attacks, utilizing 22 features selected from the feature dataset for training and testing. The MLP was used to classify different types of malware, utilizing 30 features selected from the feature dataset for training and testing.

Finally, the results of the Autoencoder and MLP were evaluated. Multiple related experiments were conducted, including those with different topologies of DFL, to assess the robustness of the experimental results. The performance of the experiment was evaluated and compared using metrics such as precision, recall, and f1-score, confirming the effectiveness of the ML pipeline.

1.3 Thesis Outline

The remainder of this thesis encompasses the following content: Chapter 2 introduces background knowledge on spectrum sensing data and ElectroSense, malware affecting IoT devices, malware analysis methods, ML and DL techniques, FL, Fedstellar, and feature extraction techniques. This knowledge is instrumental in understanding the design philosophy of the implementation. Chapter 3 conducts a comparative analysis of research work related to anomaly detection and malware classification, contrasts traditional DL models with FL models, and collects and organizes the effects of different types of malware on behavioral sources. In Chapter 4, the system architecture and its six modules are elaborated in detail: the control module, monitoring module, transmission module, data processing module, FL module, and evaluation module. Chapter 5 delineates the implementation of the experiments, covering steps including setup, feature extraction, integration, data preprocessing, feature selection, and model training. Chapter 6 designs related experiments to assess the robustness of the implementation, including a comparison between ML, CFL, and DFL approaches, as well as a comparison between different DFL topologies, and so on. The final chapter, Chapter 7, concludes the thesis and provides a prospectus for future work.

Chapter 2

Background

In recent years, the development of IoT technology has led to the integration of IoT devices into modern society, providing unprecedented convenience, automation, and connectivity to every aspect of people's daily lives. This has resulted in an increased use of RF bands, necessitating solutions for the challenges posed by large-scale spectrum monitoring, leading to the creation of ElectroSense. Moreover, the significant growth in the number of IoT devices has also made them increasingly targeted by malicious actors seeking to exploit vulnerabilities for personal gain.

This chapter provides a brief introduction to spectrum sensing data and ElectroSense and describes the types of malware that affect IoT devices. It then explains the two phases of malware analysis: detection and classification. Additionally, it introduces techniques for anomaly detection and malware classification, including ML-based and DL-based methods. Furthermore, the Fedstellar framework built for FL is introduced. Lastly, it provides an introduction to feature extraction methods.

2.1 Spectrum Sensing Data and ElectroSense

The rapid development of IoT, which connects physical devices, vehicles, household items, and other everyday objects to the internet, has led to improvements in production efficiency and life quality. However, it also presents a challenge due to the increasing use of RF bands, making spectrum management more important than ever [1].

Spectrum sensing data is crucial for optimizing spectrum resources, as it involves monitoring the use of the radio spectrum to identify which frequencies are being used and which are not. The extensive use of RF bands has made spectrum monitoring increasingly difficult, because it requires intensive sampling in time, frequency, and space.

To address the challenges brought by large-scale spectrum monitoring, ElectroSense was created. It is a crowdsourced network (see Figure 2.1), using low-cost sensors as the primary devices for sensing the spectrum to collect and analyze spectrum data [3]. ElectroSense aims to sense the entire spectrum in different regions of the world and provide processed spectrum data to any user interested in making research on spectrum. The spectrum sensing data from different sensors can be accessed through an open API [4] from the ElectroSense backend, significantly increasing the accessibility of spectrum sensing data for the general public.

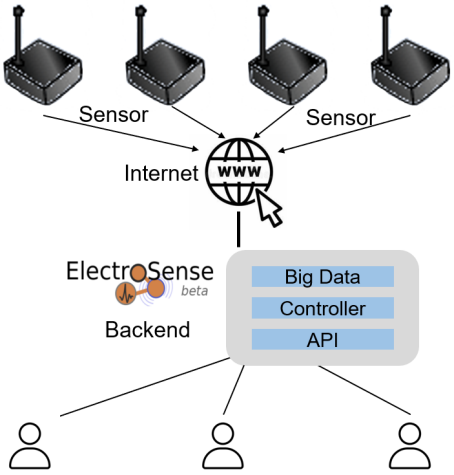


Figure 2.1: ElectroSense Network Overview

The sensors used in the ElectroSense project are built on software-defined radio (SDR) technology and cheap embedded platforms, with one example of an embedded platform being the Raspberry Pi. This reduces the cost for contributors of spectrum data and aids in the global deployment of ElectroSense. Raspberry Pies are the infrastructure that this project utilizes.

2.2 Malware Affecting IoT Devices

Malware targeting IoT devices poses a significant threat to user privacy and system security. Attackers exploit vulnerabilities to gain unauthorized access, steal sensitive data, or use device resources for malicious activities. There are five main families of malware affecting IoT devices, including Botnets, Backdoors, Rootkits, Coinminer, and Ransomware [5]. Each malware family contains multiple types of malware. Here is a brief overview of malware families and specific types within them.

2.2.1 Botnets

Botnets are networks of computers infected with malware, which are controlled by attackers. These infected computers can perform various tasks without user authorization,

such as distributing malware, launching Distributed Denial of Service (DDoS) attacks, sending spam, etc. Botnets usually infect computers through backdoors, worms, or malicious downloaders. Attackers can remotely manage Botnets through command-and-control servers and use them to carry out large-scale cyberattacks and illegal activities.

- Bashlite [6]. In 2014, the original version of this program was developed and it gained immense popularity over the next two years. This botnet comprises of two files: ”client.c” and ”server.c”. The former file infects a device, while the latter acts as the command and control system that manages every bot in the network. The bots can spread the payload to other devices and can also use brute-force attacks to take advantage of vulnerable telnet credentials. Such actions eventually lead to the botnet's growth until it becomes powerful enough to launch a DDoS attack [5].

2.2.2 Backdoors

Backdoors is a security vulnerability that enables unauthorized access to a system by bypassing normal security mechanisms. Attackers can use the backdoor to gain direct access to a compromised system, undetected, and perform malicious actions such as stealing sensitive data or planting other malware. Backdoors can be created directly by attackers or planted by other malware.

- HttpBackdoor [7] is a type of backdoor that is known to be the simplest and most basic among other backdoors. It operates by setting up a Web server on an IoT spectrum sensor, Raspberry Pi, and then waits for HTTP requests from the attacker, also known as the client. This particular backdoor applies three behaviors, namely execution, download, and removal, while also showcasing two fundamental functionalities. The first functionality enables the attacker to retrieve fundamental system information. This includes the name and version of the operating system, or saved SSH keys, by simply sending a GET request. The second functionality allows the attacker to execute command line commands on the Raspberry Pi by sending a POST request [5].
- Backdoor [8] is made up of two parts - the client and server sides. The server side sends commands to a specified IP address and port, which the client side (running on the Raspberry Pi) listens to. This backdoor provides several functionalities like executing commands, downloading files, removing files, and leaking data. One of the most remarkable features of this backdoor is its ability to open a shell on the IoT spectrum sensor. Additionally, the backdoor enables the attacker to retrieve the contents of a file on the Raspberry Pi and transfer it to their machine [5].
- TheTick [9] is the most comprehensive backdoor among these three types of Backdoors due to its complex behaviors. It operates through a Command and Control (C&C) server structure, but with a unique feature - TheTick has the ability to support multiple clients simultaneously and switch between them without causing any disruption. To elaborate further, multiple IoT spectrum sensors can be infected and act as simultaneous clients. Additionally, the attacker has the capability to send commands to each client individually [5].

2.2.3 Rootkits

Rootkits is a type of malware that is designed to hide in the operating system. They give attackers remote access and control without the user's knowledge. By manipulating core components or the kernel of the operating system, rootkits make their presence in the system imperceptible, allowing attackers to steal sensitive information, abuse system resources, and perform malicious actions.

- Beurk [10]. This particular preload rootkit is specifically designed to prevent detection and debugging. It comes with a range of features such as concealing processes, and logins and bypassing various types of analysis such as unhide, lsof, ps, ldd, and netstat. Additionally, it can hide pseudo-terminal backdoor clients, directories, and files. It also has a real-time log cleanup feature that operates on utmp/wtmp [5].
- Bdvl [11]. Vlany-based rootkit preloading is aimed at achieving a more efficient and manageable rootkit that is free from malfunctions. It has a wide range of functions, including hidden backdoors that enable multiple connection methods, as well as keylogging and the theft of passwords and files [5].

2.2.4 Coinminer

Coinminer is a type of harmful software that infiltrates a computer system and uses the device's processing power, particularly the CPU and RAM, to mine for digital coins such as Monero or Zcash. The malware is designed to be extremely persistent, often installing one of several open-source miners without the knowledge or consent of the victim. To avoid detection, these malicious miners often include timer settings or limit the amount of CPU usage to remain undetected. Despite their stealthy behavior, these coin miners can cause significant damage to a victim's computer system, resulting in slower performance, overheating, and even hardware failure.

- XMRig [12] is a mining software that is open-source and is most commonly used for mining Monero, a privacy-focused cryptocurrency. This mining tool is highly efficient and can be used across multiple platforms including Windows, Linux, and macOS. It can mine using both CPUs and GPUs and offers a variety of customization options. However, it is important to note that XMRig may be used for illegal mining activities by malware, so users should ensure the security of the source when downloading and using the software.

2.2.5 Ransomware

Ransomware is a type of malware that encrypts a victim's files or system and demands a ransom for the decryption key. Victims cannot access their data once files are encrypted. This can have serious implications for individuals, businesses, and institutions. Ransomware typically spreads through email attachments, malicious downloads, or exploits.

Attackers exploit victims' urgency and fear to force them to pay a ransom. Therefore, it is crucial to protect against ransomware attacks and regularly back up data.

- Ransomware-PoC [13]. The encryption functionality of this particular ransomware is similar to that of a typical one, with the exception that it doesn't rely on a C&C server to operate. Instead, the encryption keys are integrated into the source code. To encrypt all files present on the Raspberry Pi, Ransomware-PoC uses an AES 256 key. Afterward, the AES 256 key is encrypted using an RSA public key, and the encryption algorithm scans through the system directories, encrypting the content of all valid file extensions [5].

2.3 Malware Analysis Methods

Malware analysis is divided into two phases. The first is the anomaly detection phase, in which the system first catches the presence of malware. Next comes the malware classification phase, where the security system attempts to classify each threat sample into one of the relevant malware families [14]. The analysis methods used in the anomaly detection and malware classification phases are described below.

2.3.1 IoT Anomaly Detection

One way to detect IoT anomalies is through behavioral fingerprinting, which captures unique patterns in device behavior. By analyzing behavioral fingerprinting, deviations from normal behavior can be identified as potentially malicious. This approach creates new defense dimensions against the ever-changing threat landscape. According to [15], there are five primary behavior sources for the anomaly detection scenario.

(1) Network Communications.

One of the core characteristics of IoT devices is that they communicate over a network. By monitoring the network activity of the device, including incoming and outgoing data traffic, communication protocols, and connection patterns, it is possible to build a fingerprint of the device's network behavior. Unusual network traffic patterns can indicate malicious activities.

(2) Hardware Events.

Hardware events, such as startup, shutdown, and external device connection, are unique and stable throughout the device's lifecycle. By monitoring the pattern and frequency of these events, it is possible to detect unusual activity by capturing a fingerprint of the device's hardware behavior.

(3) Resource Usage.

Malware can cause unusual use of device resources, such as unusual CPU, memory, and storage usage. By monitoring resource usage, it is possible to create a behavior fingerprint to detect anomalies.

(4) Software and Processes.

Each device has unique software configurations and processes. By monitoring the software applications and processes running on the device, it is easy to capture a fingerprint of the device's software behavior. Unusual software installation or process activity can mean the presence of malware activities.

(5) Sensors and Actuators.

IoT devices are often equipped with a variety of sensors and actuators. By observing the usage patterns of these sensors and actuators, a fingerprint of the sensor-actuator behavior of the device can be formed to spot behavior that does not match expectations.

Combining these behavioral sources, behavioral fingerprinting provides a multi-dimensional, comprehensive approach to detecting potential malware activity in IoT devices. By understanding the normal behavior pattern of the device, it is possible to more accurately identify abnormal behavior that does not comply with that pattern, thereby strengthening the IoT security strategy.

2.3.2 IoT Malware Classification

Malware classification is a complex task that involves various steps, one of which is the selection of feature vectors. Feature vector selection is a critical process that entails picking out the most relevant attributes from raw data for more efficient analysis and pattern recognition. The goal is to improve the accuracy and efficiency of classification algorithms, which in turn helps in detecting and preventing malware attacks. Feature vector selection methods can be categorized into two types: static and dynamic analysis [14].

The dynamic analysis method involves executing malware samples and monitoring their behavior while observing any changes in the execution environment. However, due to the high complexity of the environment setup, it takes longer to perform a complete execution of all malware samples and observe their results. Nevertheless, this approach provides the safest, most efficient, and most reliable analysis results that provide insight into the behavioral characteristics of malware.

Static analysis involves detecting and analyzing malware by examining its executable metadata, assembly code instructions, and binary content. It is less expensive and faster than dynamic analysis, and it can reveal malware's structure, attack characteristics, and possible functionality.

In summary, dynamic analysis provides detailed information on malware behavior but takes longer, while static analysis offers quick detection and preliminary analysis, providing critical information in a shorter time frame.

2.4 Machine Learning and Deep Learning Techniques

2.4.1 Overview

ML and DL algorithms are highly advanced and sophisticated techniques that offer superior capabilities in various domains. They are particularly useful in complex tasks such as anomaly detection and malware classification, where traditional methods may fall short.

ML/DL algorithms fall under two categories: supervised and unsupervised learning. Supervised learning aims to predict output labels or values based on input data and includes classification and regression techniques. Classification algorithms like *Decision Tree* (DT), *Random Forest* (RF), *Logistic Regression* (LR), *Naïve Bayes* (NB), and *Support Vector Machines* (SVM) classify unknown data based on training data. Regression algorithms like linear and polynomial regression predict continuous numeric outputs [15].

Unsupervised learning algorithms are used to discover patterns, structures, and features in unlabeled data. Some common tasks include clustering (dividing data into similar groups), dimensionality reduction (reducing the dimensionality of the data while retaining important information), and anomaly detection. Three popular anomaly detection algorithms are *One-Class SVM* (OC-SVM), *Local Outlier Factor* (LOF), and *Isolation Forest* (IF) [15].

From a DL perspective, Recurrent Neural Networks (RNNs) like *Long Short-Term Memory* (LSTMs) networks and *Gated Recurrent Unit* (GRU) networks are popular for anomaly detection and malware classification. Along with RNNs, Artificial Neural Networks (ANNs) are also widely used. For instance, *Autoencoders* are useful for detecting anomalous behavior and reducing dimensionality, while *MLP* can be used for malware classification [15].

2.4.2 Autoencoder

In this project, the methodology from [16] is adopted, which involves utilizing Autoencoder as the model for anomaly detection. This choice is based on its optimal balance between simplicity and effectiveness.

The architecture of an Autoencoder, depicted in Figure 2.2, comprises two main components: an encoder and a decoder. The encoder's role is to transform the input data into a condensed, low-dimensional representation in the latent space. Conversely, the decoder's function is to reconstruct the input from this encoded format.

The training process of the Autoencoder is exclusively conducted using normal data. During this phase, the model learns to identify the common patterns found in normal data. The training objective is to minimize the reconstruction error, which is the difference between the original input and the reconstructed output. In an ideal scenario, a well-trained Autoencoder is adept at reconstructing inputs that resemble its training data, thereby exhibiting low reconstruction errors for normal instances. When it encounters

data that deviates from the training data. Such unfamiliar, anomalous inputs are not reconstructed accurately, resulting in significantly higher reconstruction errors. By setting an appropriate threshold for these errors, the model can effectively detect anomalies.

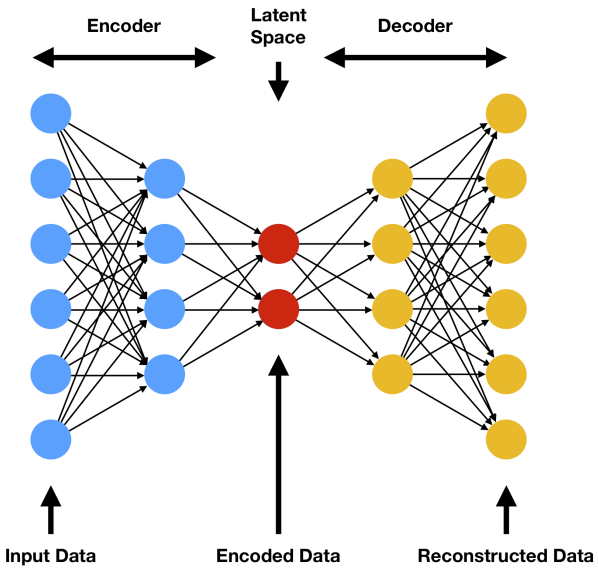


Figure 2.2: Architecture of Autoencoder [17]

2.4.3 MultiLayer Perceptron

For malware classification, this project employs a MLP, aligning with the approach used in [18], due to its powerful and flexible capabilities.

The MLP model can be adapted for a variety of complex problems by adjusting the network structure, including the number of layers and neurons. The architecture of the MLP as shown in Figure 2.3, consists of an input layer, some hidden layers, and an output layer. Hidden layers enable the MLP to learn complex patterns and non-linear relationships in the data. Applying the activation function after the hidden layers introduces non-linearity, enhancing the MLP's ability to process complex data.

MLP training utilizes a dataset comprising data under malware attacks and in a normal state. Throughout the training process, the MLP model adjusts its weights and biases to reduce the value of the CrossEntropyLoss, which reflects the difference between predicted and actual labels. Therefore, the MLP model gradually learns to recognize patterns and features in the data. Upon completion of training, the model has minimized the difference between predicted and actual labels, enabling it to distinguish among different malware attacks and normal state.

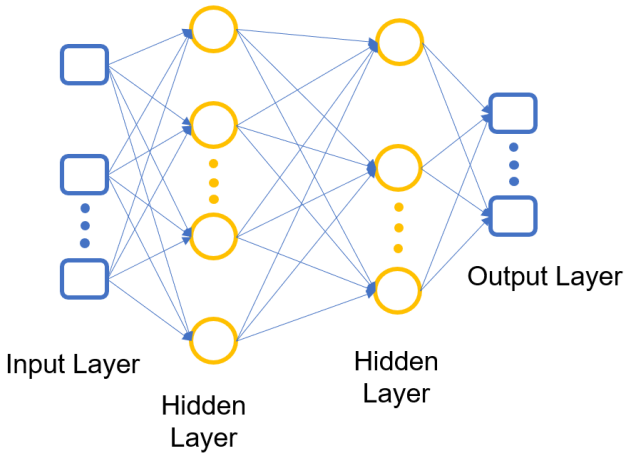


Figure 2.3: Architecture of MLP

2.5 Federated Learning

ML/DL excels in anomaly detection and malware classification tasks by extracting latent patterns from supervised or unsupervised data. However, the traditional ML models discussed in previous section are centralized, implying that they collect, store, process and train data in a single location.

In the traditional ML/DL pipeline, as shown in Figure 2.4, data are collected by individual devices and then upload to a central sever for model training and evaluation. This centralized approach raises substantial privacy and security concerns, especially when dealing with sensitive data.

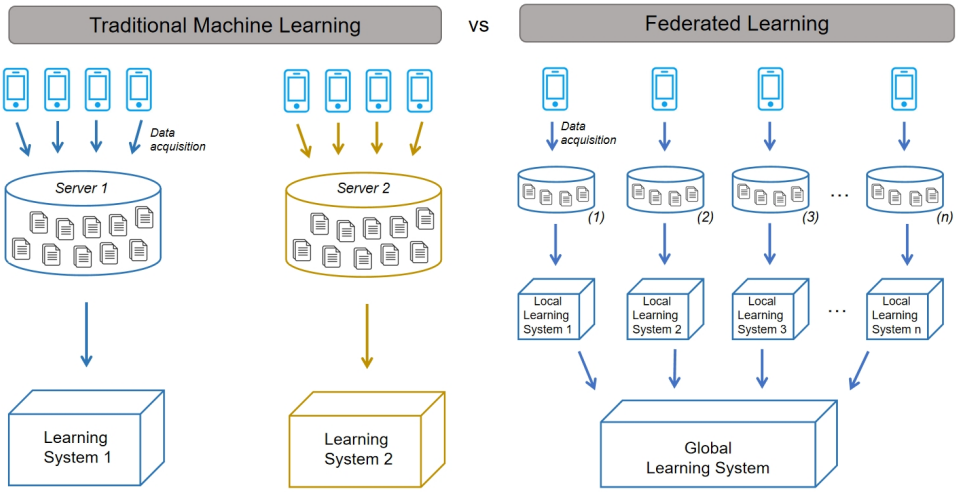


Figure 2.4: Traditional ML (left) and FL (right)

On the other hand, traditional ML/DL techniques face scalability problem with the rapid

development of IoT where enormous amount of data generated. It becomes increasingly difficult for users to collect and analyse tons of executables and trace data at a centralized site.

These limitations of ML/DL techniques, mainly privacy and scalability concerns, have prompted the emergence of innovative solutions such as FL. In FL, as shown in Figure 2.4, each node does not upload data to the central server, but stores data and trains the model locally. This decentralized approach mitigates privacy risks associated with centralizing sensitive data, and it efficiently scales with the growing volume of data generated by the IoT.

In the following, FL, including CFL and DFL, is briefly introduced, explaining the rationale behind adopting DFL as the preferred approach in this project.

2.5.1 Centralized Federated Learning

CFL is a type of FL where a central server orchestrates several steps of the algorithms and coordinates all the participating nodes throughout the learning process. Compared to traditional ML, CFL enables collaborative model training across decentralized devices while keeping data localized.

As depicted in Figure 2.5, each participating node collects data and trains its model locally and sends its model parameters to the central server, which aggregates and updates the global model. Subsequently, the updated model parameters are transmitted back to the individual nodes. After multiple iterations of training and aggregation, a converged global model is integrated by the central sever.

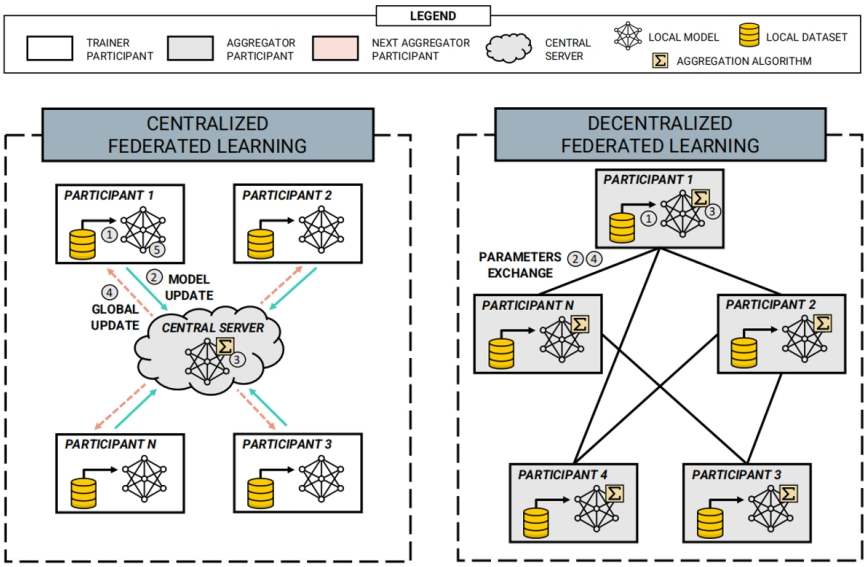


Figure 2.5: CFL (left) and DFL (right) [19]

Due to its centralized nature, CFL still may encounter challenges related to data privacy, as all local updates are transmitted to the central server. Additionally, because of its reliance on a central sever, CFL is exposed to single point of failure and communication bottlenecks. To solve these issues introduced by centralization, researchers proposed DFL.

2.5.2 Decentralized Federated Learning

DFL, another type of FL, characterized by the absence of a central server. In DFL, participating nodes coordinate themselves to obtain the global model.

As depicted in Figure 2.5, each node taking part in the learning process collects and processes data independently. Every node trains its model using its data and sends the model updates or gradients directly to other interconnected nodes, without involving any central server. The DFL approach is highly flexible and accommodates various network topologies including fully connected, ring, star, and random topology. Each topology may have a different impact on the performance of the learning process.

DFL offers numerous advantages over traditional approaches like ML, DL, and CFL. One of the most notable benefits of DFL is that the raw data is not transmitted during model training and updates, which greatly mitigates the risks of data breaches. Moreover, DFL eliminates the single-point-of-failure issue that is inherent in centralized models. This is because model updates are exchanged only between interconnected nodes without the need for orchestration by a central server. Due to its enhanced security, privacy, and resilience, this project used DFL for model training.

2.6 Fedstellar

Fedstellar is an open-source platform for training FL models in a decentralized, semi-decentralized, and centralized fashion across many physical and virtualized devices. Provided with user-friendly frontend (see Figure 2.6), users can setup their experiments easily. With this interactive platform, users are allowed to create federations by customizing parameters such as the number and type of devices engaged in training FL models, the network topology connecting them, the specific ML algorithms employed, and the datasets that participants train on.

Except for these basic functionality, Fedstellar provides an advanced mode, where users can test their model's robustness. As shown in Figure 2.7, users can apply predefined attacks such as label flipping, sample poisoning, and model poisoning.

Figure 2.6: Fedstellar User Interface

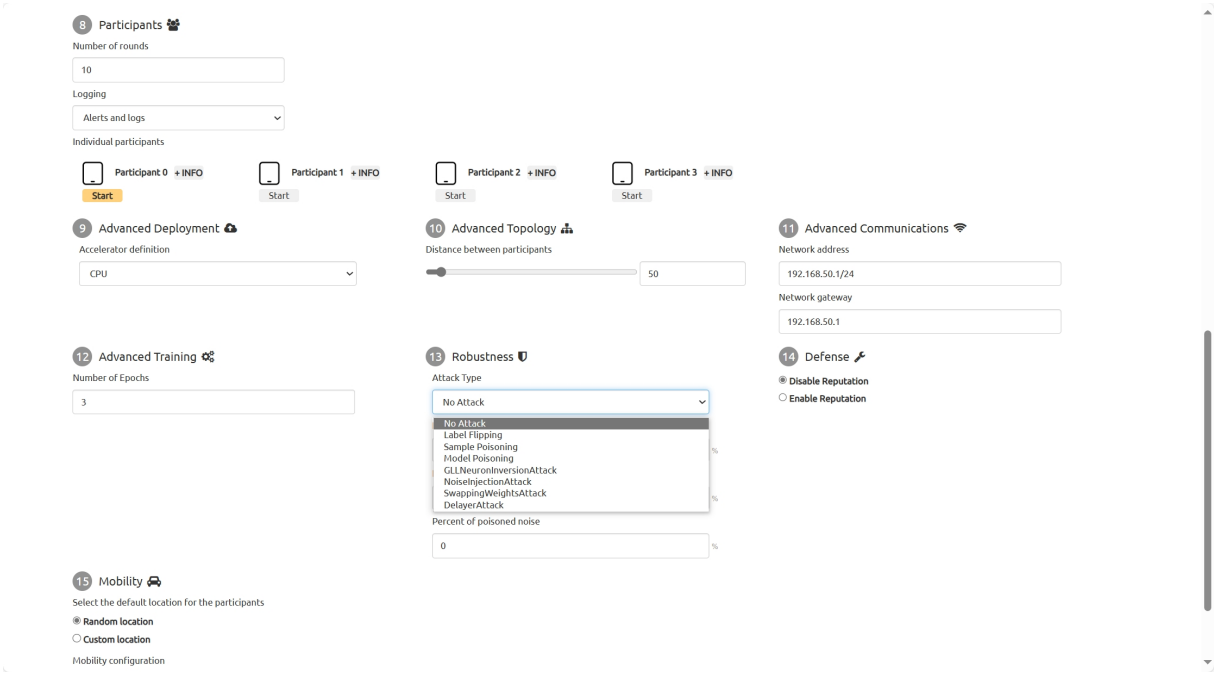


Figure 2.7: Fedstellar Advanced Mode

Moreover, Fedstellar provides real-time monitoring capabilities, allowing users to track both model and network performance through a range of informative metrics (see Figure 2.8). This facilitates comprehensive insights into the ongoing training process.

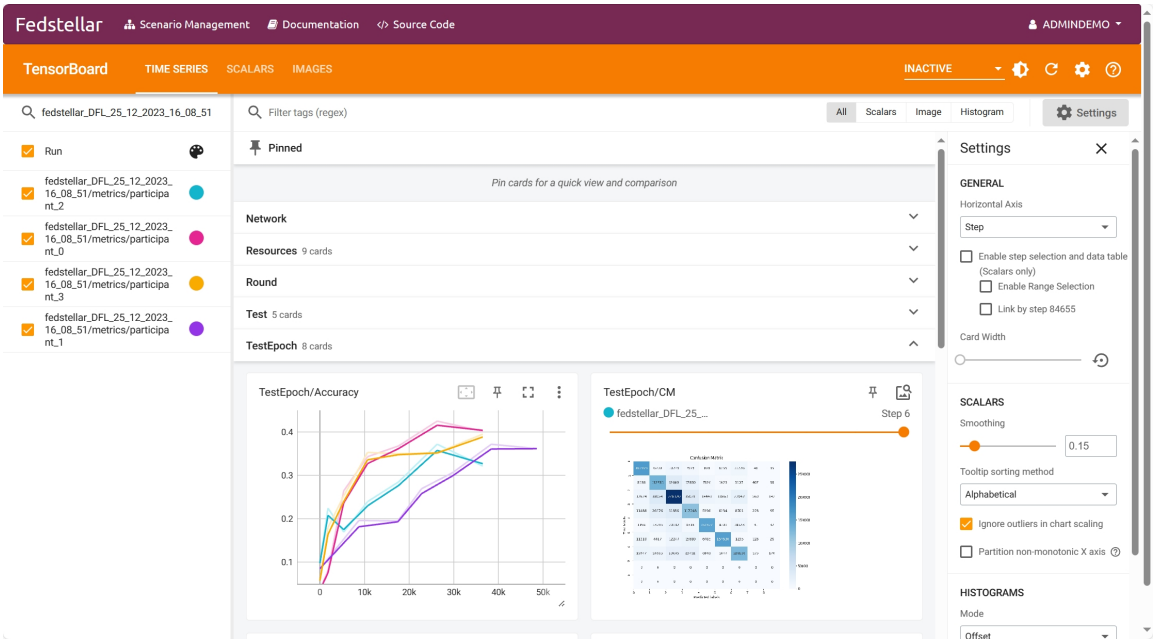


Figure 2.8: Fedstellar Real-time Monitoring Metrics

2.7 Feature Extraction Techniques

2.7.1 Bag-of-words

Bag-of-words (BoW) is a feature extraction technique widely applied in text classification. The core idea of this method is to transform text (for instance, system call data) into a set of tokens (or "words") and to count the frequency of each token within the text. As a result, each text can be represented as a vector, where each dimension corresponds to a specific token, and its value is the occurrence frequency of that token in the text.

Suppose there is a situation where need to analyze the system call logs. If there are two files, each containing ten system call entries, then it is possible to use the BoW method to convert the system call data from each file into a feature vector. This will result in two distinct feature vectors, each representing the system call data of one file.

For instance, suppose the first file contains the following sequence of system calls:

- open, read, write, close, open, read, write, open, read, close.

The second file contains:

- read, write, open, read, write, close, read, write, open, read.

Applying the BoW method, the first step is to identify all distinct system calls, such as open, read, write, and close, and then count how many times each call occurs in each file. Therefore, the feature vectors for the two files look like the following:

- File 1: [open: 3, read: 3, write: 2, close: 2]

- File 2: [open: 2, read: 4, write: 3, close: 1]

The numbers in each vector represent the count of the respective system calls in that file. This representation allows us to quantify system call data without considering the order of calls. As can be seen from the above example, the BoW method is advantageous for its simplicity and effectiveness in feature extraction and data representation.

2.7.2 Entropy with Relation to Files

Entropy is a measurement of randomness in a given set of data. There are different ways to calculate entropy, among which, Shannon Entropy is one of the commonly used. The Shannon Entropy is defined by the following formula:

$$\mathrm{H(X)} = -\sum_{i=1}^{m} p_i \log_2(p_i) \tag{2.1}$$

The resulting Shannon Entropy value ranges between zero and eight. The closer the value is to zero, the more orderly or non-random the data is. The closer the value is to eight, the more random or non-uniform the data is.

The focus of this project lies in the entropy value of files, which serves as a metric for determining the degree of randomness in a file. As encryption algorithms' output usually consists of random data, entropy value of a file is a useful metric in detecting whether the file has been encrypted or not. However, certain file types inherently have high entropy values. Therefore, relying solely on entropy may not be adequate for encrypted file detection. To address this, one approach could be comparing the entropy of a specific file to the average entropy of its file type.

Chapter 3

Related Work

This chapter offers a thorough review of the literature in the fields of anomaly detection and malware classification. Additionally, it encompasses an in-depth examination of related works that address these tasks through the implementation of FL approaches. Concluding this chapter, this work presents the key insights obtained from the comprehensive literature review. These insights serve as the foundational inspiration guiding the methodology and approach of the research.

3.1 Anomaly Detection and Malware Classification

Anomaly detection and malware classification methods can generally be divided into two main categories: static and dynamic. Table 3.1 provides a summary of these solutions, with more detailed explanations given below.

Static detection methods operate without executing the potentially malicious executable files. They rely on extracting various file attributes such as text strings, byte sequences, and imports of dynamically linked libraries (DLLs), among others. By comparing these attributes against known malicious signatures, malware can be identified. In previous research, different approaches have been explored to enhance the accuracy of static detection. For instance, [20] extracted consecutive printable characters and byte sequences from binary files. They employed three distinct data mining algorithms to effectively distinguish between malicious executables and clean programs. Drawing inspirations from text categorization, [21] identified malicious code based on character n-grams. A more comprehensive method, presented in [22], integrated multiple features, including byte entropy, import address table, printable strings, and portable executable (PE) metadata. By utilizing these combined features, they built deep neural network classifiers to effectively detect malware. In the field of Android mobile, [23] applied a similar methodology by extracting various attributes such as strings, method opcodes, method application programming interface (API), shared library function opcode, permission, component, and environmental attributes. Using those features, a DL model was trained to effectively identify Android mobile malware. In an IoT scenario, [24] stored operational code sequences

Table 3.1: Comparison of Related Work Regarding Anomaly Detection and Malware Classification

Method type	**Work**	**Scenario**	**Feature type**	**Post-processing**	**Algorithm**	**Performance**
Static	[20] (2000)	Computer	Printable characters, byte sequences	ML	RIPPER, NB, Multi-Classifier	Acc: 0.97
Static	[21] (2004)	Computer	Character n-grams	ML	KNN	Acc: 0.98
Static	[22] (2015)	Computer	Byte entropy, import address table, printable strings, PE metadata	DL	DNN	TPR: 0.95
Static	[23] (2018)	Mobile phones	Strings, method opcodes, method API, shared library function opcode, permission, component, environmental attributes	DL	Multimodal Neural Networks	Acc: 0.98
Static	[24] (2018)	IoT	Operational code sequences	DL	CNN	Acc: 0.98
Static	[25] (2020)	IoT	Operational code sequences	DL	Hybrid Neural Networks	Acc: 0.97
Dynamic	[26] (2016)	Computer	System calls	DL	Hybrid Neural Networks	Acc: 0.89
Dynamic	[27] (2017)	Mobile phones	System call logs	ML	NB, RF	F1-score: 0.98
Dynamic	[28] (2019)	Computer	System call sequences, byte sequences	DL	Hybrid Neural Networks	TPR: 0.96
Dynamic	[29] (2020)	IoT	Network traffic	DL	LAE, BLSTM	Acc: 0.99
Dynamic	[30] (2020)	IoT	Memory, network, system calls, virtual file system, process	DL	CNN	Acc: 0.99
Dynamic	[5] (2022)	IoT	CPU, virtual memory, network, file system, scheduler, device drivers, random number generation	ML	OC-SVM, IF, LOF	Average TNR&TPR: 0.90
Dynamic	[31] (2022)	Computer	Memory	ML	NB, RF, DT, LR	Acc: 0.99
Dynamic	[32] (2023)	IoT	System calls	ML	KNN, DT, RF, SVM	Acc: 0.98
Dynamic	[33] (2023)	Computer	API calls	DL	ResNet50v2, MobileNetV2	F1-score: 0.98

Acc: accuracy; TPR: true positive rate; FPR: false positive rate; TNR: true negative rate.

in a vector space to generate a graph. Subsequently, they applied a deep Eigenspace learning method to classify applications as either malicious or benign. Extending this approach, [25] introduced a method for cross-architecture IoT malware detection. In this framework, operational code sequences were extracted from multi-architecture samples and a large hybrid neural network consisting of two sub-networks was trained to detect malware. While these static detection methods have proven effective, they can be vulnerable to obfuscation techniques. These techniques can modify file attributes, enabling them to bypass conventional detection methods.

In contrast, dynamic detection methods involve executing the malware to observe its behavior and determine whether the code is malicious. This behavior can be scrutinized from various perspectives, such as system calls, network traffic, and file activities, among others. [26] recorded system call sequences to form binary vector sequences and employed a deep neural network with both convolutional and recurrent layers to classify malware into predefined virus families. Similarly, [27] analyzed system call logs of Android applications and developed a NB model and a RF model to identify malicious apps. [28] expanded

the concept of n-gram from byte sequences to system call sequences. The researchers trained multiple DL models (CNN and RNN) and organized the models in a cluster tree to make the final decision. [29] reduced the feature dimensionality of large-scale network traffic data using the encoding phase of long short-term memory autoencoder (LAE). Subsequently, they trained a deep bidirectional long short-term memory (BLSTM) model to achieve efficient botnet detection. Another research [30] focused on IoT malware, selecting representative features such as memory, network, system call, virtual file system, and process. These were analyzed using a CNN model for malware detection. A different approach was adopted in [5], where a detection framework was designed using device behavioral fingerprinting and ML to identify anomalies and classify a variety of malware types, including botnets, rootkits, backdoors, ransomware, and cryptojackers. This framework monitored kernel events across seven different data sources: cpu, memory, network interface, file system, scheduler, random number generation, and device drivers. In terms of feature extraction, research [31] extracted features from memory dumps and employed a stacked ensemble ML model for malware detection. Meanwhile, a study [34] proposed an association IoT malware detection model which includes a process of extracting the dynamic feature (system call) and feeding the feature into traditional ML models to detect IoT malware. A recent approach proposed in [33] converted sequences of API calls into images, subsequently employing convolutional network architectures for the purpose of malware identification. Unlike static methods, dynamic approaches are less vulnerable to obfuscation techniques.

To delve deeper into the characteristics of different malware types, particularly those targeting IoT devices as discussed in Section 2.2, this work conducted an analysis to understand how specific behaviors are influenced by various malware categories. The findings are summarized in Table 3.2, which has been instrumental in guiding the research to determine the types of data sources that are critical to monitor and collect in the IoT context.

Table 3.2: Behavioral sources affected by different types of malware

Malware type	**Network**	**I/O**	**File system**	**Resource usage**	**System call**	**Kernel events**
Botnets	[35] [36] [37]			[38]	[39] [40] [41]	[35] [40]
Backdoors	[42] [5]		[5]	[5]	[43]	[5]
Rootkits	[44]	[45]	[46] [47]	[48]	[48] [49]	[48] [50]
Ransomware	[51]	[51]	[51]	[51]	[40] [51]	[40] [51]
Coinminer	[52] [53]			[52] [53]	[53]	[5]

As shown in the table, *Botnets* affect behavioral sources including network [35] [36] [37], resources usage [38], system call [39] [40] [41], and kernel events [35] [40]; *Backdoors* affect behavioral sources including network [42] [5], file system [5], resources usage [5], system call [43], and kernel events [5]; *Rootkits* affect behavioral sources including network [44], input or output of block [45], file system [46] [47], resources usage [48], system call [48] [49], and kernel events [48] [50]; *Ransomware* affect behavioral sources including network [51], input or output of block [51], file system [51], resources usage [51], system call [40] [51], and kernel events [40] [51]; *Coinminer* affect behavioral sources including network [52] [53], resources usage [52] [53], system call [53], and kernel events [5].

Based on these insights, this work proposes a more generalized approach that encompasses

all six identified dimensions. The objective is to collect as comprehensive a dataset as possible.

3.2 Federated Learning

While anomaly detection and malware classification techniques continue to evolve, most methods have been designed for computers and devices with relatively high computational power. The main contribution of this work is to propose an effective anomaly detection and malware classification strategy for IoT spectrum sensors, which are typically resource-limited. Furthermore, there is a critical need for solutions that prioritize user security and privacy, a key motivating factor for the present work.

The foregoing discussion has introduced the use of ML and DL methods for detecting malicious software. Building upon this, this project will next explore the employment of FL methods for anomaly detection and malware classification, as detailed in Table 3.3. Compared to ML and DL approaches, FL methods are decentralized, thereby offering protection for user privacy. Notably, in the use case of federated anomaly detection and malware classification, dynamic device behavioral fingerprints have been employed to a lesser extent than static device fingerprints.

[18] employs a dynamic detection approach. This method observes device behaviors such as CPU usage, memory utilization, network interactions, and file system activities during non-infected states. In a subsequent stage, it identifies deviations caused by Subtle System Design Flaws (SSDF) attacks, including delays, confusion, freezes, and five other types of SSDF attacks. [16], [54], and [55] all employ static detection methods. Among them, [16] and [55] utilize communication-based fingerprints, while [54] utilizes APP information. In [16], an Autoencoder and MLP model are utilized for anomaly detection and malware classification. [55], on the other hand, exploits device-type communication profiles to identify malware. Despite [54] using behavioral fingerprints in the form of API calls, it treats them as static fingerprints and employs device features to train a SVM.

Table 3.3: Comparison of Related Work Regarding Anomaly Detection and Malware Classification Using FL Methods

Source	Device Types	Attack Type	Data/Fingerprints	Approach	Prediction	Robustness
[18] 2022	Raspberry Pis	SSDF	Usage of Resources	FL, DL	Anomaly Detection and Classification	Yes, aggregation
[16] 2022	IoT devices	Model poisoning attacks	Communication-based	FL	Anomaly Detection and Classification	Yes, aggregation
[54] 2020	Mobile (Android)	Android Malware	App Information	FL, ML	Classification (SVM)	No
[55] 2019	IoT devices	IoT Malware	Communication-based	FL, ML	Anomaly Detection	No

Compared to DL models, the FL model [16], [18] mainly has two distinctions as shown in Table 3.4: The first one is in the data preprocessing phase, where preprocessing is required. By using the formula (x-min)/(max-min), a unified data scale is achieved.

The second difference lies in the selection of a global threshold after the model training process. Each participant computes a threshold locally and then sends their threshold

values to the coordinator. After a certain screening process, the maximum value is chosen as the global threshold. This global threshold will be used for anomaly detection.

The models in FL cover two scenarios. The first is an anomaly detection scenario, which employs an Autoencoder. The Autoencoder can be divided into an encoder and a decoder: The encoder transforms the input into values defined as encoding dimensions by reducing its dimensionality, while the decoder attempts to map the encoded input back to the original input.

The second is a binary classification scenario using the MLP. The input goes through multiple hidden layers, with these layers applying batch normalization and activation functions, and finally returns a single output neuron.

The rapid growth of spectrum sensors has accelerated the growth of cyber attacks, posing significant challenges to privacy. This limits the applicability of traditional DL methods, but the FL approach has largely addressed this issue while achieving comparable detection performance.

Table 3.4: Comparison of Models

Work	**Approach**	**Model**
[24] 2018	DL	CNN
[18] 2022	FL	(1) Data preprocessing: scaling: $(x - \min)/(\max - \min)$ (2) After training: threshold = $\mu + 3 \cdot \sigma$ (3) Two FL algorithms: – Anomaly detection scenarios: Autoencoder activation function: GELU – Binary classification scenarios: MLP activation function: GELU; training function: logits (BCEwithLogitsLoss)
[16] 2022	FL	(1) Data preprocessing: scaling: $(x - \min)/(\max - \min)$ (2) After training: threshold = $\mu + \sigma$ (3) Two FL algorithms: – Anomaly detection scenarios: Autoencoder activation function: ELU – Binary classification scenarios: MLP activation function: ELU; training function: logits (BCEwithLogitsLoss)

3.3 Summary and Insights

From the comprehensive literature review, several key insights have been derived that inspire the research approach:

- **Dynamic Feature**: The analysis reveals that dynamic approaches are more resilient to obfuscation techniques compared to static methods. Consequently, this project focus on collecting device behavioral data as the source of fingerprints, rather than relying on static attributes.
- **Multi-Dimensional Monitoring**: Different malware exhibit distinct behaviors. To develop a system capable of detecting and classifying a broad spectrum of malware, the strategy involves monitoring device behavior across six key dimensions: network activity, I/O usage, file system activity, resource usage, system calls, and kernel events.

- **Methodology with Privacy Focus**: While most current methods predominantly utilize ML and DL techniques, there is a growing trend toward prioritizing user privacy in anomaly detection and malware classification. Aligning with this trend, this project is inspired by pioneering works in the use of FL approaches. In terms of specific models, this work considers the implementation of Autoencoders for anomaly detection and MLP for malware classification.

Chapter 4

Architecture

This chapter introduces the system architecture, comprising six modules: Control Module, Monitoring Module, Transmission Module, Data Processing Module, FL Module, and Evaluation Module. The architecture is shown in Figure 4.1. Each component is described in details below.

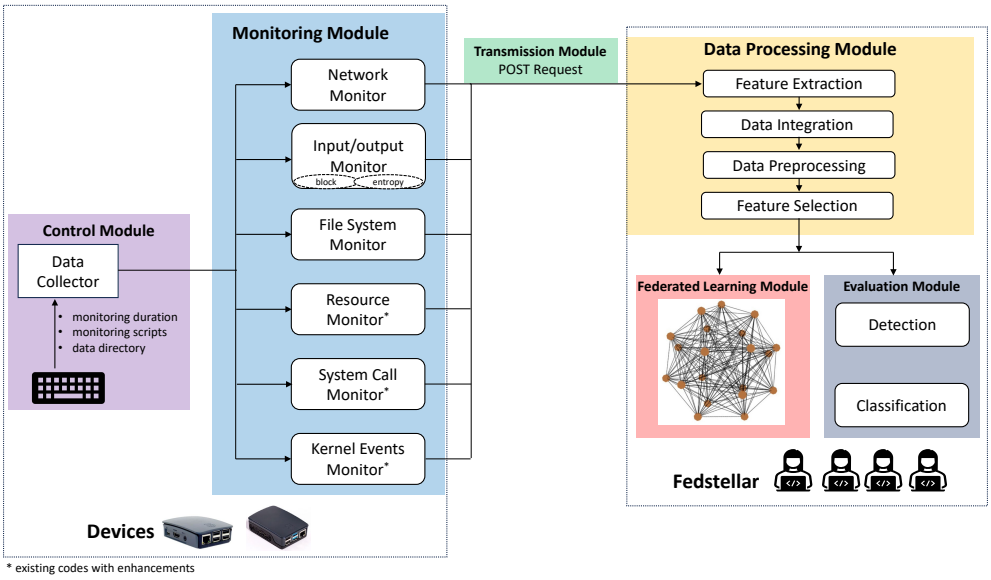


Figure 4.1: System Architecture Overview

4.1 Control Module

The Control Module, as depicted in Figure 4.2, orchestrates the execution of monitoring scripts to gather behavioral data from the IoT devices. It operates interactively, prompting the user to enter three key parameters via the keyboard: the monitoring duration, the specific monitoring scripts to be executed, and the directory path where the output will be stored. Once provided, these parameters guide the Control Module to initiate the appropriate monitoring scripts.

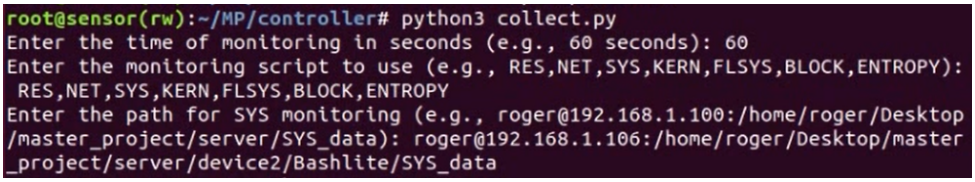


Figure 4.2: Interactive Prompt of the Control Module for Parameter Input

The monitoring scripts, which will be elaborated upon in the following section, are implemented as "systemd" services. This design choice ensures that they can be efficiently managed using the "systemctl" command, allowing for seamless and concurrent operation. The implementation of the network monitoring service, as an instance, is demonstrated in the code excerpt in 4.1.

```
1  [Unit]
2  Description=Monitors events coming from the network (NET)
3  After=multi-user.target
4
5  [Service]
6  Type=simple
7  Restart=on-failure
8  User=root
9  ExecStart=/bin/bash -c 'cd /root/MP/monitors/NET/ && source env/bin/
       activate && python3 network_monitor.py'
10
11 [Install]
12 WantedBy=multi-user.target
```

Listing 4.1: Network Monitoring Service

Upon initiating the control script, the Control Module starts the designated monitoring services. To maintain uninterrupted monitoring, it checks the status of each service at ten-second intervals, relaunching any that have stopped. This procedure is delineated in the code excerpt presented in 4.2.

```
1  def check_services(services):
2      for service in services:
3          status = os.system('systemctl is-active --quiet {service}'.
       format(service=service))
4          if status != 0:
5              os.system("systemctl restart {service} > /dev/null".format(
       service=service))
6
7
8  def thread_work(active_services: array, total: int):
9      check_services(active_services)
10
11 def start_monitor(seconds: int, active_services: array):
12     total = 0
13     for service in active_services:
14         os.system("systemctl start {service} > /dev/null".format(service
       =service))
15     start = time.perf_counter()
```

```
16      while total < int(seconds):
17          time.sleep(1)
18          if total % 10 == 0 and total != 0:
19              t1 = threading.Thread(target=thread_work, args=(
    active_services, total))
20              t1.start()
21          total += 1
22      finish = time.perf_counter()
23      actual_running_time = round(finish-start, 2)
24      print("Finished montioring for {total} seconds.".format(total=
    actual_running_time))
25      for service in active_services:
26          os.system("systemctl stop {service} > /dev/null".format(service=
    service))
```

Listing 4.2: Monitoring Scripts Control

4.2 Monitoring Module

The Monitoring Module is an essential component of the system, comprising six different monitoring scripts. These scripts are designed to perform various functions, including monitoring resource usage, kernel events, system calls, networks, I/O of block, and file system. The resource usage (RES), kernel events (KERN), and system call (SYS) monitors are based on existing scripts [56]. The remaining three scripts were developed by ourselves. The Network (NET) Monitor is responsible for observing network activity behavior and providing detailed insights into the same. The I/O Monitor is further divided into two parts, namely Block Monitor and Entropy Monitor. The former is responsible for monitoring block usage, whereas the latter calculates the entropy of the files. Additionally, the File System (FLSYS) Monitor is utilized to monitor perf events of the file system, providing comprehensive data on the same.

4.2.1 Resource Usage Monitoring

The script constantly monitors the device's hardware performance and gathers a range of behavioral statistics at regular intervals of 5 seconds. These statistics encompass a wide range of aspects such as the device's CPU and memory usage, disk utilization, kernel tracepoint events, and high-performance computing (HPC) [57]. Table 4.1 has been created to showcase the various features that the RES Monitor tracks and reports on.

4.2.2 Kernel Events Monitoring

The KERN Monitor tracks various system metrics at regular intervals of 5 seconds [57]. It keeps a close eye on the disk I/O, CPU usage, kernel memory, and system calls made by the operating system. These features are presented in the Table 4.2.

Table 4.1: Features Tracked by the RES Monitor

time	ext4:ext4_ext_load_extent	armv7_cortex_a7/br_pred/
ioread	ext4:ext4_writepages_result	armv7_cortex_a7/bus_cycles/
iowrite	ext4:ext4_journal_start	armv7_cortex_a7/cpu_cycles/
ioreadbytes	filemap:mm_filemap_add_to_page_cache	armv7_cortex_a7/exc_return/
iowritebytes	jbd2:jbd2_handle_stats	armv7_cortex_a7/exc_taken/
ioreadtime	ext4:ext4_da_update_reserve_space	armv7_cortex_a7/inst_retired/
iowritetime	ext4:ext4_sync_file_enter	armv7_cortex_a7/l1d_cache/
iobusytime	jbd2:jbd2_checkpoint_stats	armv7_cortex_a7/l1d_cache_refill/
read	ext4:ext4_free_inode	net:netif_rx
merge	ext4:ext4_evict_inode	timer:tick_stop
write	ext4:ext4_releasepage	sched:sched_process_exec
merge	ext4:ext4_unlink_enter	sched:sched_waking
memory	block:block_bio_remap	task:task_newtask
net_in	LLC-store-misses	sched:sched_stat_runtime
net_out	LLC-stores	timer:timer_cancel
pkt_in	branch-load-misses	timer:timer_init
pkt_out	branch-loads	timer:timer_start
err_in	dTLB-load-misses	workqueue:workqueue_execute_start
err_out	dTLB-store-misses	branch-instructions
drop_in	iTLB-load-misses	branch-misses
drop_out	filemap:mm_filemap_delete_from_page_cache	bus-cycles
cpu	gpio:gpio_value	cache-misses
cpu-migrations	irq:softirq_exit	cache-references
minor-faults	pagemap:mm_lru_activate	cpu-cycles
page-faults	rpm:rpm_return_int	instructions
L1-dcache-load-misses	fib:fib_table_lookup	context-switches
L1-dcache-loads	raw_syscalls:sys_enter	armv7_cortex_a7/l1d_cache_wb/
L1-dcache-store-misses	random:credit_entropy_bits	armv7_cortex_a7/l1d_tlb_refill/
L1-dcache-stores	kmem:kfree	armv7_cortex_a7/l1i_cache/
L1-icache-load-misses	kmem:kmem_cache_alloc	armv7_cortex_a7/l1i_cache_refill/
L1-icache-loads	kmem:mm_page_alloc_zone_locked	armv7_cortex_a7/l1i_tlb_refill/
LLC-load-misses	kmem:mm_page_free	armv7_cortex_a7/l2d_cache/
LLC-loads	mmc:mmc_request_done	armv7_cortex_a7/l2d_cache_wb/
seconds	writeback:global_dirty_state	armv7_cortex_a7/ld_retired/
block:block_bio_frontmerge	writeback:sb_clear_inode_writeback	armv7_cortex_a7/mem_access/
block:block_dirty_buffer	writeback:wait_on_page_writeback	armv7_cortex_a7/pc_write_retired/
block:block_split	napi:napi_poll	armv7_cortex_a7/st_retired/
block:block_touch_buffer	tcp:tcp_probearmv7_cortex_a7/br_immed_retired/	armv7_cortex_a7/unaligned_ldst_retired/
ext4:ext4_es_lookup_extent_enter	armv7_cortex_a7/br_mis_pred/	armv7_cortex_a7/cid_write_retired/

Table 4.2: Features Tracked by the KERN Monitor

time	kmem:kfree	sched:sched_switch
timestamp	kmem:kmalloc	sched:sched_wakeup
seconds	kmem:kmem_cache_alloc	signal:signal_deliver
connectivity	kmem:kmem_cache_free	signal:signal_generate
alarmtimer:alarmtimer_fired	kmem:mm_page_alloc	skb:consume_skb
alarmtimer:alarmtimer_start	kmem:mm_page_alloc_zone_locked	skb:kfree_skb
block:block_bio_backmerge	kmem:mm_page_free	skb:skb_copy_datagram_iovec
block:block_bio_remap	kmem:mm_page_pcpu_drain	sock:inet_sock_set_state
block:block_dirty_buffer	mmc:mmc_request_start	task:task_newtask
block:block_getrq	net:net_dev_queue	tcp:tcp_destroy_sock
block:block_touch_buffer	net:net_dev_xmit	tcp:tcp_probe
block:block_unplug	net:netif_rx	timer:hrtimer_start
cachefiles:cachefiles_create	page-faults	timer:timer_start
cachefiles:cachefiles_lookup	pagemap:mm_lru_insertion	udp:udp_fail_queue_rcv_skb
cachefiles:cachefiles_mark_active	preemptirq:irq_enable	workqueue:workqueue_activate_work
clk:clk_set_rate	qdisc:qdisc_dequeue	writeback:global_dirty_state
cpu-migrations	random:get_random_bytes	writeback:sb_clear_inode_writeback
cs	random:mix_pool_bytes_nolock	writeback:wbc_writepage
dma_fence:dma_fence_init	random:urandom_read	writeback:writeback_dirty_inode
fib:fib_table_lookup	raw_syscalls:sys_enter	writeback:writeback_dirty_inode_enqueue
filemap:mm_filemap_add_to_page_cache	raw_syscalls:sys_exit	writeback:writeback_dirty_page
gpio:gpio_value	rpm:rpm_resume	writeback:writeback_mark_inode_dirty
ipi:ipi_raise	rpm:rpm_suspend	writeback:writeback_pages_written
irq:irq_handler_entry	sched:sched_process_exec	writeback:writeback_single_inode
irq:softirq_entry	sched:sched_process_free	writeback:writeback_write_inode
jbd2:jbd2_handle_start	sched:sched_process_wait	writeback:writeback_written
jbd2:jbd2_start_commit		

4.2.3 System Call Monitoring

This script records system call data every 10 seconds for the entire device to accurately track the requests made by the device to the OS kernel. The collected system call data is then saved as system call log files and transmitted to the server for further analysis [57].

4.2.4 Network Monitoring

To monitor the network activity of IoT devices, this work employs the Scapy package, a powerful Python-based library for packet manipulation, to capture network traffic data. The network traffic data is collected from the "eth0" network interface in discrete time windows of 5 seconds each. From TCP and UDP packets, key attributes including the packet timestamp, protocol type, source and destination IP addresses, corresponding source and destination ports, and the overall packet length are extracted. Each of these data points is systematically compiled into a row within the resultant file. The critical code underpinning this procedure is shown in 4.3.

```
1  # Function to capture and save network packets
2  def capture_and_save(networkInterface):
3      while True:
4          current_timestamp = int(time.time())
5
6          captured_packets = []
7
8          # Capture network traffic for 5 seconds
9          sniff(iface=networkInterface, prn=lambda packet: packet_handler(
       packet, captured_packets), timeout=5)
10
11         # Call render_content_from_pcap to process and send captured
       packets
12         render_content_from_pcap(captured_packets)
13
14 # Function to process and send captured packets
15 def render_row(packet):
16     if IP in packet and (TCP in packet or UDP in packet):
17         # Extract packet information
18         source_address = packet[IP].src
19         destination_address = packet[IP].dst
20
21         if TCP in packet:
22             protocol = 'TCP'
23             source_port = packet[TCP].sport
24             destination_port = packet[TCP].dport
25         else:
26             protocol = 'UDP'
27             source_port = packet[UDP].sport
28             destination_port = packet[UDP].dport
29
30         packet_time = packet.time
31         packet_length = len(packet)
32
```

```
33          data_to_send = "{},{},{},{},{},{},{}".format(
34              packet_time, protocol, source_address, source_port,
    destination_address, destination_port, packet_length
35          )
36
37          try:
38              # Send the data to the server over HTTPS
39              response = requests.post(server_url, data=data_to_send,
    verify=False)
40
41              # Check the response status code to ensure the data was sent
     successfully
42              if response.status_code == 200:
43                  print("Data sent successfully.")
44              else:
45                  print("Error: {} - {}".format(response.status_code,
    response.text))
46          except Exception as e:
47              print("Error: {}".format(str(e)))
```

Listing 4.3: Monitoring Scripts of Network

4.2.5 Input/Output Monitoring

(1) Block Monitoring

To monitor the block input and output events of the IoT devices, iostat, a tool used for monitoring system input/output statistics related to devices and partitions, was utilized. Several statistics were extracted , such as read_kps, write_kps, avg_queue and await, from iostat. These metrics denote kilobytes read per second, kilobytes written per second, the number of requests waiting for service and the average time for I/O requests to be serviced respectively.

Once the script is launched, the iostat command is executed every 10 seconds. After collecting the desired statistics, these metrics will be compiled into a row in the result file. The code snippet is shown in 4.4.

```
1  # Main monitoring loop
2  while true; do
3      # Display date and time
4      timestamp=$(($(date +%s%N)/1000000))
5
6      # Display I/O statistics using iostat
7      # the name of the block device
8      iostat_output=$(iostat -d -x 10 2 | grep '[0-9]' | tail -n 1)
9      # number of read I/O operations per second
10     read_ops=$(echo "$iostat_output" | awk '{print $4}')
11     # number of write I/O operations per second
12     write_ops=$(echo "$iostat_output" | awk '{print $5}')
13     # kilobytes read per second
14     read_kbs=$(echo "$iostat_output" | awk '{print $6}')
15     # kilobytes written per second
```

```
16      write_kbs=$(echo "$iostat_output" | awk '{print $7}')
17      # average size (in sectors) of the requests sent to the device
18      avgrq_sz=$(echo "$iostat_output" | awk '{print $8}')
19      # average queue length (number of requests waiting for service)
20      avg_queue=$(echo "$iostat_output" | awk '{print $9}')
21      # average time (in milliseconds) for I/O requests to be serviced (
        including queue time)
22      await=$(echo "$iostat_output" | awk '{print $10}')
23      # average time (in milliseconds) for read requests to be serviced
24      r_await=$(echo "$iostat_output" | awk '{print $11}')
25      # average time (in milliseconds) for written requests to be serviced
26      w_await=$(echo "$iostat_output" | awk '{print $12}')
27      # average service time (in milliseconds) for I/O requests
28      svctm=$(echo "$iostat_output" | awk '{print $13}')
29      # percentage of time the device was busy servicing I/O requests.
30      util=$(echo "$iostat_output" | awk '{print $14}')
31
32      finalOutput="$timestamp,$read_ops,$write_ops,$read_kbs,$write_kbs,
        $avgrq_sz,$avg_queue,$await,$r_await,$w_await,$svctm,$util"
33
34      #PUSH to server
35      res=$(curl -sk -X POST -d "$finalOutput" -H "Content-Type: text/csv"
        "$server:$port$directory$mac")
36
37  done
```

Listing 4.4: Monitoring Scripts of Block I/O

(2) Entropy Monitoring

To monitor file creation and modification events, inotifywait, a command-line tool that uses the inotify Linux kernel subsystem to observe changes in one or more files or directories, was employed. Of particular interest is the entropy of these altered files, as malware, such as Ransomware, will encrypt files, resulting in higher file entropy. Consequently, entropy values were computed using the Shannon entropy formula. Given the time and resource-intensive nature of performing calculations on the entire file, the computation was limited to the first 100 bytes of each file.

Once the script is launched, inotifywait continuously monitors file changes until manually stopped. When a changed file is detected, and confirming it is neither a temporary nor log file, the script calculates its entropy. Subsequently, the file path and corresponding entropy values will be compiled into a row in the result file. The code snippet is shown in 4.5.

```
1  caculate_entropy(){
2  entropy=$(${pythoncmd} <<EOF
3  import math
4  from collections import Counter
5  try:
6    with open('$tmp_path', 'rb') as f:
7      data = f.read()
8      counter = Counter(data)
9      total_bytes = len(data)
10     entropy = 0
```

```
    for count in counter.values():
      p_x = count / total_bytes
      entropy += - p_x * math.log2(p_x)
    print(entropy)
except FileNotFoundError:
  pass
EOF
)
}

monitor_write_file(){
  inotifywait -q -m -e modify,create --fromfile ${ScriptDir}/fromfile.txt -r $monitor_dir | while read path action file
  do
    if [[ -f "$path$file" && "$file" != *"log"* && "$file" != *"watchdog"* && "$file" != *".swp"* && "$file" != *".tmp"* && "$file" != *".swx"* ]]; then
      timestamp=$(($(date +%s%N)/1000000))
      file_path="$path$file"
      head -c ${header_byte} "${file_path}" >${tmp_path}
      caculate_entropy

      num="$(echo ${entropy} | awk '{printf "%.6f", $0}')"
      final="$timestamp,$file_path,$action,$num"
      res=$(curl -sk -X POST -d "$final" -H "Content-Type: application/json" "$server:$port$directory$mac")
    fi
  done
}

pythoncmd="python3"
header_byte=100 #the amount of bytes to calculate
monitor_write_file
```

Listing 4.5: Monitoring Scripts of Entropy

4.2.6 File System Monitoring

To monitor file system operations, this project defines all file system-related perf events as the events to monitor, including block:*, ext4:*, filemap:*, jbd2:* and writeback:*. Once the script is launched, it will loop indefinitely to monitor the aforementioned perf events until interrupted by the keyboard. Each data point is systematically compiled into a line in the result file. The code snippet is shown in 4.6. And the Table 4.3 (at the end of this chapter) shows the various features that the File System Monitor tracks.

```
# the value of targetEvents is reduced
targetEvents="block:block_bio_backmerge,block:block_bio_bounce"
timeWindowSeconds=5
timeAcumulative=0

while :
do
```

```
8     timestamp=$(($(date +%s%N)/1000000))
9     tempOutput=$(perf stat --log-fd 1 -e "$targetEvents" -a sleep "
    $timeWindowSeconds")
10    sample=$(echo "$tempOutput" | cut -c -20 | tr -s " " | tail -n +4 |
    head -n -2 | tr "\n" "," | sed 's/ //g'| sed 's/.$//')
11    seconds=$(echo "$tempOutput" | tr -s " " | cut -d " " -f 2 | tail -n
     1 | tr "," ".")
12    timeAcumulative=$(awk "BEGIN{ print $timeAcumulative + $seconds }")
13    finalOutput="$timeAcumulative,$timestamp,$seconds,$connectivity,
    $sample"
14    res=$(curl -sk -X POST -d "$finalOutput" -H "Content-Type:
    application/json" "$server:$port$directory")
15 done
```

Listing 4.6: Monitoring Scripts of File System

4.3 Transmission Module

After collecting data through the Monitoring Module, the data is transmitted to a PC using the client URL (cURL) method due to the limited storage capacity of the Raspberry Pi. Curl, an open source command line tool, enables developers to efficiently transfer data to and from a server. By specifying the location and the data to be sent, the data could be transferred between the Raspberry Pi and the PC using curl. The snippet of the curl command running on the Raspberry Pi is shown in 4.7. This command sends data to the specified PC with POST method.

```
1 #Server and port to push data
2 server="http://192.168.24.132"
3 port="5002"
4 directory="/sensor/"
5 mac=$( cat /sys/class/net/eth0/address | tr : _ )
6
7 #PUSH to server
8 finalOutput="$timestamp,$read_ops,$write_ops,$read_kbs,$write_kbs,
    $avgrq_sz,$avg_queue,$await,$r_await,$w_await,$svctm,$util"
9 res=$(curl -sk -X POST -d "$finalOutput" -H "Content-Type: text/csv" "
    $server:$port$directory$mac")
```

Listing 4.7: Data Transmission Code on Raspberry Pi Side

On the PC side, the Flask framework was employed to create a WSGIServer listening to a designated port. For different monitoring dimensions, data is transferred through different ports, and therefore reduce data conflicts and enhance concurrency. The code snippet running on the PC is shown in 4.8.

```
1 from flask import Flask, request
2 from flask_restful import Resource, Api
3 from gevent.pywsgi import WSGIServer
4
5 app = Flask(__name__)
```

```
api = Api(app)

class sensor(Resource):

    def post(self, sensorid):
        vector = request.data.decode("utf-8")
        ......

def launch_REST_Server():
    if not os.path.exists(data_directory):
        os.makedirs(data_directory)

    api.add_resource(sensor, '/sensor/<sensorid>')  # Route_1
    http_server = WSGIServer(('0.0.0.0', 5002), app)
    http_server.serve_forever()

if __name__ == "__main__":
    launch_REST_Server()
```

Listing 4.8: Data Transmission Code on PC Side

4.4 Data Processing Module

In this module, the previously collected data are processed for model training. The first step is data cleaning, where redundant features from the six monitoring scripts are removed. Timestamp measurement units are standardized for subsequent operations. Additionally, outliers identified as absolute (z) greater than 3 are replaced with values whose absolute (z) equals 3, and all missing values are filled with zeros.

$$z = \frac{x - x_{\min}}{\sigma_{\min}} \tag{4.1}$$

The second step is Feature Extraction. First, features from SYS Monitor, I/O Monitor, and NET Monitor were extracted. For the SYS Monitor, all features from txt files were extracted and stored in csv files. For the I/O Monitor, the `entropy_file_count` variable was calculated. For the NET Monitor, variables containing PacketCount, TotalLength, AverageLength, etc. were extracted. Next, the start time and end time of the timestamps for the six monitoring scripts were determined. Then, the timestamps were divided into multiple intervals of 20 seconds and the average value of each variable within these intervals were calculated. Finally, the data from the six monitoring scripts were merged according to these time intervals.

The third step is Normalization. the following formula was used to normalize the data, mapping all values between 0 and 1. This normalization is beneficial for enhancing ML performance.

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \in \mathbb{R}^d \tag{4.2}$$

The fourth step in the process is Feature Selection. This involved initially eliminating features considered irrelevant, such as time, seconds, connectivity, and timestamp. From an extensive pool of over 500 features, the selection was then narrowed down to 22 features for the Autoencoder and 30 features for the MLP. The details of these selected features are listed in Table 4.4.

This selection was performed using two methods. The first method calculated the minimum cosine distance (`feature_min_distances`) for each feature, which represents the distinctiveness of each feature across different categories of malware and normal states. From this, this project selected the top 22 features where the minimum cosine distance exceeded a threshold of 0.1. The second method involved plotting scatter diagrams for each feature across different types of malware and normal states, and selecting features with significant distinctions based on the data distribution observed in these diagrams.

Table 4.4: Selected Features for Autoencoder and MLP

Task	**Model**	**Selected Features**
Anomaly Detection	Autoencoder	clone_mean, ugetrlimit_mean, ftruncate64_mean, epoll_wait_mean, setresuid32_mean, setresgid32_mean, llseek_mean, kcmp_mean, accept4_mean, umask_mean, armv7_cortex_a7/exc_return/_mean, rt_sigprocmask_mean, pipe2_mean, armv7_cortex_a7/mem_access/_mean, armv7_cortex_a7/l1d_cache/_mean, armv7_cortex_a7/exc_taken/_mean, brk_mean, cache-references_mean, sendmsg_mean, set_robust_list_mean, read_mean, setsid_mean
Malware Classification	MLP	armv7_cortex_a7/br_immed_retired/_mean, armv7_cortex_a7/exc_return/_mean, armv7_cortex_a7/inst_retired/_mean, armv7_cortex_a7/l1i_cache/_mean, armv7_cortex_a7/pc_write_retired/_mean, armv7_cortex_a7/st_retired/_mean, brk_mean, connect_mean, epoll_wait_mean, execve_mean, fstat64_mean, getrandom_mean, kcmp_mean, L1-dcache-loads_mean, L1-dcache-stores_mean, L1-icache-loads_mean, llseek_mean, memory_mean, mmap2_mean, mprotect_mean, munmap_mean, openat_mean, pagemap:mm_lru_insertion_mean, raw_syscalls:sys_enter_mean, raw_syscalls:sys_exit_mean, set_robust_list_mean, set_tid_address_mean, socket_mean, statfs64_mean, uname_mean

4.5 Federated Learning Module

This work designed, developed, and trained two FL pipelines. One was an Autoencoder model used for anomaly detection, with its key code shown in Listing 4.9, and the other was

a MLP model used for malware classification, with its key code illustrated in Listing 4.10.

```
# Encoder layer
self.encoder = nn.Sequential(
    nn.Linear(input_dim, 4),
    nn.BatchNorm1d(4),
    nn.GELU(),
    nn.Linear(4, 2),
    nn.BatchNorm1d(2),
    nn.GELU()

)
# Decoder layer
self.decoder = nn.Sequential(
    nn.Linear(2, 4),
    nn.BatchNorm1d(4),
    nn.GELU(),
    nn.Linear(4, input_dim),
    nn.GELU()
)

self.criterion = MeanSquaredError()
self.anomaly_threshold = 10000
self.reconstruction_error_list = []

self.epoch_global_number = {'Train': 0, 'Validation': 0, 'Test': 0}


def forward(self, x):
encoded = self.encoder(x)
decoded = self.decoder(encoded)
return decoded

def configure_optimizers(self):
return torch.optim.Adam(self.parameters(), lr=0.001)
)
```

Listing 4.9: Code Snippet of Autoencoder Model

```
def __init__(
    self,
    input_size=30,
    hidden_size1=30,
    hidden_size2=30,
    output_size=9,
    learning_rate=1e-2,
    metrics=None,
    confusion_matrix=None,
    seed=None
):

def forward(self, x):
    x = self.l1(x)
    x = torch.relu(x)
    x = self.l2(x)
    x = torch.relu(x)
    x = self.l3(x)
```

```
19      x = torch.log_softmax(x, dim=1)
20      return x
21
22  def configure_optimizers(self):
23      optimizer = torch.optim.Adam(self.parameters(), lr=self.
        learning_rate)
24      return optimizer
```

Listing 4.10: Code Snippet of MLP

Following the previous Data Processing Module, datasets for both the Autoencoder and MLP were obtained. This project integrated these datasets and models with the Fedstellar framework to conduct DFL training and testing. If the results of the Autoencoder indicate that the reconstruction error exceeds a predefined threshold, it may suggest the presence of an anomaly.

This work evaluated the performance of the models using three metrics: precision, recall, and f1-score. Higher values of these metrics indicate better performance of the models in anomaly detection or malware classification.

4.6 Evaluation Module

To validate the robustness of the results of the two FL pipelines, this work designed and executed several related experiments, as shown in table 4.5.

Table 4.5: Related Experiments for Evaluation

Evaluation	**Model**
Evaluation 1: Comparison between ML, CFL, and DFL Approaches.	anomaly detection model, malware classification model
Evaluation 2: Comparison between Different DFL Topologies.	
Evaluation 3: Training Each Node with Data from a Specific Physical Device.	
Evaluation 4: Training Nodes wherein Some of Them Missing a Certain Type of Malware Data.	malware classification model
Evaluation 5: Training Nodes wherein All of Them Missing Certain Types of Malware Data.	
Evaluation 6: Training Nodes wherein the label of a particular malware is flipped to the label of an untargeted malware.	

Both the Autoencoder and the MLP model undergo evaluations 1, 2, and 3. Evaluation 1 and Evaluation 2 are used to analyze the differences in precision, recall, and f1-score when applying ML, CFL, and DFL approaches, as well as different DFL topologies. Evaluation 3 explores the differences in outcomes when each node is trained using data from specific physical devices and uniformly divided data.

Additionally, the malware classification model underwent further evaluations to explore variations in model performance under certain settings: Evaluation 4 explores the differences in outcomes when zero, one, two, or three nodes miss certain types of malware attack data; Evaluation 5 explores the differences in results when each node randomly misses some types of malware attack data compared to previous results; Evaluation 6 investigates the outcomes when the label of a particular malware is flipped to the label of an untargeted malware.

Table 4.3: Features Tracked by the File System Monitor

time	ext4:ext4_write_end	
timestamp	ext4:ext4_writepage	ext4:ext4_get_implied_cluster_alloc_exit
seconds	ext4:ext4_writepages	ext4:ext4_ext_convert_to_initialized_enter
connectivity	ext4:ext4_writepages_result	ext4:ext4_ext_convert_to_initialized_fastpath
block:block_bio_backmerge	ext4:ext4_ext_rm_leaf	ext4:ext4_ext_handle_unwritten_extents
block:block_bio_bounce	ext4:ext4_ext_show_extent	ext4:ext4_zero_range
block:block_bio_complete	ext4:ext4_fallocate_enter	filemap:file_check_and_advance_wb_err
block:block_bio_frontmerge	ext4:ext4_fallocate_exit	filemap:filemap_set_wb_err
block:block_bio_queue	ext4:ext4_find_delalloc_range	filemap:mm_filemap_add_to_page_cache
block:block_bio_remap	ext4:ext4_forget	filemap:mm_filemap_delete_from_page_cache
block:block_dirty_buffer	ext4:ext4_free_blocks	jbd2:jbd2_checkpoint
block:block_getrq	ext4:ext4_free_inode	jbd2:jbd2_checkpoint_stats
block:block_plug	ext4:ext4_fsmap_high_key	jbd2:jbd2_commit_flushing
block:block_rq_complete	ext4:ext4_fsmap_low_key	jbd2:jbd2_commit_locking
block:block_rq_insert	ext4:ext4_fsmap_mapping	jbd2:jbd2_commit_logging
block:block_rq_issue	ext4:ext4_get_reserved_cluster_alloc	jbd2:jbd2_drop_transaction
block:block_rq_remap	ext4:ext4_getfsmap_high_key	jbd2:jbd2_end_commit
block:block_rq_requeue	ext4:ext4_getfsmap_low_key	jbd2:jbd2_handle_extend
block:block_sleeprq	ext4:ext4_getfsmap_mapping	jbd2:jbd2_handle_start
block:block_split	ext4:ext4_ind_map_blocks_enter	jbd2:jbd2_handle_stats
block:block_touch_buffer	ext4:ext4_ind_map_blocks_exit	jbd2:jbd2_lock_buffer_stall
block:block_unplug	ext4:ext4_insert_range	jbd2:jbd2_run_stats
ext4:ext4_alloc_da_blocks	ext4:ext4_invalidatepage	jbd2:jbd2_start_commit
ext4:ext4_allocate_blocks	ext4:ext4_journal_start	jbd2:jbd2_submit_inode_data
ext4:ext4_allocate_inode	ext4:ext4_journal_start_reserved	jbd2:jbd2_update_log_tail
ext4:ext4_begin_ordered_truncate	ext4:ext4_journalled_invalidatepage	jbd2:jbd2_write_superblock
ext4:ext4_collapse_range	ext4:ext4_journalled_write_end	writeback:balance_dirty_pages
ext4:ext4_da_release_space	ext4:ext4_load_inode	writeback:bdi_dirty_ratelimit
ext4:ext4_da_reserve_space	ext4:ext4_load_inode_bitmap	writeback:flush_foreign
ext4:ext4_da_update_reserve_space	ext4:ext4_mark_inode_dirty	writeback:global_dirty_state
ext4:ext4_da_write_begin	ext4:ext4_mb_bitmap_load	writeback:inode_foreign_history
ext4:ext4_da_write_end	ext4:ext4_mb_buddy_bitmap_load	writeback:inode_switch_wbs
ext4:ext4_da_write_pages	ext4:ext4_mb_discard_preallocations	writeback:sb_clear_inode_writeback
ext4:ext4_da_write_pages_extent	ext4:ext4_mb_new_group_pa	writeback:sb_mark_inode_writeback
ext4:ext4_direct_IO_enter	ext4:ext4_mb_new_inode_pa	writeback:track_foreign_dirty
ext4:ext4_direct_IO_exit	ext4:ext4_mb_release_group_pa	writeback:wait_on_page_writeback
ext4:ext4_discard_blocks	ext4:ext4_mb_release_inode_pa	writeback:wbc_writepage
ext4:ext4_discard_preallocations	ext4:ext4_mballoc_alloc	writeback:writeback_bdi_register
ext4:ext4_drop_inode	ext4:ext4_mballoc_discard	writeback:writeback_congestion_wait
ext4:ext4_error	ext4:ext4_mballoc_free	writeback:writeback_dirty_inode
ext4:ext4_es_cache_extent	ext4:ext4_mballoc_prealloc	writeback:writeback_dirty_inode_enqueue
ext4:ext4_es_find_extent_range_enter	ext4:ext4_nfs_commit_metadata	writeback:writeback_dirty_inode_start
ext4:ext4_es_find_extent_range_exit	ext4:ext4_other_inode_update_time	writeback:writeback_dirty_page
ext4:ext4_es_insert_delayed_block	ext4:ext4_punch_hole	writeback:writeback_exec
ext4:ext4_es_insert_extent	ext4:ext4_read_block_bitmap_load	writeback:writeback_lazytime
ext4:ext4_es_lookup_extent_enter	ext4:ext4_readpage	writeback:writeback_lazytime_iput
ext4:ext4_es_lookup_extent_exit	ext4:ext4_releasepage	writeback:writeback_mark_inode_dirty
ext4:ext4_es_remove_extent	ext4:ext4_remove_blocks	writeback:writeback_pages_written
ext4:ext4_es_shrink	ext4:ext4_request_blocks	writeback:writeback_queue
ext4:ext4_es_shrink_count	ext4:ext4_request_inode	writeback:writeback_queue_io
ext4:ext4_es_shrink_scan_enter	ext4:ext4_shutdown	writeback:writeback_sb_inodes_requeue
ext4:ext4_es_shrink_scan_exit	ext4:ext4_sync_file_enter	writeback:writeback_single_inode
ext4:ext4_evict_inode	ext4:ext4_sync_file_exit	writeback:writeback_single_inode_start
ext4:ext4_ext_in_cache	ext4:ext4_sync_fs	writeback:writeback_start
ext4:ext4_ext_load_extent	ext4:ext4_trim_all_free	writeback:writeback_wait
ext4:ext4_ext_map_blocks_enter	ext4:ext4_trim_extent	writeback:writeback_wait_iff_congested
ext4:ext4_ext_map_blocks_exit	ext4:ext4_truncate_enter	writeback:writeback_wake_background
ext4:ext4_ext_put_in_cache	ext4:ext4_truncate_exit	writeback:writeback_write_inode
ext4:ext4_ext_remove_space	ext4:ext4_unlink_enter	writeback:writeback_write_inode_start
ext4:ext4_ext_remove_space_done	ext4:ext4_unlink_exit	writeback:writeback_written
ext4:ext4_ext_rm_idx	ext4:ext4_write_begin	

Chapter 5

Implementation

This chapter presents implementation details on anomaly detection and malware classification tasks. It begins with introducing the devices used in this thesis and describing the experimental setup. Serving as the foundation for model training, the second part of this chapter delves into the processes of feature extraction, data integration, data preprocessing, and feature selection. The last section offers insights into model architectures and the methodologies employed for training and evaluation.

5.1 Setup

In this research, a total of 8 Raspberry Pi devices, consisting of 6 Raspberry Pi 3 and 2 Raspberry Pi 4, were used. Each device was equipped with SDR kits, functioning as the sensing infrastructure. Figure 5.1 presents an example of the sensors utilized in this project. Regarding their configuration, these devices were equipped with either 32 GB or 64 GB SD cards. They operated on an ARM-based CPU architecture and made use of the ElectroSense sensor image for data collection and analysis.

These eight devices were deployed for data collection, capturing behaviors in both a normal state and during eight distinct malware attacks. As detailed in Table 5.1, the selected malware included one botnet (Bashlite), three backdoors (HttpBackdoor, Backdoor, and TheTick), two rootkits (Beurk and Bdvl), one Ransomware (Ransomware-PoC), and one Coinminer (XMRig). Each data collection session lasted for a duration of four hours, resulting in a cumulative dataset of 288 hours.

To simulate the behavior of potential attackers, scripts were created for each malware. These scripts implemented infinite loops to execute actions such as creating files, writing to files, reading files, encrypting files, deleting files, listing directories, or other relevant operations. This approach facilitated the collection of valuable data on the impact of each malware type.

Figure 5.1: An Example of the Sensors Utilized in this Project

Table 5.1: Malware Used in this Project

Malware Family	**Malware Type**	**Source Code**
Botnet	Bashlite	`https://github.com/hammerzeit/BASHLITE`
Backdoor	HttpBackdoor	`https://github.com/SkryptKiddie/httpBackdoor`
	Backdoor	`https://github.com/jakoritarleite/backdoor`
	TheTick	`https://github.com/nccgroup/thetick`
Rootkits	Beurk	`https://github.com/unix-thrust/beurk`
	Bdvl	`https://github.com/Error996/bdvl`
Ransomware	Ransomware-PoC	`https://github.com/jimmy-ly00/Ransomware-PoC`
Coinminer	XMRig	`https://github.com/xmrig/xmrig`

5.2 Feature Extraction

After collecting raw data on six types of device behaviors, statistical methods were employed. The following describes the methodology for extracting features from each dimension of device behavior.

(1) Network Activity.

The raw data about network activity were captured network packages. To obtain useful features, the data were segmented into 20-second intervals. For each interval, the following steps were taken:

- Computed the total number of packets.
- Calculated aggregate statistics for packet length, including sum, mean, median, minimum, maximum values, and variance.
- Determined the count of distinct source and destination ports used within the interval.

This process enables efficient collection of information on network traffic volume, derivation of essential statistics related to packet characteristics, and acquisition of insights into the diversity of network connections.

(2) SYS Activity.

Files with system call data are often large and contain thousands of entries. The frequency of system calls was calculated by following a set of steps.

- Extracted system call names from every file.
- Stored system call names extracted from each file as a string (separated by spaces).
- Created an array of individual strings, each representing the system call data from one file.
- Used the feature extraction technique BoW to construct unigram feature vectors.

To illustrate, if there are two files, and each contains ten system call entries, the resulting dataset would be an array made up of two strings, where each string encapsulates ten system call names.

(3) Entropy Activity.

When a file is modified or created, its entropy value was calculated. As indicated by Alberto and Chao [58], different file types exhibit distinct entropy scores. For instance, .jpg, .pdf, and .ppt files typically have entropy scores higher than 5.5, while .txt, .csv, and .xls files have entropy scores lower than 5.5. Ransomware encrypts files, resulting in unstructured and random data, leading to high entropy scores. For Ransomware PoC, the average write entropy for all file types is approximately 6. In the experiment, the Ransomware PoC was utilized to encrypt files with low entropy scores, such as .txt files. Detection of a file with an entropy value exceeding 6 may indicate a ransomware infection.

To extract useful features, the following steps were taken:

- Partitioned the data into 20-second time windows.
- Calculated the number of files with entropy values greater than or equal to 6 within each interval.

(4) File System, Resource, Kern and Block Activity.

For the FLSYS, RES, KERN and BLOCK Monitoring Modules, the collected data are the features and do not require additional transformation processes.

5.3 Data Integration

After extracting features from each dimension of device behavior, the feature tables of each device behavior were merged into a single table based on the timestamp.

First, the timestamp units were standardized to seconds, rounding down if there were decimals. The "Timestamp" in the collected table from NET module was renamed to "timestamp", and the "Time" in the collected table from SYS module was renamed to "timestamp".

Then, some overlapping features were found between the KERN, RES, FLSYS modules, as all three modules monitor perf events and some perf events are related to both kernel, resource, and file system, such as "ext4:ext4_free_inode" and "ext4:ext4_writepages_result", which are directly related to ext4 file system operations and also involve managing resources like memory and storage. For these duplicate features, only one of each is retained.

Next, the start time and end time of the timestamp were calculated and the time from start time to end time were divided into numerous continuous intervals of 20 seconds each, known as time_duration, which served as indices. For all rows with "timestamp" belonging to a certain time_duration, the average of these rows was calculated and used as the value for the current time_duration; if no rows with "timestamp" belonging to a certain time_duration, then the value for that time_duration was null. After calculating the averages, the RES, KERN, FLSYS, SYS, NET, ENTROPY and BLOCK modules each resulted in one table, and six tables in total.

Finally, the six tables were merged into a single table with over 400 columns, and further integration was conducted as follows. The above operations are shown in the code snippet 5.1.

```
time_intervals = np.arange(start_time, end_time, time_window)
time_intervals = np.append(time_intervals, end_time)
all_means = []
for name, df in dataframes.items():
    if name in ['entropy', 'net']:
        continue
    df['time_duration'] = pd.cut(df['timestamp'], bins=time_intervals,
   right=False, include_lowest=True)
    means_list = []
    for col in df.select_dtypes(include=[np.number]).columns:
        if col != 'timestamp':
            means_list.append(df.groupby('time_duration')[col].mean().
   rename(col + '_mean'))
    df_means = pd.concat(means_list, axis=1)
    df_means.reset_index(inplace=True, drop=True)
    df_means['time_duration'] = time_intervals[:-1]
    all_means.append(df_means)
    df_means.to_csv(f'C:/DATA/MP/output/{name}_means.csv', index=False)
for df in all_means:
    df.reset_index(inplace=True, drop=True)
merged_df = reduce(lambda left, right: pd.merge(left, right, on='
   time_duration'), all_means)
cols = list(merged_df.columns)
```

```
cols.insert(0, cols.pop(cols.index('time_duration')))
merged_df = merged_df[cols]
merged_df.columns = ['timestamp' if col == 'time_duration' else col for col in merged_df.columns]
merged_with_net = pd.merge(merged_df, dataframes['net'], on='timestamp', how='left', suffixes=('', '_net'))
merged_final = pd.merge(merged_with_net, dataframes['entropy'], on='timestamp', how='left', suffixes=('', '_entropy'))
```

Listing 5.1: Integration Script 1

The data from six dimensions of behavioral sources collected on each device were processed and integrated. Subsequently, the data from eight IoT devices were integrated into one large table. The normal data and malware samples were labeled accordingly to facilitate further data preprocessing. Finally, a large, single table with over 500 columns was obtained. The above operations are shown in the code snippet 5.2.

```
table_list_with_label = []
data_folder = "/Users/xicheng/dataset_all/raw_data_20s"
label_mapping = {
    "normal": 0,
    "httpbackdoor": 1,
    "backdoor": 2,
    "thetick": 3,
    "bashlite": 4,
    "beurk": 5,
    "bdvl": 6,
    "ransomware": 7,
    "xmrig": 8
}

for filename in os.listdir(data_folder):
    if filename.endswith(".csv"):
        for keyword, label in label_mapping.items():
            if keyword in filename:
                filepath = os.path.join(data_folder, filename)
                df = pd.read_csv(filepath)
                df["label"] = label
                cols = df.columns.tolist()
                cols = ["label"] + [col for col in cols if col != "label"]
                df = df[cols]
                table_list_with_label.append(df)

combined_table = pd.concat(table_list_with_label, axis=0, ignore_index=True)
combined_table = combined_table.sort_values(by="label", ascending=True)
```

Listing 5.2: Integration Script 2

5.4 Data Preprocessing

The integrated data obtained in the previous steps underwent simple data preprocessing to enable subsequent feature selection for Autoencoder and MLP. This involved a series of steps, as implemented in 5.3, which can be summarized as follows:

- **Eliminate Useless Features**: Useless features such as time, seconds, connectivity, and timestamp were eliminated.
- **Handle Outliers**: Outliers were handled by calculating the Z-score for all feature columns. Values with a Z-score greater than or equal to 3 were replaced with the mean of that column plus 3 times the standard deviation.
- **Fill Missing Values**: Missing values were filled with 0.
- **Eliminate Constant Features**: Features with variance equal to 0 were removed, that is, constant features.

```
1  # Eliminate Useless Features
2  combined_table.drop(columns=["time_mean", "seconds_mean", "
       connectivity_mean", "timestamp"], inplace=True)
3
4
5  # Handle Outliers
6  X = combined_table.iloc[:, 1:]
7  y = combined_table.iloc[:, 0]
8
9  z_scores = np.abs((X - X.mean()) / X.std())
10
11 X_replaced = X.copy()
12 X_replaced = X_replaced.apply(lambda col: np.where(z_scores[col.name] >=
       3, X[col.name].mean() + 3 * X[col.name].std(), col))
13
14 df = pd.DataFrame(X_replaced, columns=X.columns)
15 df.insert(0, 'label', y)
16 df = df.sort_values(by="label")
17
18
19 # Fill Missing Values
20 df1 = df.fillna(0)
21
22
23 # Eliminate Constant Features
24 X = df1.iloc[:, 1:]
25 y = df1.iloc[:, 0]
26
27 variance_threshold = 0
28 selector_variance = VarianceThreshold(threshold=variance_threshold)
29
30 X_filtered = selector_variance.fit_transform(X)
31 selected_feature_indices_variance = selector_variance.get_support(
       indices=True)
32
```

```
constant_feature_indices = [i for i in range(len(df1.columns[1:])) if i
    not in selected_feature_indices_variance]
print("Constant Feature Indices:", constant_feature_indices)

df1 = pd.DataFrame(X_filtered, columns=df1.columns[1:][
    selected_feature_indices_variance])
df1["label"] = y

new_columns = ["label"] + df1.columns[:-1].tolist()
df2 = df1[new_columns].sort_values(by=["label"])
```

Listing 5.3: Simple Data Preprocessing

5.5 Feature Selection

Following basic data preprocessing, the feature set remained extensive, which posed a challenge due to its high dimensionality. To streamline the training process and reduce the risk of overfitting, a feature selection strategy was employed. The methods of feature selection for the anomaly detection and malware classification tasks are detailed below.

5.5.1 Feature Selection for Anomaly Detection

To effectively differentiate between normal and abnormal data, a thorough analysis and comparison of features across different labels was undertaken. This involved assessing the distribution of each feature using histogram binning and cosine distance as primary metrics to evaluate the similarity or dissimilarity among these distributions. The procedure for evaluating each feature, as implemented in 5.4, can be summarized as follows:

- **Binning Process**: The first step involved segmenting the feature values into 10 distinct bins for each label. This converted the feature data into a histogram format, facilitating easier analysis of distribution patterns.
- **Vector Formation**: The next step was to count the number of samples in each bin, resulting in a ten-dimensional vector representing the distribution of feature values for a specific label.
- **Distance Calculation**: For each vector corresponding to a malware label (any label other than 0), the cosine distance was calculated relative to the vector of normal data (label 0). This quantified the similarity or dissimilarity of the malware data compared to normal data for the feature being examined.

```
def cosine_distance(a, b):
    if np.all(a == 0) or np.all(b == 0):
        return np.nan
    return 1.0 - cosine_similarity(a.reshape(1, -1), b.reshape(1, -1))
    [0][0]
```

```
5
6  def normalize_to_unit_vector(counts):
7      norm = np.linalg.norm(counts)
8      return counts / norm if norm > 0 else counts
9
10 feature_min_distances = []
11
12 # Define the number of bins
13 num_bins = 10
14
15 for feature_column in tqdm(feature_columns, desc="Calculating feature
       distances"):
16     label_0_bins = np.zeros(num_bins)
17     label_0_values = df1[df1['label'] == 0][feature_column].values
18     for value in label_0_values:
19         bin_index = min(int(value * num_bins), num_bins - 1)
20         label_0_bins[bin_index] += 1
21
22     normalized_label_0_bins = normalize_to_unit_vector(label_0_bins)
23
24     min_distance = None
25     min_distance_label = None
26
27     for other_label in df1['label'].unique():
28         if other_label == 0:
29             continue
30
31         other_label_bins = np.zeros(num_bins)
32         other_label_values = df1[df1['label'] == other_label][
       feature_column].values
33         for value in other_label_values:
34             bin_index = min(int(value * num_bins), num_bins - 1)
35             other_label_bins[bin_index] += 1
36
37         normalized_other_label_bins = normalize_to_unit_vector(
       other_label_bins)
38
39         # Calculate the cosine distance between normalized bin counts of
        label 0 and this other label
40         distance = cosine_distance(normalized_label_0_bins,
       normalized_other_label_bins)
41
42         if min_distance is None or distance < min_distance:
43             min_distance = distance
44             min_distance_label = other_label
45
46     if min_distance is not None:
47         feature_min_distances.append((feature_column, min_distance,
       min_distance_label))
```

Listing 5.4: Feature Selection for Autoencoder

After evaluating all candidate features, they were sorted by their minimum cosine distance to normal data. Those exhibiting a minimum cosine distance greater than 0.1 were selected as the definitive set for both training and evaluating the anomaly detection model, culminating in a selection of 22 features. This approach ensures a focus on the most

pertinent features for distinguishing between normal and abnormal data.

5.5.2 Feature Selection for Malware Classification

To effectively distinguish among data under eight types of malware attacks as well as normal data, a method combining visual inspection and calculation of the minimum cosine distance is employed.

(1) Visual Inspection.

In the first step, by visual inspection, 96 features from over 500 features were initially selected. First, each column of data was min-max normalized to a value between 0 and 1, then scatter plots were generated for these features, each plot containing 9 columns representing data under eight types of malware attacks and the normal data. Then the plots where there is a significant difference in the distribution of the 9 columns of dots were filtered out. If the distribution of the 9 columns of dots in the plot differs significantly, it implies a significant difference in the performance under eight types of malware attacks and the normal performance, suggesting that the feature corresponding to the plot plays a relatively significant role in distinguishing 9 states, therefore the feature "armv7_cortex_a7/br_immed_retired/_mean" corresponding to the Figure 5.2 was selected; whereas if the distribution of the 9 columns of dots in the plot is very similar, the feature "net:net_dev_xmit_mean" corresponding to the Figure 5.3 was excluded. Through visual inspection, 96 features were selected initially selected for the second step of selection.

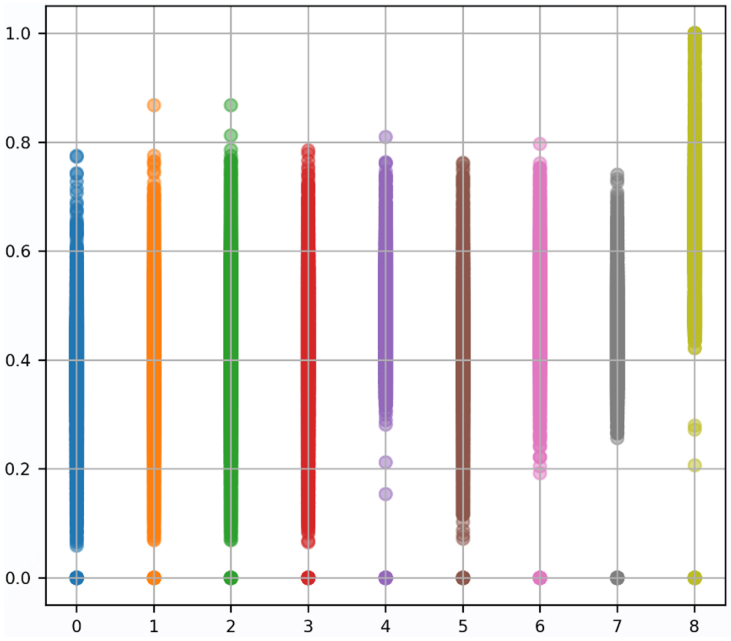


Figure 5.2: Distribution of "armv7_cortex_a7/br_immed_retired/_mean"

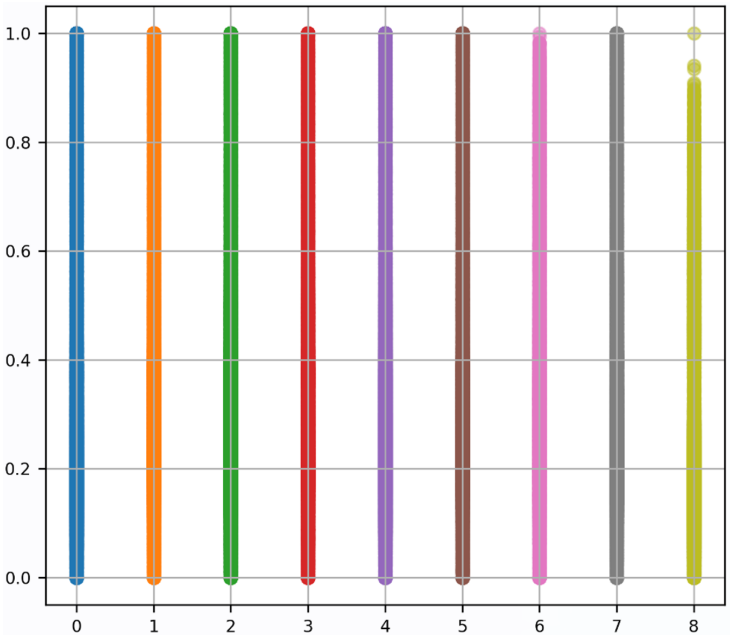


Figure 5.3: Distribution of "net:net_dev_xmit_mean"

(2) Calculation of the Minimum Cosine Distance.

In the second step, 30 features were further selected from the 96 by calculating the minimum cosine distance. This operation is very similar to the feature selection for anomaly detection. The difference is that for each feature, the cosine distances between pairs of the 9 vectors corresponding to data under eight types of malware attacks and normal data were calculated, and then the smallest of the 36 cosine distances was recorded as min_distance. After traversing the 96 features, this project sorted their min_distances from largest to smallest, and chose the top 30 features with the highest min_distance as the final set for training and evaluating the malware classification model.

The reason the first step was not skipped is that doing so would exclude some informative features. For example, a certain feature that distinguishes well among data under six kinds of malware attacks as well as normal data but struggles to distinguish between data under the other two kinds of malware attacks, would have a very small min_distance. If going directly to the second step, such informative features will be eliminated.

By combining the methods of visual inspection and calculating the minimum cosine distance, a better distinction can be made among data under eight types of malware attacks as well as normal data.

5.6 Model Training

After identification of the relevant features, two distinct models were developed for anomaly detection and malware classification:

(1) Anomaly Detection.

The approach to anomaly detection draws inspiration from the methodology outlined in [16]. An Autoencoder served as the core model for this purpose. The encoder part of the Autoencoder was composed of two linear layers: the first layer was fitted with 4 neurons, and the following layer had 2 neurons. Each of these linear layers was succeeded by a batch normalization layer and a GELU activation layer. This configuration of the encoder compressed the input data into a two-dimensional vector. This vector was then fed into the decoder, which aimed to reconstruct the original input data. The decoder mirrored the encoder's two-layer structure, with the first layer containing 2 neurons and the second layer having 4 neurons. Each linear layer in the decoder was also followed by a batch normalization layer and a GELU activation layer.

The Autoencoder model underwent training with normal data for 50 epochs, using the Adam optimizer to minimize the mean square loss. The learning rate was set at 0.001 to achieve a balanced and efficient training process. In the testing phase, the model's performance was assessed by measuring the reconstruction error for each input sample. Samples with a reconstruction error that surpassed a predetermined threshold were identified as anomalies, signaling a potentially abnormal state.

(2) Malware Classification.

The approach to malware classification follows the methodology used in [18], employing a MLP. The model comprises four layers: the first layer is the input layer, receiving an input of size 30, corresponding to the 30 features in the dataset. The second and third layers are hidden layers, which take the output of the previous layer as input and then use the ReLU activation function to enhance the model's ability to handle complex data. The fourth layer is the output layer, which maps the output of the second hidden layer to an output size of 9, as the model is tasked with classifying eight types of malware and normal states. The log-softmax function is used in the output layer, suitable for multi-class classification tasks. Throughout training, the MLP model continuously adjusts its weights and biases to reduce the CrossEntropyLoss value, which indicates the difference between predicted and actual labels. Additionally, the model utilizes the Adam optimizer, which adaptively adjusts the learning rate for a more efficient and stable training process. A learning rate of 0.01 is set to accomplish effective learning and convergence to the optimal solution.

The MLP was trained using data under eight malware attacks and in a normal state, over 180 training epochs. Training would be stopped early if there was no significant decrease in loss on the validation set. During the testing phase, the model predicted the labels of the data, which represented eight types of malware and normal state. Finally, a confusion matrix was outputted. From this confusion matrix, metrics including precision, recall, and f1-score were calculated to evaluate the performance of the MLP in classifying malware.

Chapter 6

Evaluation

This chapter provides a detailed evaluation of the anomaly detection and malware classification models trained on the malware dataset. Initially, three methodologies (traditional ML, CFL, and DFL) are compared. This comparison is aimed at assessing the efficacy of each approach in the context of anomaly detection and malware classification. The analysis then extends to a more focused analysis of the DFL approach, examining how it performs under various network topologies. This exploration is critical to understanding the influence of network structure on the effectiveness of decentralized learning processes. Another key aspect of this chapter is the exploration of the models' performance in scenarios involving non-independent and identically distributed (Non-IID) data. This aspect is crucial as it mirrors real-world situations where data may not follow a uniform distribution. Lastly, the chapter includes supplementary experiments to evaluate the models' resilience against attacks. These experiments provide valuable insights into the robustness of models, offering a gauge of their practical reliability and effectiveness in real-world applications.

6.1 Comparison between ML, CFL, and DFL Approaches

The analysis began with testing the anomaly detection and malware classification models using three distinct approaches: ML, CFL, and DFL.

(1) Anomaly Detection.

For the evaluation of the anomaly detection model (specifically, the Autoencoder), a performance comparison was conducted between ML, CFL, and DFL, focusing on three key metrics: precision, recall, and f1-score. These metrics were crucial in assessing the effectiveness of each approach in identifying anomalies accurately. Precision is calculated as the ratio of true positive results to the total number of positive results, encompassing both correctly identified and incorrectly identified cases. Recall, on the other hand, is the ratio of true positive results to the total number of cases that should have been positively identified. F1-score, serving as the harmonic mean of precision and recall, effectively

consolidates both metrics into a singular, balanced measure. The formulas for these metrics are as follows:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{6.1}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{6.2}$$

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6.3}$$

The results of this evaluation are depicted in Figure 6.1. It is observed that the three models—ML, CFL, and DFL—each exhibit a high level of precision. Notably, the ML model shows a slightly better performance in recall than its CFL and DFL counterparts. In assessing overall effectiveness using f1-score, the ML model stands out with a score of 0.736. This is slightly ahead of the CFL and DFL models, which both show comparable performances with scores around 0.687.

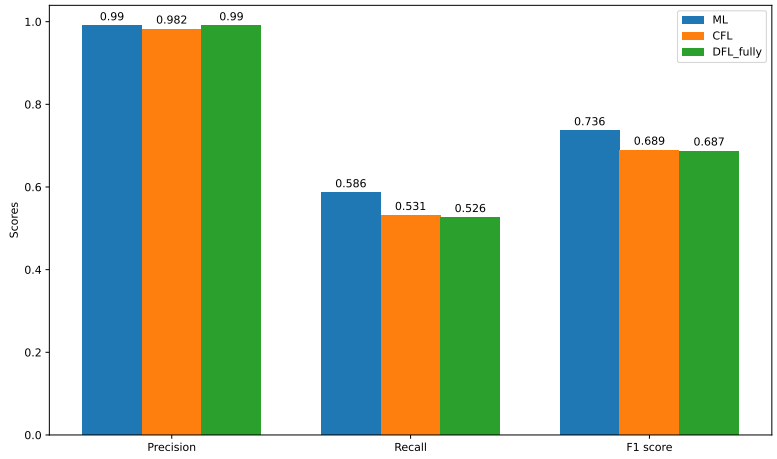


Figure 6.1: Comparison between ML, CFL, and DFL for Anomaly Detection

(2) Malware Classification.

Similar to the anomaly detection, in examining the malware classification model (specifically, the MLP), the precision, recall, and f1-score were compared across ML, CFL, and DFL approaches.

The results, as depicted in Figure 6.2, indicate that the precision, recall, and f1-score for ML, CFL, and DFL are all roughly around 0.70. The ML outcomes are slightly superior to DFL, with a margin within 0.02, while DFL slightly outperforms CFL, again with a margin within 0.02. The discrepancies between ML, CFL, and DFL are not significant, because parameters of the MLP model are identical except for the parameter aggregation required after each round in FL. The marginally better results of ML over DFL might be due to each aggregator not fully representing the overall data distribution as well as the

varying training speeds of different aggregators affecting the final model performance. The reason DFL slightly outperforms CFL may be that each aggregator in DFL independently updates its model, leading to a more diversified feature learning and better classification of different types of malware.

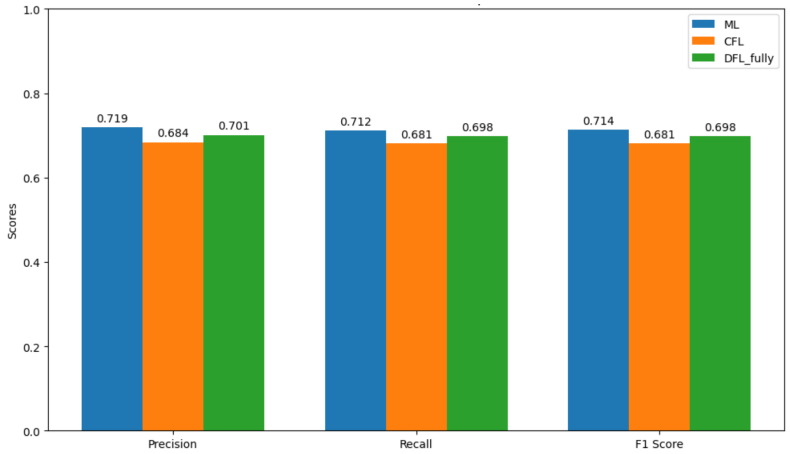


Figure 6.2: Comparison between ML, CFL, and DFL for Malware Classification

In addition, the differences in precision for various types of malware across the ML, CFL, and DFL methods were compared, as illustrated in the heatmap shown in Figure 6.3. This comparison allows us to evaluate the variations in precision both horizontally across the three methods and vertically across different types of malware. A horizontal comparison reveals that each malware type exhibits very similar hue under the three methods, indicating minor differences in their precision. Specifically, except for Coinminer, which shows a precision range of 0.05, and Ransomware, with a range of 0.07, the precision range for each other type of malware remains within 0.04. A vertical comparison shows that variations in hue among different malware types indicating differences in the precision of classification among different malware types, and the precision for Botnet and Rootkits is higher. This is easily understandable since the selected features have varying degrees of matching with different malware types.

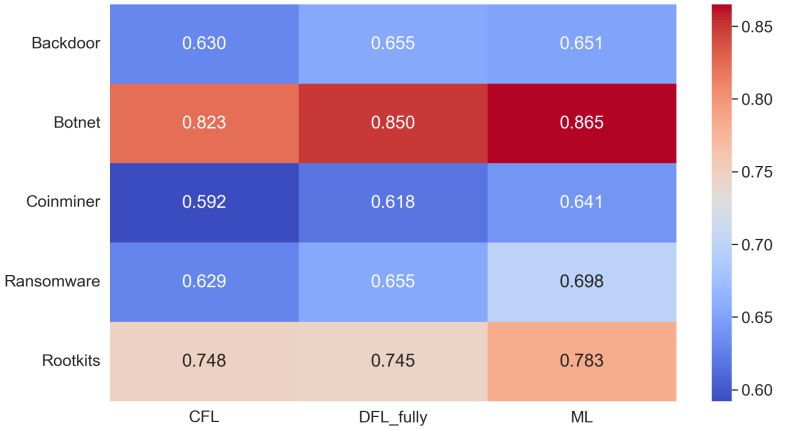


Figure 6.3: Differences in Precision for Different Types of Malware across the ML, CFL, and DFL Methods

Similarly, the differences in recall and f1-score for different types of malware across the ML, CFL, and DFL methods were examined. As depicted in Figure 6.4 and Figure 6.5, horizontally, the recall and f1-score for each malware type do not vary significantly across the three methods, with all ranges within 0.05. Vertically, Botnet, Ransomware, and Rootkit exhibit relatively higher precision, and Botnet and Rootkit also show higher f1-score.

Figure 6.4: Differences in Recall for Different Types of Malware across the ML, CFL, and DFL Methods

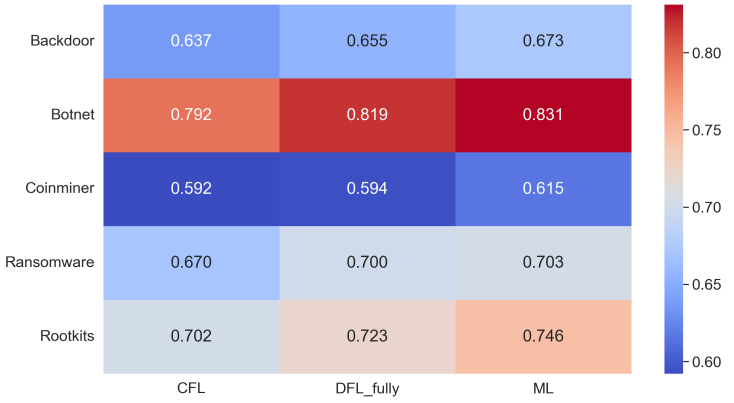


Figure 6.5: Differences in F1-score for Different Types of Malware across the ML, CFL, and DFL Methods

Furthermore, the results of a fully-connected DFL with 4 aggregators were compared to that with 8 aggregators, as shown in Figure 6.6. The former slightly outperforms the latter, with a difference within 0.02, which is due to the reduction in data volume of each aggregator when the data is divided into more slices, increasing the likelihood of each aggregator's data deviating from the overall dataset's distribution.

Moreover, the performance differences in precision, recall, and f1-score for various types of malware between DFL with 4 nodes and DFL with 8 nodes were compared. As can be observed from Figure 6.7, Figure 6.8, and Figure 6.9, the DFL with 4 nodes consistently exhibits higher precision, recall, and f1-score for each type of malware compared to the 8-node DFL. This conclusion aligns with previous findings.

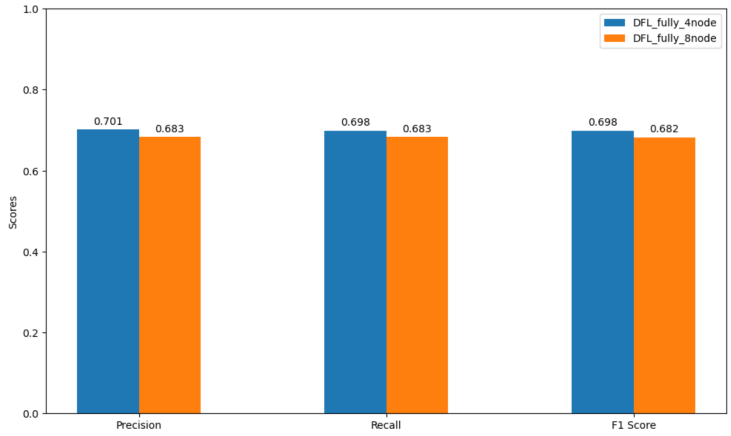


Figure 6.6: Comparison between DFL with 4 Aggregators and 8 Aggregators for Malware Classification

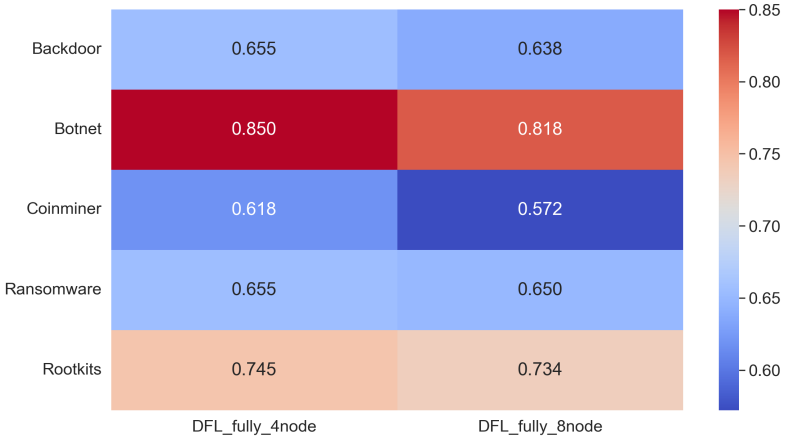


Figure 6.7: Differences in Precision for Different Types of Malware between DFL with 4 Aggregators and 8 Aggregators

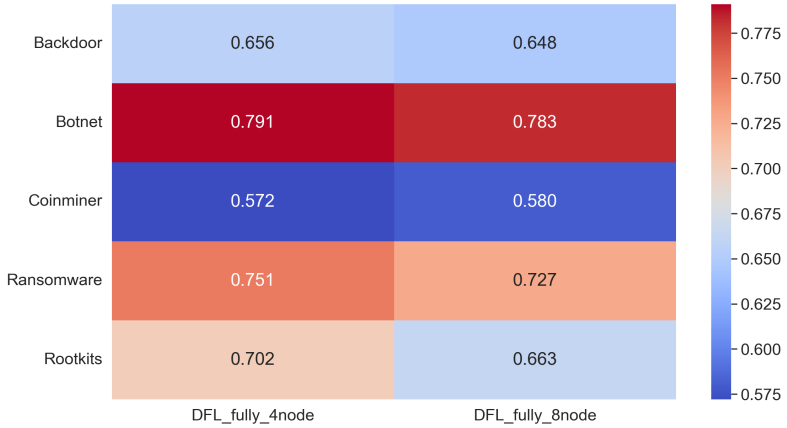


Figure 6.8: Differences in Recall for Different Types of Malware between DFL with 4 Aggregators and 8 Aggregators

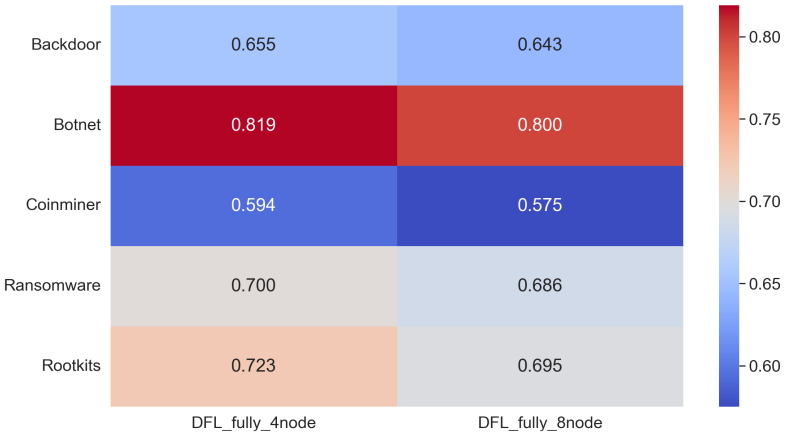
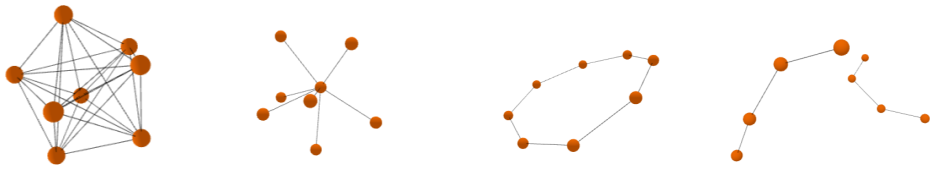


Figure 6.9: Differences in F1-score for Different Types of Malware between DFL with 4 Aggregators and 8 Aggregators

6.2 Comparison between Different DFL Topologies

The second series of experiments aimed at comparing the performance of different DFL topologies: fully-connected, star, ring, and random. Figure 6.10 illustrates these configurations in detail. The fully-connected topology is characterized by each node being interconnected with every other node in the network. In the star topology, there exists a central node to which all other nodes are connected. The ring topology is designed such that each node is linked only to its immediate neighbors on either side. Finally, the random topology is distinguished by its nodes being connected in an arbitrary pattern, without a specific structural design.



(a) Fully (b) Star (c) Ring (d) Random

Figure 6.10: Different DFL Topologies

The various topologies were applied to assess the performance of both the anomaly detection and malware classification models. Detailed results and analyses of these tests are outlined in the following:

(1) Anomaly Detection.

The outcomes for the anomaly detection task are illustrated in Figure 6.11. The results indicated a similarity in performance across the different topologies, with only marginal differences. Notably, the fully-connected structure emerged as the most effective, closely followed by the star topology. The ring and random configurations were slightly less

effective but the difference was small — about 2%. These observations indicate that the way nodes are connected in a DFL network does affect its performance, but not drastically. Generally, having more connections between nodes leads to slightly better performance.

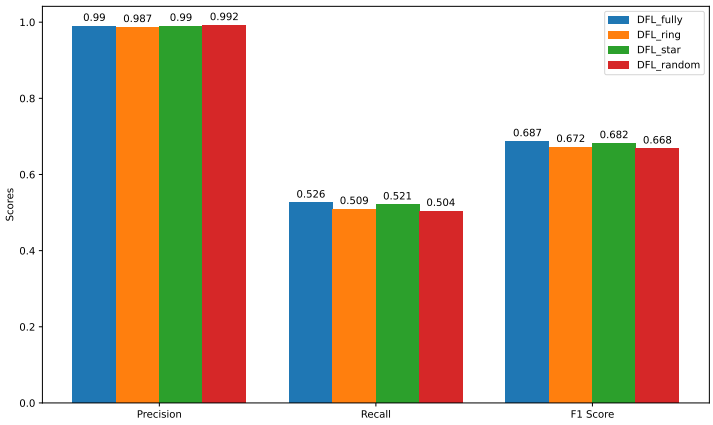


Figure 6.11: Comparison between Different DFL Topologies for Anomaly Detection

(2) Malware Classification.

The results of malware classification model demonstrate that the performance of the four topologies—fully-connected, star, ring, and random—is very close, with the range of precision, recall, and f1-score differences not exceeding 0.02, as shown in Figure 6.12. These observations suggest that the connection mode of nodes within a DFL network does not significantly affect model performance.

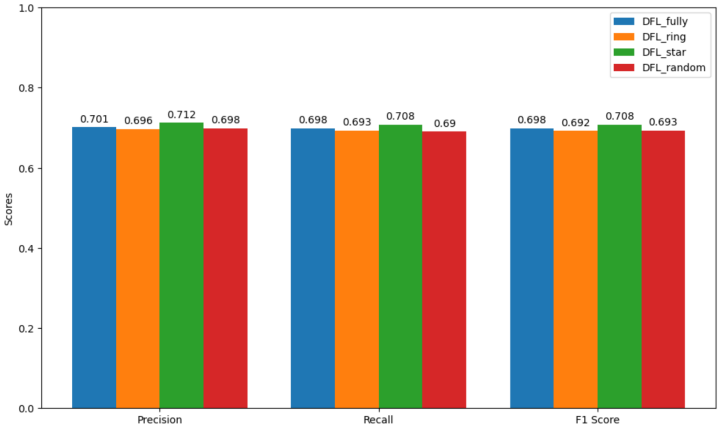


Figure 6.12: Comparison between Different DFL Topologies for Malware Classification

Furthermore, the differences in precision, recall, and f1-score for different types of malware across these four topologies were compared, as shown in Figure 6.13, Figure 6.14, and Figure 6.15. It was found that the variations in performance metrics for each type of malware across the four topologies are minimal. The range of precision for each type of malware is less than 0.04, except for Ransomware with a range of 0.06. Similarly, the range of recall for each malware class is less than 0.04, except for Ransomware with a range of

0.06. Additionally, the range of f1-score for each type of malware is less than 0.03, except for Backdoor with a range of 0.07. These findings indicate that when considering each specific type of malware, the performance across the four different network topologies are also very close.

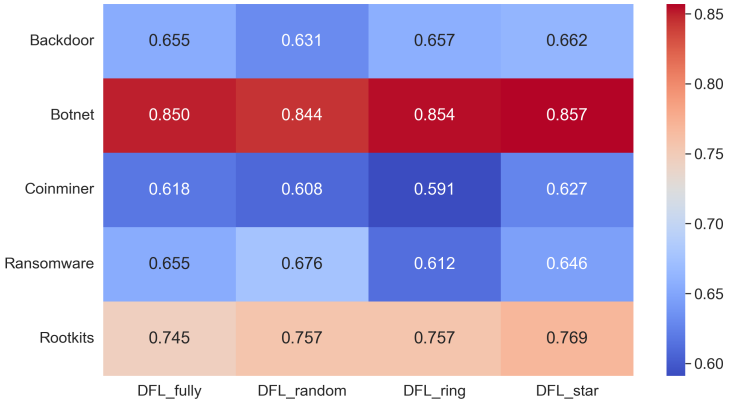


Figure 6.13: Differences in Precision for Different Types of Malware across Different DFL Topologies

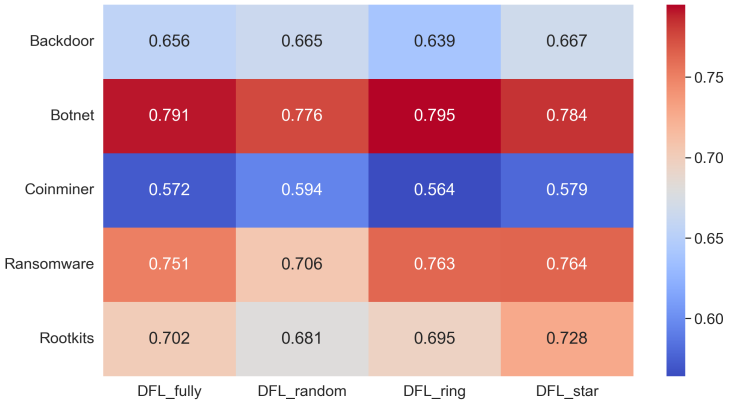


Figure 6.14: Differences in Recall for Different Types of Malware across Different DFL Topologies

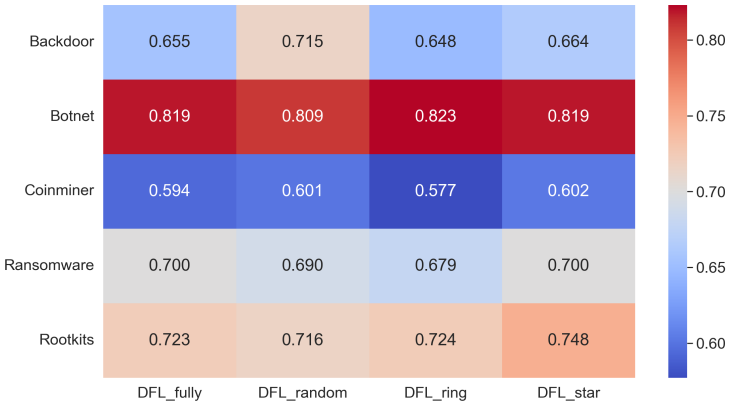


Figure 6.15: Differences in F1-score for Different Types of Malware across Different DFL Topologies

6.3 Non-IID Scenarios

For the Non-IID scenarios, tests in three distinct settings were conducted: 1) each node was trained using data collected by an individual physical device; 2) certain nodes were missing one specific type of malware data; 3) all nodes lacked multiple types of malware data. These scenarios were contrasted against a fully-connected DFL approach, wherein the complete dataset was randomly distributed across all nodes. The following sections provide an in-depth exploration and detailed analysis of each of these scenarios.

6.3.1 Non-IID Scenario 1: Training Each Node with Data from a Specific Physical Device

In this scenario, each node in the network was trained exclusively on data that had been collected by a single, unique physical device. This setup aims to simulate a realistic environment where each device generates its own data.

(1) Anomaly Detection.

The performance of the anomaly detection model under this scenario is depicted in Figure 6.16. A comparative analysis with the IID scenario revealed that, in the Non-IID setting, there was a slight reduction in precision. However, both recall and f1-score experienced an increase, leading to an overall enhancement in performance. This improvement might be attributable to the more uniform data patterns associated with individual devices, which were potentially simpler to learn and adapt to. Conversely, in the IID scenario, where data from various sources were integrated and randomly distributed across nodes, the resultant complex data patterns posed a more challenging learning environment.

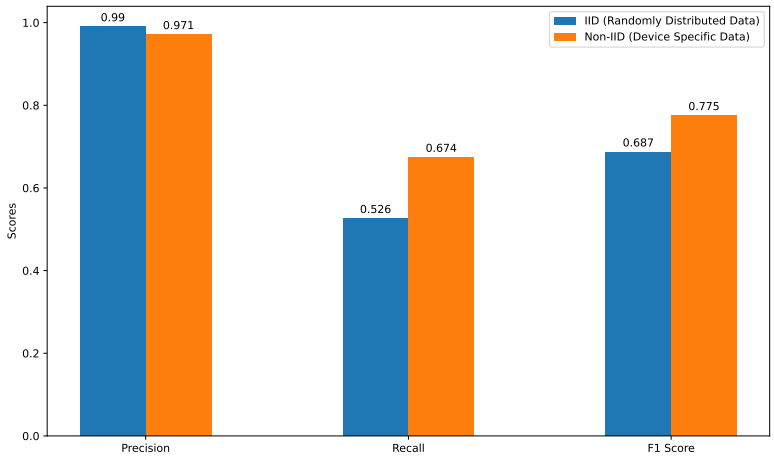


Figure 6.16: Comparison for Anomaly Detection in IID (Randomly Distributed Data) vs. Non-IID (Device Specific Data) Scenarios

(2) Malware Classification.

The performance of the malware classification model, as illustrated in Figure 6.17, shows that precision, recall, and f1-score are all notably higher in the Non-IID scenario compared to the IID scenario, indicating better performance. This is because, in the Non-IID scenario, the data comes from different devices with considerable variation, thus forming more complex data patterns, which makes the learning process of the malware classification model more challenging. As a result, the model's ability to classify malware is stronger in the Non-IID scenario.

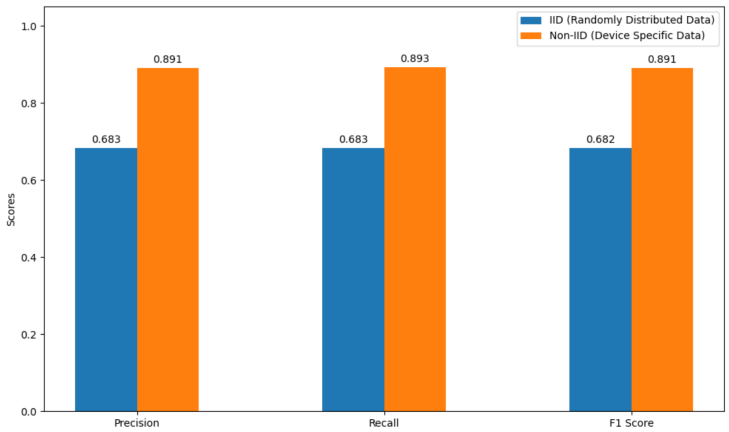


Figure 6.17: Comparison for Malware Classification in IID (Randomly Distributed Data) vs. Non-IID (Device Specific Data) Scenarios

The differences in precision, recall, and f1-score for different types of malware between the IID and Non-IID scenarios were further compared. From Figure 6.18, Figure 6.19 and Figure 6.20, it was observed that for each type of malware, the three metrics, precision, recall, and f1-score are all higher in the Non-IID scenario compared to the IID scenario. This underscores the fact that in the Non-IID scenario, the model's capability to classify malware is stronger than in the IID scenario.

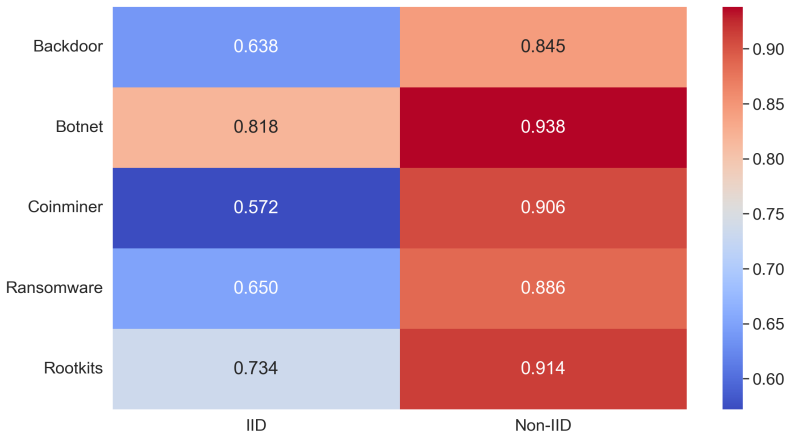


Figure 6.18: Differences in Precision for Different Types of Malware in IID vs. Non-IID Scenarios

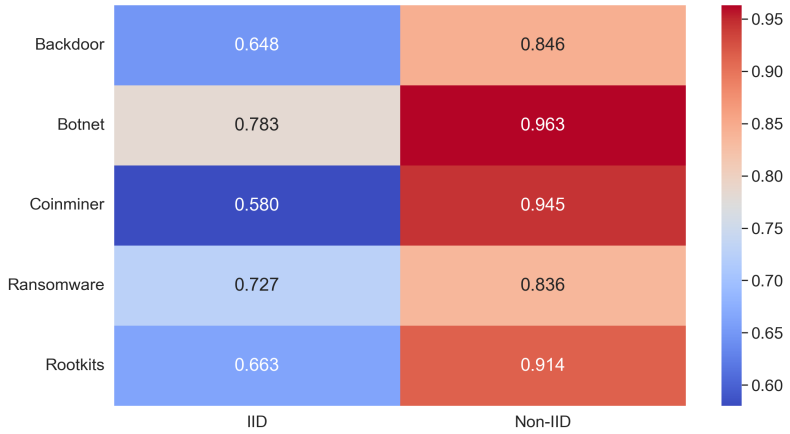


Figure 6.19: Differences in Recall for Different Types of Malware in IID vs. Non-IID Scenarios

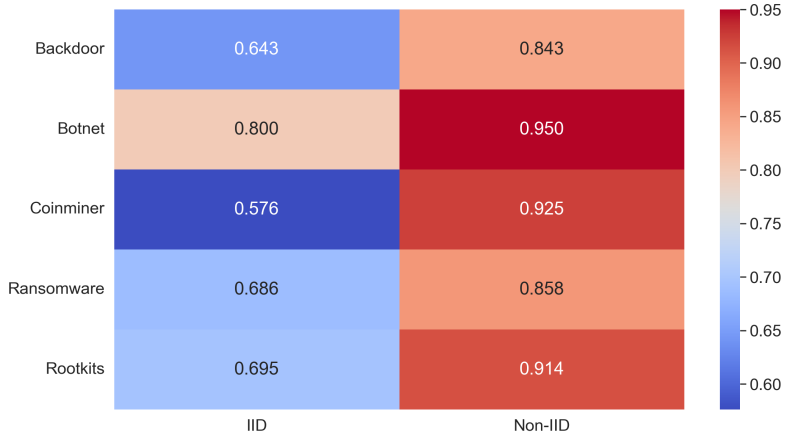


Figure 6.20: Differences in F1-score for Different Types of Malware in IID vs. Non-IID Scenarios

6.3.2 Non-IID Scenario 2: Training Nodes wherein Some of Them Missing a Certain Type of Malware Data

The nodes were trained in this scenario using the malware classification model that is based on the fully connected DFL approach. However, not all nodes have complete datasets, i.e., some nodes miss a certain type of malware data. For instance, two of the nodes lack Botnets data, whereas other nodes have data from all types of malware. The purpose of this setup is to evaluate the robustness of the DFL model to node data incompleteness, evaluate whether the model can continue to work properly when some nodes are missing data, and compensate through cross-node communication where possible.

The following will be analyzed one by one from the perspectives of the five families of malware: Botnets, Backdoors, Rootkits, Coinminer, and Ransomware. In the analysis of each malware category, zero (i.e., fully connected DFL), one, two, and three nodes were missing corresponding malware data respectively.

(1) Botnets.

The results of the malware classification model for this scenario are shown in Figure 6.21. It is generally observed that when more nodes are missing Botnets data, the precision, recall, and f1-score tend to decrease. However, the figure reveals that the precision value is slightly higher in the scenario where two nodes are missing Botnets data (0.820) as compared to the scenario where one node is missing Botnets data (0.814). This anomaly can be attributed to accidental circumstances or better communication compensation between the nodes in the former scenario. Furthermore, the fully connected DFL scenario, which has no missing data, has larger values for all three indicators than the other scenarios.

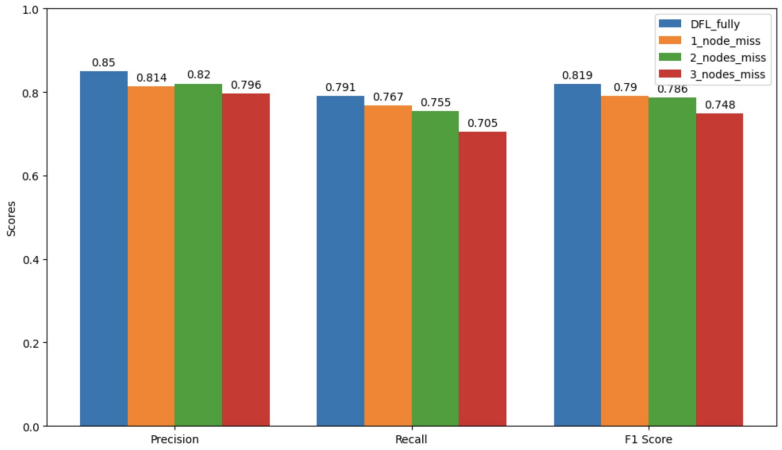


Figure 6.21: Comparison for Malware Classification between Fully Connected DFL and Scenarios with One, Two, and Three Nodes Missing Botnets Data

(2) Backdoors.

Figure 6.22 displays the outcomes of the malware classification model in this situation. Similar to the situation described in (1) about Botnets, it has been commonly noticed that when a greater number of nodes are lacking Backdoors data, the evaluation metrics such as precision, recall, and f1-score tend to exhibit a declining trend. Notably, as the number of nodes missing backdoor data increases, the magnitude of the decline of these indicators increases. This means the model's performance degrades more for the Backdoors category, as missing data nodes cannot contribute enough in training and compensate for this deficiency through communication between nodes.

(3) Rootkits.

The performance of the malware classification model in this scenario are presented in Figure 6.23. By comparing the four data sets, it is evident that the increase in the number of nodes missing Rootkits data harms the model's performance. In other words, with each additional node missing Rootkits data, the precision, recall, and f1-score will decrease further. Notably, the recall rate shows a more noticeable downward trend than the precision rate in this scenario. The decline in recall means that missing nodes have a more significant impact on the model's ability to identify all relevant instances. This is because each node could contain exclusive information that is a positive example for the model. With this information missing, the detection ability of the model will be impaired.

In practical terms, this could lead to reduced security, especially with Rootkits detection, where a lower recall rate may result in more malware being missed.

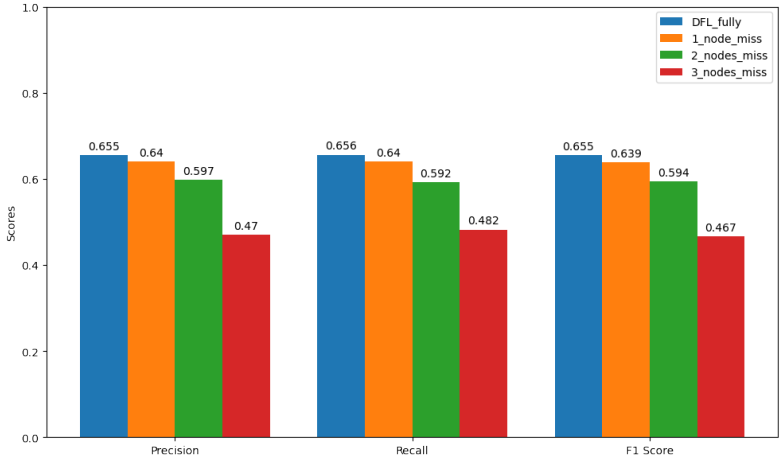


Figure 6.22: Comparison for Malware Classification between Fully Connected DFL and Scenarios with One, Two, and Three Nodes Missing Backdoors Data

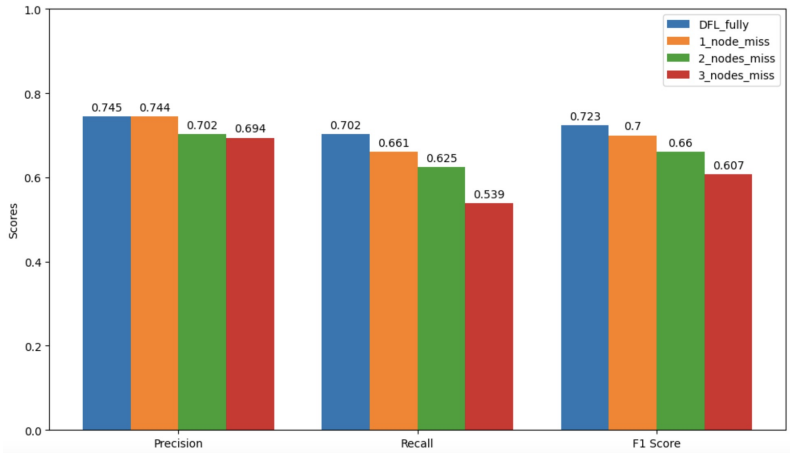


Figure 6.23: Comparison for Malware Classification between Fully Connected DFL and Scenarios with One, Two, and Three Nodes Missing Rootkits Data

(4) Coinminer.

The outcomes of the malware classification model for this scenario are illustrated in Figure 6.24. When evaluating overall performance using f1-score, a notable decrease is observed as more nodes miss Coinminer data. This decline is expected, given the reduced amount of data available for the model to learn from. However, it is noteworthy that the recall scores for the DFL fully connected, one node missing, and two nodes missing Coinminer data are similar, with the differences not exceeding 0.01. This phenomenon can be partially attributed to communication compensation between the nodes and the features and distributions of the missing data.

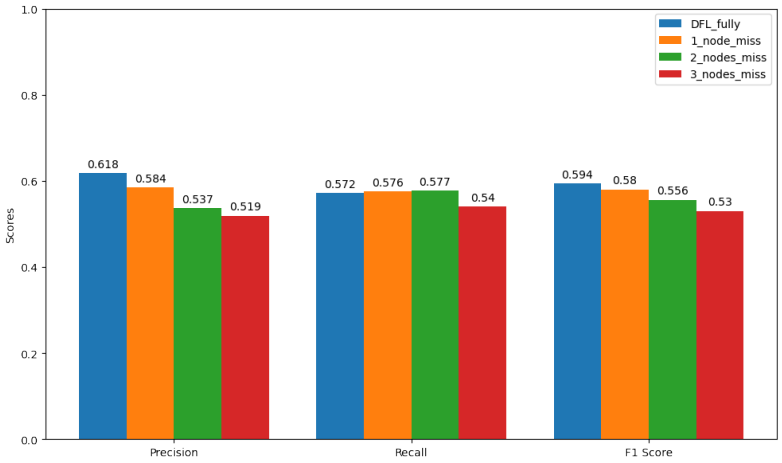


Figure 6.24: Comparison for Malware Classification between Fully Connected DFL and Scenarios with One, Two, and Three Nodes Missing Coinminer Data

(5) Ransomware.

The performance of the malware classification model under this scenario, as depicted in Figure 6.25, follows a similar trend as previous analyses.

When the number of nodes missing Ransomware data increases, the model's performance worsens in general. This degradation is clearly demonstrated by the decreasing precision, recall, and f1-score. Overall, there is no drastic drop between consecutive scenarios and interestingly, the precision and recall values for scenarios with 2 nodes missing and 3 nodes missing Ransomware data are found to be similar. This observation suggests the potential impact of cross-node communication on maintaining the robustness of the DFL model, even in the face of node data incompleteness.

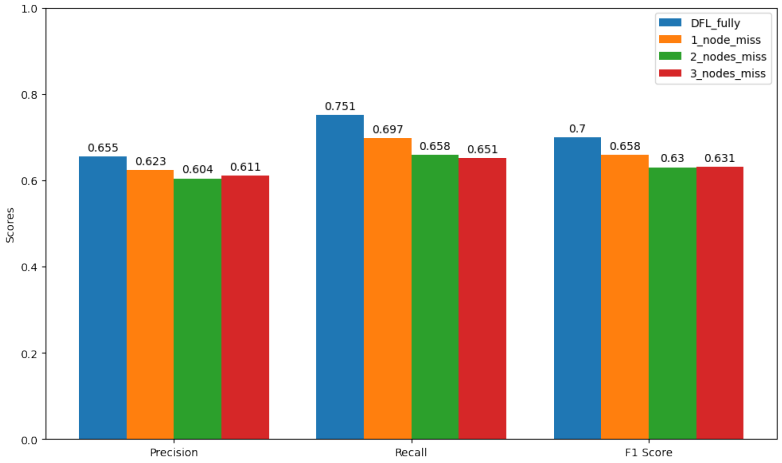


Figure 6.25: Comparison for Malware Classification between Fully Connected DFL and Scenarios with One, Two, and Three Nodes Missing Ransomware Data

6.3.3 Non-IID Scenario 3: Training Nodes wherein All of Them Missing Certain Types of Malware Data

In this scenario, an experiment involving four nodes using the malware classification model based on the fully connected DFL approach was conducted. Each node was intentionally configured to be devoid of one or two malware families. Specifically, node 1 lacked Botnet data, node 2 lacked Backdoor data, node 3 lacked Rootkits data and node 4 lacked Ransomware and Coinminer data. The purpose of this setup is to further evaluate the robustness of the DFL model in the context of node data incompleteness.

The performance of the malware classification model for the Backdoor malware family is depicted in Figure 6.26. The f1-scores are lower for the configuration where all nodes are missing one or two malware families (0.645) compared to the DFL fully connected scenario (0.655), where all nodes process full datasets without missing any type of malware.

Despite the decrease in precision, recall and f1-score, the model's performance did not degrade significantly. This may be attributed to the information compensation facilitated by cross-node communication. Additionally, since each malware family is only missing in one node, the impact on the overall model's ability to classify that specific malware family may not be substantial.

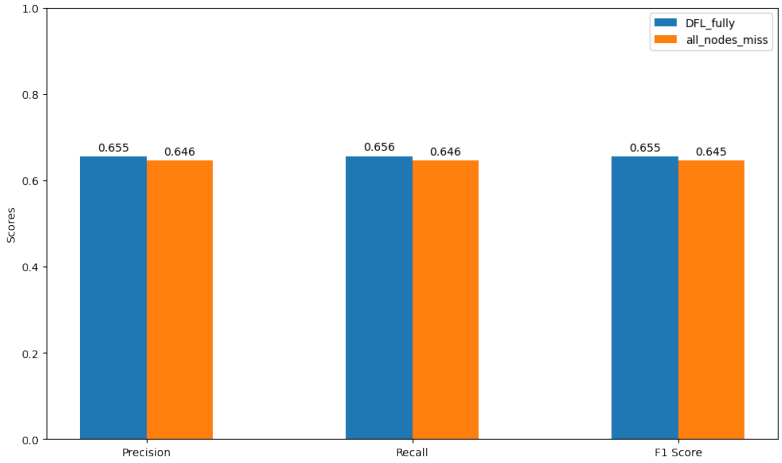
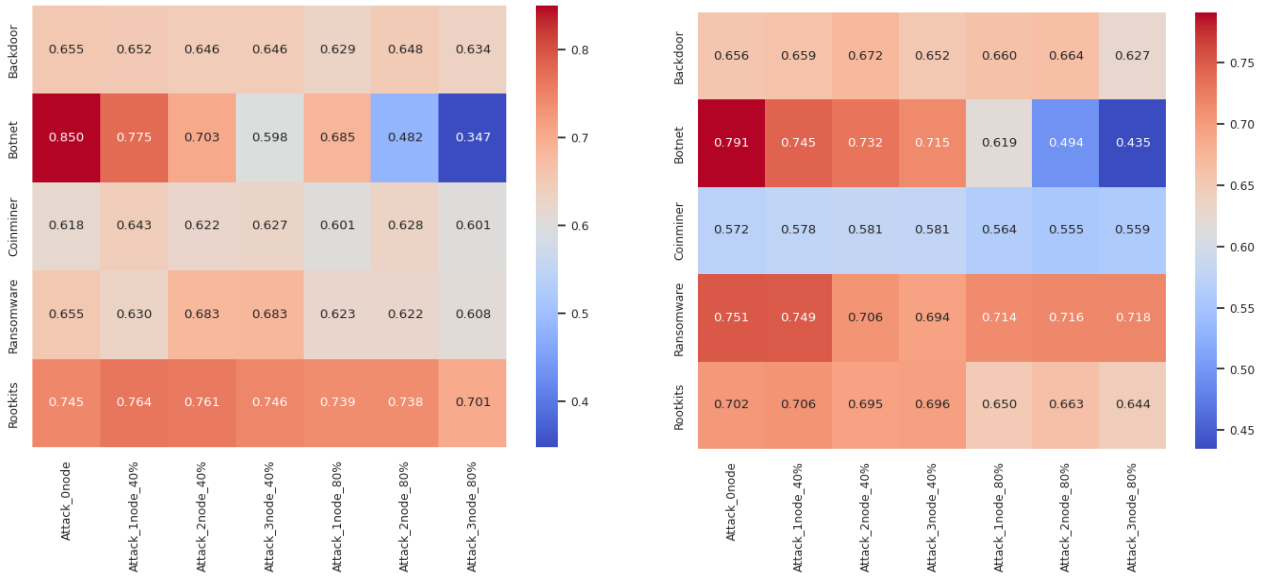


Figure 6.26: Comparison of Backdoor Malware Classification Performance between Fully Connected DFL and Scenarios with All Nodes Missing One or Two Types of Malware Data

6.4 Resilience Against Attacks

To assess the model's resilience against adversarial attacks, supplementary experiments were conducted using the label-flipping feature of the Fedstellar platform. These experiments aimed to simulate attack scenarios and evaluate the model's response under such conditions.

Specifically, botnet data was chosen as a representative to test the malware classification model's performance when faced with different levels of label manipulation. Initially, 40% of the botnet labels were modified on one, two, and three nodes, respectively. The performance of the model was then analyzed in each of these scenarios to understand the impact of this level of label modification. To further test the model's resilience, the challenge was escalated by inverting 80% of the botnet data labels on the same nodes. This step was designed to assess the model's robustness under more extreme conditions. The resulting precision and recall metrics are depicted in Figure 6.27, while the comprehensive f1-score metrics are illustrated in Figure 6.28.



(a) Precision (b) Recall

Figure 6.27: Precision and Recall under Varying Attack Intensities

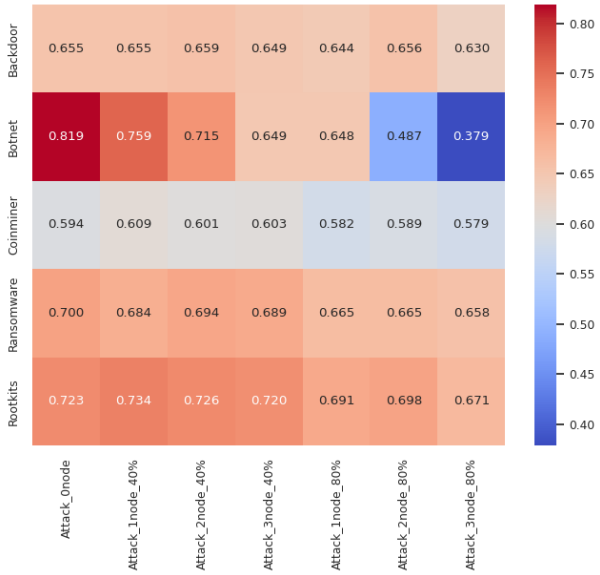


Figure 6.28: F1-score under Varying Attack Intensities

The results indicate the resilience of the model to adversarial attacks. Specifically, when 40% of the botnet labels on a single node were altered, there was an approximate 8% decline in precision for the botnet category, while recall decreased by about 4%. The overall f1-score experienced a reduction of approximately 6%. Given the substantial proportion of label modification, this modest decline underscores the model's relative robustness against attacks. In more extreme scenarios, such as flipping 80% of the data on one node, the overall f1-score maintained a value exceeding 0.6. This resilience can potentially be attributed to the weight-aggregation mechanism inherent in the DFL approach.

Additionally, the observations revealed a trend where increasing the number of compromised nodes led to a more pronounced degradation in performance. A similar effect was seen when a larger proportion of the data underwent label modification. The performances across different types of malware were also impacted, with botnets being the most severely affected. This is primarily because the botnet labels were randomly modified to other labels, leading to more significant disruptions in the model's accuracy for this category.

Chapter 7

Summary, Conclusions and Future Work

This chapter provides a comprehensive summary of the entire project and delves into potential directions for future research. The first section offers a concise overview of the project, while the subsequent section explores prospective paths for further development.

7.1 Summary and Conclusions

The primary goal of this project is the creation of a comprehensive dataset encompassing both normal and abnormal behaviors of IoT devices. It aims to develop models capable of detecting and classifying malware within a DFL framework. To achieve this goal, the project has been structured into several key phases. Initially, this work introduces the necessary background knowledge, encompassing malware types, detection and classification methodologies, and foundational concepts of DFL. Subsequently, an extensive analysis of the state-of-the-art in anomaly detection and malware classification is presented. Drawing from this literature review, a system architecture is conceptualized to outline the structure and functionality of the proposed system.

In the practical phase, a system was implemented to collect device behavior data across six critical dimensions: kernel events, system calls, resource usage, network activity, I/O usage, and file system activities. This data was meticulously integrated, processed, and analyzed to select features for model training and evaluation carefully. Utilizing this dataset, an Autoencoder model was developed for anomaly detection, and an MLP model was constructed for malware classification. The effectiveness of these models was thoroughly evaluated in various scenarios, ensuring the robustness and reliability of the system.

In conclusion, this work presents several key findings:

- Comparative analysis between traditional ML, CFL, and DFL models revealed that ML models slightly outperform CFL and DFL models in terms of effectiveness.
- Examination of different DFL network topologies indicated a minor impact of node connectivity on performance. Denser connections generally lead to slightly improved outcomes.

- In scenarios involving Non-IID data, it was observed that complex data patterns pose a substantial challenge to the learning process. However, the joint learning mechanism of DFL ensures performance is not significantly hampered, even when some training nodes lack certain data types.
- The evaluation under attack scenarios demonstrated that DFL approaches maintain relative robustness against adversarial conditions.

This project makes a significant contribution to IoT security through the creation of a substantial dataset, coupled with novel insights into malware detection and classification within the DFL framework. The results highlight DFL's potential in privacy-preserving scenarios, thereby establishing a foundation for ongoing research and development in this critical area.

7.2 Future Work

This project constructs a large dataset containing device behaviors in both normal and abnormal situations. The abnormal situations cover eight types of malware belonging to five families of malware. To strengthen the effectiveness of the system, future work could incorporate a broader range of malware samples, particularly those that are newly emerging. Additionally, while this work effectively detects and classifies malware, it does not encompass subsequent actions to mitigate these threats. Future improvements could consider integrating the system with defensive solutions to address this gap.

Bibliography

[1] S. Rajendran, R. Calvo-Palomino, M. Fuchs, *et al.*, "Electrosense: Open and big spectrum data", *IEEE Communications Magazine*, vol. 56, no. 1, pp. 210–217, 2017.

[2] L. Li, Y. Fan, M. Tse, and K.-Y. Lin, "A review of applications in federated learning", *Computers & Industrial Engineering*, vol. 149, p. 106 854, 2020.

[3] ElectroSense, *Collaborative spectrum monitoring*, `https://electrosense.org/`, Last accessed on 2024-01-02, 2024.

[4] E. association, *Electrosense - collaborative spectrum monitoring*, Accessed 2024-01-10, 2016. [Online]. Available: `https://electrosense.org/open-api-spec.html`.

[5] A. Huertas, P. M. Sánchez, M. Castillo, G. Bovet, G. Martinez Perez, and B. Stiller, "Intelligent and behavioral-based detection of malware in iot spectrum sensors", *International Journal of Information Security*, vol. 22, Jul. 2022. DOI: `10.1007/s10207-022-00602-w`.

[6] hammerzeit, *An archive of bashlite source code*, Accessed 2024-01-02, 2016. [Online]. Available: `https://github.com/hammerzeit/BASHLITE`.

[7] SkryptKiddie, *Httpbackdoor*, Accessed 2024-01-02, 2020. [Online]. Available: `https://github.com/SkryptKiddie/httpBackdoor`.

[8] J. ao Koritar, *Backdoor*, Accessed 2024-01-02, 2020. [Online]. Available: `https://github.com/jakoritarleite/backdoor`.

[9] Nccgroup, *The tick: A simple embedded linux backdoor.* Accessed 2024-01-02, 2021. [Online]. Available: `https://github.com/nccgroup/thetick/`.

[10] unix thrust, *Beurk experimental unix rootkit.* Accessed 2024-01-02, 2015. [Online]. Available: `https://github.com/unix-thrust/beurk`.

[11] Error996, *Bdvl*, Accessed 2024-01-02, 2020. [Online]. Available: `https://github.com/Error996/bdvl`.

[12] xmrig, *Xmrig*, Accessed 2024-01-02, 2023. [Online]. Available: `https://github.com/xmrig/xmrig`.

[13] jimmy-ly00, *Ransomware poc github repository*, Accessed 2024-01-02, 2020. [Online]. Available: `https://github.com/jimmy-ly00/Ransomware-PoC`.

[14] B. Cakir and E. Dogdu, "Malware classification using deep learning methods", in *Proceedings of the ACMSE 2018 Conference*, ser. ACMSE '18, Richmond, Kentucky: Association for Computing Machinery, 2018, ISBN: 9781450356961. DOI: `10.1145/3190645.3190692`. [Online]. Available: `https://doi.org/10.1145/3190645.3190692`.

[15] P. M. S. Sanchez, J. M. J. Valero, A. H. Celdran, G. Bovet, M. G. Perez, and G. M. Perez, "A survey on device behavior fingerprinting: Data sources, techniques, application scenarios, and datasets", *IEEE Communications Surveys &amp Tutorials*, vol. 23, no. 2, pp. 1048–1077, 2021. DOI: 10.1109/comst.2021.3064259. [Online]. Available: https://doi.org/10.1109%2Fcomst.2021.3064259.

[16] V. Rey, P. M. S. Sánchez, A. H. Celdrán, and G. Bovet, "Federated learning for malware detection in iot devices", *Computer Networks*, vol. 204, p. 108 693, 2022.

[17] S. Flores. "Variational autoencoders are beautiful". (2019), [Online]. Available: https://www.compthree.com/blog/autoencoder/ (visited on 12/20/2023).

[18] P. M. S. Sánchez, A. H. Celdrán, T. Schenk, *et al.*, "Studying the robustness of anti-adversarial federated learning models detecting cyberattacks in iot spectrum sensors", *IEEE Transactions on Dependable and Secure Computing*, 2022.

[19] E. T. M. Beltrán, M. Q. Pérez, P. M. S. Sánchez, *et al.*, "Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges", *IEEE Communications Surveys & Tutorials*, 2023.

[20] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables", in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, IEEE, 2000, pp. 38–49.

[21] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code", in *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, IEEE, vol. 2, 2004, pp. 41–42.

[22] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features", in *2015 10th international conference on malicious and unwanted software (MALWARE)*, IEEE, 2015, pp. 11–20.

[23] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features", *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[24] A. Azmoodeh, A. Dehghantanha, and K.-K. R. Choo, "Robust malware detection for internet of (battlefield) things devices using deep eigenspace learning", *IEEE transactions on sustainable computing*, vol. 4, no. 1, pp. 88–95, 2018.

[25] D. Vasan, M. Alazab, S. Venkatraman, J. Akram, and Z. Qin, "Mthael: Cross-architecture iot malware detection based on neural network advanced ensemble learning", *IEEE Transactions on Computers*, vol. 69, no. 11, pp. 1654–1667, 2020.

[26] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences", in *AI 2016: Advances in Artificial Intelligence: 29th Australasian Joint Conference, Hobart, TAS, Australia, December 5-8, 2016, Proceedings 29*, Springer, 2016, pp. 137–149.

[27] S. Chaba, R. Kumar, R. Pant, and M. Dave, "Malware detection approach for android systems using system call logs", *arXiv preprint arXiv:1709.08805*, 2017.

[28] W. Zhong and F. Gu, "A multi-level deep learning system for malware detection", *Expert Systems with Applications*, vol. 133, pp. 151–162, 2019.

[29] S. I. Popoola, B. Adebisi, M. Hammoudeh, G. Gui, and H. Gacanin, "Hybrid deep learning for botnet attack detection in the internet-of-things networks", *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4944–4956, 2020.

[30] J. Jeon, J. H. Park, and Y.-S. Jeong, "Dynamic analysis for iot malware detection with convolution neural network model", *IEEE Access*, vol. 8, pp. 96 899–96 911, 2020. DOI: 10.1109/ACCESS.2020.2995887.

[31] T. Carrier, P. Victor, A. Tekeoglu, and A. H. Lashkari, "Detecting obfuscated malware using memory feature engineering.", in *Icissp*, 2022, pp. 177–188.

[32] N. Quoc-Dung, "Malware detection in internet of things devices based on association models", in *Proceedings of the 2023 4th International Conference on Computing, Networks and Internet of Things*, ser. CNIOT '23, <conf-loc>, <city>Xiamen</city>, <country>China</country>, </conf-loc>: Association for Computing Machinery, 2023, pp. 743–748, ISBN: 9798400700705. DOI: 10.1145/3603781.3603913. [Online]. Available: https://doi.org/10.1145/3603781.3603913.

[33] J. P. Barona, J. A. Alvarez, C. J. Farfán, J. M. Aguilar, and R. I. Bonilla, "Malware detection using api calls visualisations and convolutional neural networks", in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, IEEE, 2023, pp. 153–159.

[34] N. Quoc-Dung, "Malware detection in internet of things devices based on association models", in *Proceedings of the 2023 4th International Conference on Computing, Networks and Internet of Things*, 2023, pp. 743–748.

[35] E. Ilavarasan and K. Muthumanickam, "A survey on host-based botnet identification", in *2012 International Conference on Radar, Communication and Computing (ICRCC)*, 2012, pp. 166–170. DOI: 10.1109/ICRCC.2012.6450569.

[36] Y. Meidan, M. Bohadana, Y. Mathov, *et al.*, "N-baiot—network-based detection of iot botnet attacks using deep autoencoders", *IEEE Pervasive Computing*, vol. 17, no. 3, pp. 12–22, 2018.

[37] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull, "Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset", *Future Generation Computer Systems*, vol. 100, pp. 779–796, 2019, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.05.041. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18327687.

[38] V. H. Bezerra, V. G. T. da Costa, S. B. Junior, R. S. Miani, and B. B. Zarpelão, "Iotds: A one-class classification approach to detect botnets in internet of things devices", *Sensors (Basel, Switzerland)*, vol. 19, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:198170644.

[39] V. G. T. da Costa, S. Barbon, R. S. Miani, J. J. P. C. Rodrigues, and B. B. Zarpelão, "Detecting mobile botnets through machine learning and system calls analysis", in *2017 IEEE International Conference on Communications (ICC)*, 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7997390.

[40] F. Martinelli, F. Mercaldo, and A. Saracino, "Bridemaid: An hybrid tool for accurate detection of android malware", Apr. 2017, pp. 899–901. DOI: 10.1145/3052973.3055156.

[41] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention", *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2018. DOI: 10.1109/TDSC.2016.2536605.

[42] Y. Zhang and V. Paxson, "Detecting backdoors", Feb. 2001.

[43] R. Canzanese, S. Mancoridis, and M. Kam, "System call-based detection of malicious processes", in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 119–124. DOI: 10.1109/QRS.2015.26.

[44] G. Hoglund and J. Butler, *Rootkits: subverting the Windows kernel.* Addison-Wesley Professional, 2005.

[45] L. Y. Li J, "Rootkits", 2010. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-8378.

[46] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis", in *20th Annual Computer Security Applications Conference*, 2004, pp. 91–100. DOI: 10.1109/CSAC.2004.19.

[47] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants", in *2008 Annual Computer Security Applications Conference (ACSAC)*, 2008, pp. 77–86. DOI: 10.1109/ACSAC.2008.29.

[48] J. Nick L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot—a coprocessor-based kernel runtime integrity monitor", in *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA: USENIX Association, Aug. 2004. [Online]. Available: https://www.usenix.org/conference/13th-usenix-security-symposium/copilot%7B%5Ctextemdash%7D-coprocessor-based-kernel-runtime-integrity.

[49] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants", *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, pp. 670–684, 2011. DOI: 10.1109/TDSC.2010.38.

[50] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking", in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, Chicago, Illinois, USA: Association for Computing Machinery, 2009, pp. 555–565, ISBN: 9781605588940. DOI: 10.1145/1653662.1653729. [Online]. Available: https://doi.org/10.1145/1653662.1653729.

[51] S. H. Kok, A. B. Abdullah, N. Z. Jhanjhi, and M. Supramaniam, "Ransomware , threat and detection techniques : A review", 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:197868067.

[52] S. Barbhuiya, Z. Papazachos, P. Kilpatrick, and D. S. Nikolopoulos, *Rads: Real-time anomaly detection system for cloud data centres*, 2018. arXiv: 1811.04481 [cs.DC].

[53] D. Tanana, "Behavior-based detection of cryptojacking malware", in *2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)*, 2020, pp. 0543–0545. DOI: 10.1109/USBEREIT48449.2020.9117732.

[54] R.-H. Hsu, Y.-C. Wang, C.-I. Fan, *et al.*, "A privacy-preserving federated learning system for android malware detection based on edge computing", in *2020 15th Asia Joint Conference on Information Security (AsiaJCIS)*, IEEE, 2020, pp. 128–136.

[55] T. D. Nguyen, S. Marchal, M. Miettinen, H. Fereidooni, N. Asokan, and A.-R. Sadeghi, "Dïot: A federated self-learning anomaly detection system for iot", in *2019 IEEE 39th International conference on distributed computing systems (ICDCS)*, IEEE, 2019, pp. 756–767.

[56] A. H. Celdrán, P. M. Sánchez Sánchez, E. J. Scheid, *et al.*, "Policy-based and behavioral framework to detect ransomware affecting resource-constrained sensors", in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–7. DOI: 10.1109/NOMS54207.2022.9789790.

[57] D. Shushack, *BA_Thesis_PI*, https://github.com/dennisshushack/BA_Thesis_PI, 2022.

[58] R. Oles and C. Feng, "Detection and classification of malware using file system dimensions for mtd on iot", [Online]. Available: https://files.ifi.uzh.ch/CSG/staff/vonderassen/extern/theses/ma-oles.pdf.

Abbreviations

Acc	Accuracy
AES	Advanced Encryption Standard
ANNs	Artificial Neural Networks
API	Application Programming Interface
APP	Application
BLSTM	Bidirectional Long Short-Term Memory
BoW	Bag-of-Words
CFL	Centralized Federated Learning
CNN	Convolutional Neural Network
CPU	Central Processing Unit
cURL	Client URL
C&C	Command and Control
DDoS	Distributed Denial of Service
DFL	Decentralized Federated Learning
DL	Deep Learning
DLLs	Dynamically Linked Libraries
DT	Decision Tree
ELU	Exponential Linear Unit
FL	Federated Learning
FLSYS	File System
FPR	False Positive Rate
GELU	Gaussian Error Linear Unit
GPUs	Graphics Processing Units
GRU	Gated Recurrent Unit
HPC	High-Performance Computing
HTTP	HyperText Transfer Protocol
IF	Isolation Forest
IID	Independent and Identically Distributed
IoT	Internet of Things
IP	Internet Protocol
I/O	Input/Output
KERN	Kernel Events
KNN	K-Nearest Neighbors
LAE	Long Short-Term Memory Autoencoder
LOF	Local Outlier Factor
LR	Logistic Regression

LSTMs	Long Short-Term Memory
ML	Machine Learning
MLP	MultiLayer Perceptron
NB	Naïve Bayes
NET	Network
Non-IID	Non-Independent and Identically Distributed
OC-SVM	One-Class SVM
OS	Operating System
PC	Personal Computer
PE	Portable Executable
RAM	Random Access Memory
RES	Resource Usage
RF	Radio Frequency
RF	Random Forest
RNNs	Recurrent Neural Networks
RSA	Rivest-Shamir-Adleman
SDR	Software-Defined Radio
SSDF	Subtle System Design Flaws
SVM	Support Vector Machines
SYS	System Call
TCP	Transmission Control Protocol
TNR	True Negative Rate
TPR	True Positive Rate
UDP	User Datagram Protocol
WSGI	Web Server Gateway Interface

List of Figures

List of Tables

Appendix A

Installation Guidelines

This chapter provides a guideline for installing and deploying scripts aimed at monitoring behavior data on Raspberry Pis and developing models for anomaly detection and malware classification.

A.1 Data Monitoring

This section details the steps for the installation and deployment of the monitor controller. You can find the required code, along with installation guides for related malware samples, at GitHub Repository. Before starting the installation process, it's crucial to have a Raspberry Pi ElectroSense sensor properly configured. Additionally, users should establish a dedicated folder on their server or desktop to store monitoring data.

A.1.1 Initial setup

Prior to installing the monitor controller on the Raspberry Pi, SSH needs to be enabled on the server/desktop. The following code presents the installation on a server/desktop machine running Ubuntu:

```
1 sudo apt-get install openssh-server
2 sudo systemctl enable ssh
3 sudo systemctl start ssh
```

After SSH has been enabled, the following commands need to be run on the Raspberry Pi sensor:

```
1 # Update the packages on the sensor:
2 apt-get update
3
4 # Install git on the sensor:
5 apt-get install git
```

```
6
7 # install necessary packages
8 pip3 install scapy
9 pip3 install requests
10 sudo apt-get install sysstat
```

A.1.2 Monitor Controller Installation

An installer script is provided for users to set up the monitor controller. The script will automatically download all dependencies and establish a passwordless SSH connection between the Raspberry Pi sensor and the server. The following commands illustrate the procedure:

```
1 # Clone GitHub repository
2 git clone https://github.com/JingHan0724/MP.git
3
4 # Change directory into the Git repository:
5 cd MP
6
7 # Give access to the installer script:
8 chmod +x install_source.sh
9
10 # Run installer script
11 ./install_source.sh -s username@desktopipaddress
```

A.1.3 Monitoring Scripts

In the "./monitors" directory, a suite of monitoring scripts is available, each with a specific function:

- *KERN.sh*: Monitors HPC and ressource usage (provided by Dr.Huertas and Dr.Feng; 5 seconds time window)
- *RES*: Monitors HPC and ressource usage (provided by Dr.Huertas and Dr.Feng; 5 seconds time window)
- *SYS.sh*: Monitors systemcalls (provided by Dr.Huertas and Dr.Feng; 10 seconds time window)
- *network_monitor.py*: Monitors events coming from the network (5 seconds time window)
- *block_monitor.sh*: Monitors events coming from the input/output (10 seconds time window)
- *calculate_entropy.sh*: Calculate the entropy from the input/output (10 seconds time window)

- *file_system_monitor.sh*: Monitors events coming from the file system (5 seconds time window)

Each of these scripts can be executed independently or can be invoked through the monitoring controller. For details on how to control these scripts via the monitoring controller, refer to Section A.1.4.

A.1.4 Data Collecting

To initiate the monitoring scripts and ensure the transmission of collected data to the personal computer, specific scripts need to be executed on both the personal computer and the Raspberry Pi.

(1) Server-side (Personal Computer):

The scripts responsible for listening and data transmission are located in the "./server" directory. Modify these scripts to specify your preferred IP address and choose a data directory according to your requirements. Once configured, run the data transmission scripts to start listening for incoming data.

(2) Client-side (Raspberry Pi):

Adjust the monitoring scripts to utilize the server IP address and ports. Then execute the control script:

```
# Change Directory to Monitor Controller:}
cd controller

# Enable Virtual Environment:
source env/bin/activate

# Start collecting:
python3 collect.py
```

Upon initiating the data collection process, the user will be prompted to input several parameters, each crucial for customizing the monitoring session:

- **Time:** Specify the duration for monitoring, in seconds. For example, 60 for one minute.
- **Monitors:** Choose which monitoring scripts to invoke, running them concurrently. If selecting multiple monitors, separate them with commas. For instance, RES, KERN, SYS.
- **Server Path:** Indicate the storage path on the server or personal computer for the collected data. An example format is: username@serverip:/home/username/Desktop/data.

By carefully setting these parameters, users gain the flexibility to tailor the monitoring process to their specific needs, allowing for data collection from chosen dimensions over the desired time frame.

A.2 Model Development

This section is dedicated to the construction of models for anomaly detection and malware classification. The process begins with the crucial step of processing the collected monitoring data, followed by model building and testing on the Fedstellar platform.

A.2.1 Data Processing

After completing the previous operations, data tables from the NET, BLOCK, ENTROPY, FLSYS, RES, SYS, and KERN modules were collected.

In this phase, the preprocessing_data.ipynb notebook will be executed firstly to extract features and merge the data tables of the six modules based on timestamps. Subsequently, the merge_data.ipynb notebook will be run to combine the data collected from all eight devices. Afterwards, feature_selection_autoencoder.ipynb and feature_selection_mlp.ipynb will be separately executed to select features specifically for the Autoencoder model and the MLP model. A more detailed introduction of code function is as follows:

- *preprocessing_data.ipynb*: This script extracts features from the NET, ENTROPY, and SYS modules, then calculates the start time and end time for the tables of all six modules. The time span from start time to end time is divided into numerous continuous intervals of 20 seconds each, and then the script calculates the row average in each intervals for six tables. Finally, the script merges these six processed tables into a single table comprising over 400 columns.

- *merge_data.ipynb*: This script integrates the data from eight IoT devices into one large table, and then label normal data and eight types of malware samples accordingly. Since eight devices have some difference in SYS features, a final table with over 500 columns is obtained.

- *feature_selection_autoencoder.ipynb*: This script is instrumental in generating a refined dataset with 22 features specifically for the Autoencoder model.

- *draw_scatterplot.py*: This script is used to generate scatter plots for all features, each plot containing 9 columns representing data under eight types of malware attacks and the normal data. Then 96 features corrsponding to 96 plots were filtered out where there is a significant difference in the distribution of the 9 columns of dots as listed in the step 1 of Table A.1.

- *feature_selection_mlp.ipynb*: This script is instrumental in generating a refined dataset with 30 features specifically for the MLP model as listed in the step 2 of Table A.1.

Step	Selected Features
Step 1	access_mean, armv7_cortex_a7/br_immed_retired/_mean, armv7_cortex_a7/exc_return/_mean, armv7_cortex_a7/inst_retired/_mean, armv7_cortex_a7/l1i_cache/_mean, armv7_cortex_a7/l1i_cache_refill/_mean, armv7_cortex_a7/pc_write_retired/_mean, armv7_cortex_a7/st_retired/_mean, await_mean, block:block_bio_backmerge_mean, block:block_getrq_mean, block:block_plug_mean, block:block_rq_complete_mean, brk_mean, clone_mean, connect_mean, cpu_mean, DifferentDestPorts, DifferentSourcePorts, dup2_mean, epoll_wait_mean, execve_mean, exit_group_mean, ext4:ext4_alloc_da_blocks_mean, ext4:ext4_begin_ordered_truncate_mean, ext4:ext4_drop_inode_mean, ext4:ext4_es_find_extent_range_enter_mean, ext4:ext4_es_find_extent_range_exit_mean, ext4:ext4_es_insert_extent_mean, ext4:ext4_es_lookup_extent_enter_mean, ext4:ext4_writepages_result_mean, faccessat_mean, fchmod_mean, fcntl64_mean, filemap:mm_filemap_add_to_page_cache_mean, flock_mean, fstat64_mean, fstatat64_mean, getdents64_mean, getegid32_mean, getrandom_mean, getsockname_mean, gettimeofday_mean, getxattr_mean, gpio:gpio_value_mean, ioctl_mean, iowrite_mean, iowritebytes_mean, kcmp_mean, keyctl_mean, L1-dcache-loads_mean, L1-dcache-stores_mean, L1-icache-load-misses_mean, L1-icache-loads_mean, llseek_mean, lstat64_mean, MaxLength, MedianLength, memory_mean, mkdir_mean, mmap2_mean, mmc:mmc_request_done_mean, mprotect_mean, mremap_mean, munmap_mean, openat_mean, pagemap:mm_lru_insertion_mean, perf_event_open_mean, pipe_mean, poll_mean, prctl_mean, r_await_mean, raw_syscalls:sys_enter_mean, raw_syscalls:sys_exit_mean, read_mean, read_ops_mean, recvfrom_mean, rpm:rpm_suspend_mean, rt_sigaction_mean, set_robust_list_mean, set_tid_address_mean, setitimer_mean, sigreturn_mean, socket_mean, statfs_mean, statfs64_mean, ugetrlimit_mean, uname_mean, unlink_mean, w_await_mean, wait4_mean, waitid_mean, workqueue:workqueue_execute_start_mean, write_kbs_mean, writeback:sb_clear_inode_writeback_mean, writeback:writeback_dirty_inode_enqueue_mean
Step2	armv7_cortex_a7/br_immed_retired/_mean, armv7_cortex_a7/exc_return/_mean, armv7_cortex_a7/inst_retired/_mean, armv7_cortex_a7/l1i_cache/_mean, armv7_cortex_a7/pc_write_retired/_mean, armv7_cortex_a7/st_retired/_mean, brk_mean, connect_mean, epoll_wait_mean, execve_mean, fstat64_mean, getrandom_mean, kcmp_mean, L1-dcache-loads_mean, L1-dcache-stores_mean, L1-icache-loads_mean, llseek_mean, memory_mean, mmap2_mean, mprotect_mean, munmap_mean, openat_mean, pagemap:mm_lru_insertion_mean, raw_syscalls:sys_enter_mean, raw_syscalls:sys_exit_mean, set_robust_list_mean, set_tid_address_mean, socket_mean, statfs64_mean, uname_mean

Table A.1: Selected Features for MLP in Two Steps

A.2.2 Model Training and Evaluation

In the initial stage of model development, traditional ML techniques are utilized, with scripts written in PyTorch. The process involves several key steps:

- **Data Preparation:** The malware.py script is designed for the dataset preparation specific to the MLP model. Its primary functions include dividing the data into training, validation, and testing subsets and converting the data into the TensorDatasets format. In parallel, the malware2.py script performs a similar role for the Autoencoder model, ensuring the dataset is appropriately loaded and prepared.
- **Model Training and Evaluation:** The training and evaluation of the MLP and Autoencoder models are executed via the mlp.py and autoencoder.py scripts, respectively. These scripts are integral to both the training of the models and the subsequent evaluation of their performance.

Following the development of these scripts, they are integrated into the Fedstellar framework. This integration allows users to train and evaluate their models in a FL environment, leveraging the distributed and collaborative nature of Fedstellar to enhance the ML processes.

Appendix B

Contents of the CD

The CD contains all the documents, project source code, and installation instructions for this project.