MASTER'S DEGREE IN TELECOMMUNICATION ENGINEERING

MASTER'S THESIS

LSTM-BASED ATTACK CLASSIFICATION IN IOT NETWORKS

9 INDUSTRIA, INNOVACIÓN E INFRAESTRUCTURA

11 CIUDADES Y COMUNIDADES SOSTENIBLES

Student: Alcorta Gascon, Iker

Instructor: Dr. Saniie, Jafar

Co-instructor: Gromov, Mikhail

Course: 2023-24

Date: 09/09/2024, Sopela

Contents

ABSTRACT

The proliferation of Internet of Things (IoT) devices has heightened the need for robust security measures to protect against a growing number of cyberattacks. In this project, a Long Short-Term Memory (LSTM) neural network approach is presented for classifying attacks in IoT networks, leveraging the comprehensive IoT-23 dataset. The objective of the research is to develop an effective Artificial Intelligence (AI) model that accurately identifies various types of network intrusions and anomalies indicating botnet infiltration, thereby enhancing the security posture of IoT environments.

The IoT-23 dataset is preprocessed to extract relevant features, and the LSTM model is trained using these inputs. Superior performance is demonstrated by this approach, achieving an accuracy of 98.8% in classifying attacks, thereby improving upon traditional machine learning models. These results underscore the potential of LSTM networks in IoT security applications, offering a scalable and adaptive solution to detect emerging threats.

Future work will explore the integration of this model into real-time security systems and its applicability to diverse IoT architectures.

Keywords—Internet of Things (IoT), Artificial Intelligence, Botnets, Long Short-Term Memory Neural Network, IoT-23 dataset

LABURPENA

Gauzen Interneteko (IoT) gailuen ugaritzeak segurtasun-neurri sendoen beharra areagotu du zibererasoen kopuru gero eta handiagotik babesteko. Proiektu honetan, IoT sareetako erasoak sailkatzeko Epe Luzeko eta Laburreko Memoriaren sare neuronal batean (LSTM, ingelesezko sigletan) oinarritutako ikuspegia aurkezten da, IoT-23 datu integralen multzoa aprobetxatuz. Ikerketaren helburua da Adimen Artifizialaren (IA) eredu eraginkor bat garatzea, botneten infiltrazioa adierazten duten sareko hainbat intrusio eta anomalia zehaztasunez identifikatzeko, eta, horrela, IoT inguruneetako segurtasun-jarrera hobetzeko.

IoT-23 datu-multzoa aurrez prozesatzen da ezaugarri garrantzitsuak ateratzeko, eta LSTM eredua sarrera horiek erabiliz entrenatzen da. Ikuspegi horrek errendimendu handiagoa erakutsi du, eta erasoen sailkapenean % 98,8ko zehaztasuna lortu du; horrek esan nahi du hobetu egin dela ikasketa automatikoko eredu tradizionalekin alderatuta. Emaitza horiek LSTM sareek IoTko segurtasun-aplikazioetan duten ahalmena azpimarratzen dute, sortzen ari diren mehatxuak detektatzeko irtenbide eskalagarria eta moldagarria eskainiz.

Etorkizuneko lanak eredu hori denbora errealeko segurtasun-sistemetan integratzea eta hainbat IoT arkitekturatan aplikatzea aztertuko du.

Gako-hitzak — Gauzen Internet (IoT), Adimen Artifiziala, Botnetak, Epe Luzeko eta Laburreko Memoriaren Neurona-sarea, IoT-23 datu-multzoa.

RESUMEN

La proliferación de dispositivos del Internet de las Cosas (IoT) ha incrementado la necesidad de medidas de seguridad robustas para protegerse contra un número creciente de ciberataques. En este proyecto, se presenta un enfoque basado en una red neuronal de Memoria a Largo y Corto Plazo (LSTM, por sus siglas en inglés) para clasificar ataques en redes IoT, aprovechando el conjunto de datos integral IoT-23. El objetivo de la investigación es desarrollar un modelo de Inteligencia Artificial (IA) efectivo que identifique con precisión varios tipos de intrusiones y anomalías en la red que indiquen infiltración de botnets, mejorando así la postura de seguridad en los entornos IoT.

El conjunto de datos IoT-23 se preprocesa para extraer características relevantes, y el modelo LSTM se entrena utilizando estas entradas. Este enfoque ha demostrado un rendimiento superior, logrando una precisión del 98.8% en la clasificación de ataques, lo que supone una mejora con respecto a los modelos tradicionales de aprendizaje automático. Estos resultados subrayan el potencial de las redes LSTM en aplicaciones de seguridad en IoT, ofreciendo una solución escalable y adaptable para detectar amenazas emergentes.

El trabajo futuro explorará la integración de este modelo en sistemas de seguridad en tiempo real y su aplicabilidad a arquitecturas IoT diversas.

Palabras clave—Internet de las Cosas (IoT), Inteligencia Artificial, Botnets, Red Neuronal de Memoria a Largo y Corto Plazo, conjunto de datos IoT-23.

1 Introduction

The Internet of Things (IoT) is transforming the technological landscape by enabling a vast network of connected devices to communicate and exchange data seamlessly. From smart homes and cities to healthcare, industrial automation, and transportation, IoT technologies are revolutionizing various sectors by providing unprecedented convenience and efficiency. According to projections, by 2025, the number of IoT devices will exceed 75 billion, a significant increase from the 27 billion devices reported in 2017. While the rapid expansion of IoT presents numerous opportunities, it also introduces substantial security challenges. IoT devices, often characterized by limited computational resources and diverse architectures, are particularly vulnerable to cyberattacks, making IoT networks an attractive target for malicious actors.

Cybercriminals have increasingly focused on exploiting these vulnerabilities, with botnets, such as the notorious Mirai botnet, serving as a prominent example of how compromised IoT devices can be utilized to launch large-scale cyberattacks. Distributed Denial of Service (DDoS) attacks, data breaches, and other malicious activities have become more frequent, with DDoS attacks sometimes exceeding 1 Tbps in intensity, severely disrupting online services and infrastructures [2]. These security concerns underscore the urgent need for robust and scalable solutions to protect IoT networks from potential threats.

To address the unique security challenges associated with IoT networks, edge computing has emerged as a promising solution. By processing data closer to the source, edge computing reduces latency and bandwidth usage, enabling faster and more efficient responses to cyber threats. In contrast to traditional cloud-based computing, which may introduce delays due to centralized processing, edge computing distributes computational tasks across multiple nodes closer to the devices themselves. This decentralized approach enhances the ability to detect and mitigate threats in real time, making it a vital component of modern security frameworks for IoT networks [3]. Additionally, the decentralized nature of edge computing minimizes the risk of a single point of failure, increasing the overall resilience of IoT infrastructures.

Intrusion Detection Systems (IDS) are essential tools in maintaining network security by monitoring and analyzing network traffic to detect potential threats. However, traditional IDS approaches often struggle to adapt to the diverse and resource-constrained nature of IoT devices. The heterogeneous environment, coupled with the large volume of data generated by IoT networks, presents challenges for conventional IDS architectures. To overcome these limitations, machine learning (ML) techniques, particularly deep learning models, have gained attention as effective tools for enhancing IDS performance in IoT environments.

Among deep learning models, Long Short-Term Memory (LSTM) networks have shown significant promise due to their ability to model temporal dependencies in sequential data. LSTM networks are especially suited for tasks such as time-series analysis, where the identification of patterns over time is critical for detecting abnormal behavior. In the context of IoT networks, LSTMs can analyze patterns in network traffic over time and detect deviations that indicate the presence of malicious activity. This capability makes LSTM-based IDS solutions highly effective for attack detection and classification in dynamic and distributed IoT systems.

This report presents a novel approach to attack classification in IoT networks using an LSTM neural network, leveraging the IoT-23 dataset, which consists of labeled IoT network traffic data [5]. The research aims to develop a robust and scalable model capable of accurately detecting and classifying a wide range of cyberattacks, including DDoS attacks, brute-force attempts, and data exfiltration. By integrating edge computing principles with advanced deep learning techniques, the proposed approach seeks to address the security challenges specific to IoT networks. The model is designed to be adaptive, scalable, and capable of real-time threat detection, thereby enhancing the security infrastructure of IoT environments.

The research also highlights the importance of a comprehensive security strategy for IoT systems, one that combines robust intrusion detection mechanisms with the advantages of edge computing. By leveraging a distributed, low-latency framework, the proposed solution aims to enable timely detection and mitigation of attacks, even in resource-constrained IoT networks. Through experimental evaluation using the IoT-23 dataset, the study demonstrates the effectiveness of the LSTM-based IDS in classifying various types of cyberattacks with high accuracy and minimal false-positive rates. Ultimately, this work contributes to the development of advanced AI-driven security frameworks capable of protecting IoT networks from emerging threats.

The subsequent sections of this report provide a detailed examination of the context, objectives, benefits of the project, methodology with the explanation IoT-23 dataset, discuss the architecture and training process of the LSTM model, and present the results of experimental evaluations. Furthermore, the implications of edge computing on IDS performance are explored, alongside potential directions for future research aimed at improving IoT network security.

2 Context

2.1 Intrusion Detection Systems

Intrusion Detection Systems (IDS) are a critical component of modern network security, designed to monitor and analyze network traffic for suspicious activities that may indicate security breaches or unauthorized access. The main function of an IDS is to detect and respond to threats in real-time or near real-time, offering protection against cyberattacks. Various types of IDS exist, each suited to different environments and threat models. The key types of IDS are broadly categorized based on the method of detection and the location of deployment. The primary classifications include Network-based IDS (NIDS), Host-based IDS (HIDS), Signature-based IDS, Anomaly-based IDS, and Hybrid IDS.

2.1.1 Network-based IDS (NIDS)

Network-based Intrusion Detection Systems are deployed at strategic points within a network to monitor inbound and outbound traffic. NIDS examine packets traveling across the network and analyze them for suspicious patterns or behaviors that indicate possible attacks. These systems operate by listening to network traffic and comparing it to known attack patterns or anomalies.

- **Advantages**: NIDS are effective at detecting network-wide attacks such as Distributed Denial of Service (DDoS) attacks, scanning attacks, or Man-in-the-Middle (MITM) attacks. They offer a broad view of the network and can monitor multiple devices simultaneously.

- **Challenges**: Since NIDS focus on network traffic, they may miss threats that originate from within a host or encrypted traffic. They are also less effective against encrypted attacks unless deployed with additional decryption capabilities.

Examples of NIDS: Snort, Suricata.

2.1.2 Host-based IDS (HIDS)

Host-based Intrusion Detection Systems are installed on individual devices (hosts), such as servers, workstations, or IoT devices, to monitor and analyze the activities and behavior of the host system. HIDS focus on examining system logs, file integrity, system calls, and network interface statistics to detect suspicious activity at the device level.

- **Advantages**: HIDS provide granular visibility into the behavior of individual devices, making them useful for detecting insider threats or malware that may not generate suspicious network traffic. They are also well-suited for detecting file-based attacks, privilege escalation attempts, and unauthorized access to system files.

- **Challenges**: HIDS can be resource-intensive, requiring processing power and storage on each monitored device. Moreover, they provide limited insight into network-wide activities and may need to be deployed alongside NIDS for comprehensive coverage.

Examples of HIDS: OSSEC, Tripwire.

2.1.3 Signature-based IDS

Signature-based IDS are one of the most traditional and widely used detection methods. These systems rely on a database of predefined signatures or known patterns of malicious behavior. Each signature corresponds to a specific type of attack, such as a known virus, worm, or exploit. Signature-based systems compare incoming traffic or system activities against this database to identify matches with known threats.

- **Advantages**: Signature-based IDS are highly accurate at detecting known attacks with minimal false positives. They are straightforward to implement and maintain in environments where known attack patterns can be cataloged and updated regularly.

- **Challenges**: These systems are ineffective against zero-day attacks or new, unknown threats, as they can only detect attacks for which signatures have already been developed. As a result, signature-based IDS must be continuously updated to remain effective.

Examples of signature-based IDS: Snort, Cisco Secure IPS.

2.1.4 Anomaly-based IDS

Anomaly-based IDS rely on establishing a baseline of normal network or host behavior, against which deviations can be detected. These systems use statistical models, machine learning algorithms, or artificial intelligence to learn normal patterns of activity, such as network traffic volume, types of protocols used, or the frequency of specific requests. When the IDS detects activity that deviates significantly from this baseline, it flags the behavior as potentially malicious.

- **Advantages**: Anomaly-based IDS can detect new and previously unknown threats, including zero-day attacks, as they are not restricted to predefined signatures. They are highly adaptive and can be effective in environments where threats evolve rapidly or where custom-built malware is deployed.

- **Challenges**: A major challenge for anomaly-based IDS is the high rate of false positives, especially in dynamic or changing environments where the baseline of "normal" behavior may shift. Additionally, establishing an accurate and meaningful baseline can be difficult in complex networks.

Examples of anomaly-based IDS: Bro (Zeek), Darktrace.

2.1.5 Hybrid IDS

Hybrid IDS combine both signature-based and anomaly-based detection methods to provide comprehensive protection. By leveraging the strengths of both approaches, hybrid IDS can detect known threats efficiently through signature matching while also identifying previously unknown or emerging threats via anomaly detection. These systems often incorporate machine learning algorithms to improve detection accuracy and reduce false positives.

- **Advantages**: Hybrid IDS provide a balanced solution, offering the high accuracy of signature-based detection for known attacks and the adaptability of anomaly-based detection for new threats. This approach allows organizations to benefit from both real-time threat identification and proactive monitoring of suspicious behaviors.
- **Challenges**: Hybrid IDS can be more complex to implement and maintain compared to single-method systems. Additionally, their reliance on anomaly detection still makes them prone to false positives in environments where legitimate activities may trigger alerts.

Examples of hybrid IDS: AlienVault USM, IBM QRadar.

The landscape of IDS technologies is diverse, with each type offering distinct advantages and limitations. Network-based IDS are ideal for monitoring large-scale network traffic, while host-based IDS provide detailed insights at the device level. Signature-based systems are effective for detecting known attacks with precision, whereas anomaly-based systems offer the flexibility to identify novel threats. Hybrid systems, incorporating both signature and anomaly detection techniques, offer comprehensive coverage. Selecting the appropriate IDS for an organization depends on the specific security needs, network architecture, and threat model, particularly in dynamic environments like IoT networks, where scalable and adaptive intrusion detection mechanisms are increasingly necessary.

In this project, a Network-based Intrusion Detection System (NIDS) is implemented to enhance the security of IoT networks. NIDS is deployed at strategic points within the network to monitor and analyze network traffic in real-time, enabling the detection of malicious activities such as Distributed Denial of Service (DDoS) attacks, unauthorized access, and data exfiltration attempts. Given the highly distributed and resource-constrained nature of IoT devices, a network-based approach provides a comprehensive view of network behavior without overloading individual devices with security responsibilities.

The NIDS in this project leverages deep learning techniques, specifically a Long Short-Term Memory (LSTM) neural network, to analyze network traffic patterns. By examining traffic for suspicious activities and comparing it to known attack behaviors or deviations from normal patterns, the system is designed to detect a wide range of cyberattacks targeting IoT networks. This approach ensures that the system can identify both known threats, using predefined attack signatures, and emerging threats, through anomaly detection.

By focusing on network-level monitoring, the NIDS solution implemented in this project provides a scalable and adaptive defense mechanism for IoT environments. This is especially critical as IoT networks are often susceptible to attacks that exploit the lack of centralized security controls. The NIDS allows for real-time detection and mitigation of threats before they can propagate through the network or cause significant damage, thus improving the overall resilience of the IoT infrastructure.

2.2 IoT Security Risks: Bonets

As the Internet of Things (IoT) continues to grow rapidly, with projections estimating over 75 billion devices by 2025 [1], the security of these networks has become a critical concern. IoT devices, often designed with limited computational and security resources, are highly vulnerable to cyberattacks. One of the most prominent and persistent threats to IoT networks is the exploitation of devices through **botnets**—large networks of compromised devices controlled remotely by attackers. These botnets can be used to launch a wide range of malicious activities, from Distributed Denial of Service (DDoS) attacks to data theft and malware propagation.

2.2.1 Botnets

A **botnet** is a collection of devices infected with malware that allows attackers to control them remotely without the device owner's knowledge. The attacker, referred to as the **botmaster**, uses these compromised devices (or "bots") to carry out coordinated attacks, usually for financial gain or to disrupt services. IoT devices, such as smart cameras, thermostats, and routers, are particularly attractive targets for botmasters due to their weak security configurations, default credentials, and lack of regular software updates.

Botnets pose significant risks to both the owners of the compromised devices and the broader internet infrastructure. IoT botnets can generate massive traffic flows, overwhelming servers and networks during DDoS attacks. A well-known example of this is the **Mirai botnet**, which, in 2016, hijacked hundreds of thousands of IoT devices and was used to launch one of the largest DDoS attacks ever recorded, bringing down major websites like Twitter, Netflix, and Reddit [2].

There are several types of botnets, each with different architectures, targets, and attack methodologies. The most common botnet types affecting IoT networks are centralized botnets, peer-to-peer (P2P) botnets, and hybrid botnets.

2.2.1.1 Centralized Botnets

Centralized botnets are the most traditional type of botnet architecture. In this model, all compromised devices, or bots, communicate with a central server controlled by the botmaster. This central server is often referred to as a **Command and Control (C&C)** server, which sends instructions to the bots and receives data from them.

- **Risks**: Centralized botnets are relatively easy for attackers to control, but they are also more vulnerable to detection and disruption. If the C&C server is identified and taken down by law enforcement or cybersecurity teams, the entire botnet can be dismantled. However, many centralized botnets, like Mirai, have been able to cause significant damage before being neutralized [3].

Example: Mirai botnet – This botnet exploited IoT devices with default credentials and was used to launch massive DDoS attacks [4].

2.2.1.2 Peer-to-Peer (P2P) Botnets

Unlike centralized botnets, **peer-to-peer (P2P) botnets** do not rely on a single C&C server. Instead, bots communicate directly with each other, forming a decentralized network. Each bot in the network can act as both a client and a server, receiving instructions from other bots and passing them along. This makes P2P botnets much more resilient to takedowns, as there is no single point of failure.

- **Risks**: P2P botnets are more challenging to disrupt than centralized botnets, as there is no central command server to target. Even if a portion of the botnet is taken offline, the remaining bots can continue to function and communicate with each other, making P2P botnets particularly dangerous and persistent.

Example: Storm botnet – One of the earliest and most notorious P2P botnets, Storm spread through email spam and P2P networks, evading detection for years [5].

2.2.1.3 Hybrid Botnets

Hybrid botnets combine elements of both centralized and P2P architectures. These botnets often start with a central C&C server but may shift to a P2P mode to evade detection once the botnet is operational. Hybrid botnets provide attackers with both the ease of control offered by centralized systems and the resilience of P2P networks.

- **Risks**: Hybrid botnets are particularly dangerous because they can adapt their communication and control mechanisms based on the level of threat they face. If centralized servers are threatened, the botnet can switch to a P2P model, making detection and disruption more difficult.

Example: Waledac botnet – This botnet began as a centralized network but later incorporated P2P mechanisms to increase its resilience against takedowns [6].

IoT botnets have become a significant threat to the security of both individual devices and the broader internet infrastructure. Due to the sheer number of IoT devices and their often-poor security configurations, attackers can build massive botnets capable of overwhelming even the most robust networks.

- **DDoS Attacks**: As seen with Mirai, IoT botnets can generate enormous volumes of traffic to flood servers and networks, causing widespread outages and service disruptions. These attacks can target critical infrastructure, businesses, and even government agencies, leading to significant financial losses and operational downtime [7].

- **Data Theft and Espionage**: Beyond DDoS attacks, IoT botnets can be used for data theft, as compromised devices often have access to sensitive information. In some cases, attackers have used botnets to conduct espionage by intercepting communications or stealing personal data from IoT devices like smart cameras or voice-activated assistants [8].

- **Malware Propagation**: Botnets can also be used to spread malware, infecting additional devices and further expanding the botnet. This creates a vicious cycle where more and more devices are compromised, providing attackers with greater firepower for subsequent attacks.

Botnets represent one of the most significant security risks to IoT networks. Their ability to compromise vast numbers of devices and launch large-scale attacks highlights the need for enhanced security measures in IoT ecosystems. As IoT continues to expand, the threat posed by botnets is expected to grow, necessitating the development of more sophisticated detection and mitigation techniques, such as Network-based Intrusion Detection Systems (NIDS) and advanced machine learning models like Long Short-Term Memory (LSTM) networks. By focusing on proactive defense mechanisms, organizations can better protect their IoT networks from the evolving threat landscape.

2.3 Long Short-Term Memory (LSTM) Networks

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) specifically designed to model sequential data and capture long-term dependencies in time-series information. Standard RNNs often suffer from the problem of vanishing or exploding gradients, making them less effective when dealing with long sequences of data. LSTM networks address this issue through a unique architecture that allows them to retain information over extended time intervals, making them particularly useful for tasks that require the analysis of sequential patterns.

2.3.1 LSTM Architecture

At the core of an LSTM network are **memory cells**, which can maintain and update information over long periods. Each memory cell is composed of three main gates:

1. **Forget Gate**: This gate determines which information from the previous time step should be discarded. It takes the current input and the hidden state from the previous step and outputs a value between 0 and 1, where 0 represents "forget completely," and 1 represents "keep entirely."

2. **Input Gate**: The input gate controls how much of the new information should be stored in the memory cell. It decides whether the incoming data should update the cell's state or be ignored.

3. **Output Gate**: After the cell's state has been updated, the output gate controls what information will be passed to the next layer or time step. It decides which parts of the cell state will influence the current output.

The memory cells' ability to control what to remember, update, or forget at each time step allows LSTMs to maintain long-term dependencies in the data, making them highly effective for tasks involving time-series or sequential patterns, such as natural language processing, speech recognition, and network traffic analysis.

2.3.2 LSTMs for Attack Classification in IoT Networks

In the context of IoT networks, the traffic generated by devices can be viewed as sequential data, where patterns evolve over time. Anomalous behaviors, such as those caused by cyberattacks, often manifest as deviations from normal traffic patterns. By analyzing network traffic as a time-series, LSTMs are well-suited for detecting these irregularities and classifying them as potential attacks.

In this project, a **LSTM-based neural network** was implemented for attack classification in IoT networks. The model was designed to analyze sequences of network traffic data, learning both normal patterns and deviations that may indicate an attack. The use of LSTM allows the system to capture complex temporal dependencies in the traffic data, making it effective at distinguishing between benign activity and various forms of cyberattacks.

The LSTM model takes as input sequences of features derived from network traffic, such as packet size, protocol type, and timing information. Over time, the model learns to identify patterns that correspond to different types of attacks, such as Distributed Denial of Service (DDoS) attacks, brute-force login attempts, or data exfiltration. By leveraging the IoT-23 dataset, which contains labeled IoT traffic data, the model was trained and evaluated on a wide range of attack types, demonstrating strong performance in detecting and classifying threats in real-time.

2.3.3 Advantages of Using LSTM for Attack Classification

- **Capturing Temporal Dependencies**: One of the primary advantages of LSTM networks is their ability to capture temporal dependencies in network traffic. Attack patterns often unfold over time, making LSTM a natural choice for detecting these anomalies as they emerge.

- **Robustness to Irregular Time Intervals**: In IoT networks, the timing between data packets can vary, and LSTMs can handle this variability better than traditional methods, as they are not constrained by fixed time intervals.

- **Effective for Sequential Data**: IoT traffic, with its sequential nature, benefits greatly from LSTM's capacity to process and learn from ordered data. This makes LSTMs more effective in detecting sophisticated attacks that evolve gradually rather than triggering immediate alerts.

By implementing an LSTM-based neural network in this project, a robust model was created for the classification of cyberattacks in IoT networks. The LSTM's ability to model temporal sequences of network traffic allows it to detect anomalies that would be difficult for other machine learning models to capture. This capability, combined with the use of real-world IoT traffic data, positions the model as an effective tool for strengthening the security infrastructure of IoT environments, providing real-time threat detection and classification.

3 Objectives and Scope of the Project

The primary objective of this project is to design and implement an effective **Long Short-Term Memory (LSTM)-based Intrusion Detection System (IDS)** for classifying cyberattacks in IoT networks. The increasing proliferation of IoT devices, coupled with their inherent vulnerabilities, necessitates the development of advanced security measures to protect these networks. This project aims to address these challenges through the following specific objectives:

1. **Develop a LSTM-based neural network model**: The project seeks to develop an LSTM-based neural network capable of analyzing network traffic in real-time. The model will be trained to detect anomalies and classify different types of cyberattacks, leveraging its ability to capture temporal dependencies in network traffic data.

2. **Utilize the IoT-23 dataset for training and evaluation**: The project will make use of the IoT-23 dataset, a labeled dataset containing both benign and malicious IoT network traffic. The dataset will be used to train the LSTM model, ensuring that it can accurately identify and classify a wide range of attacks in IoT environments.

3. **Enhance attack detection in resource-constrained IoT networks**: By focusing on IoT environments, the project will explore how LSTM-based IDS can be optimized for real-time performance, even in networks with limited processing and memory resources. The model will be evaluated on its ability to detect attacks without overwhelming the IoT devices or introducing significant latency.

4. **Provide a scalable and adaptive intrusion detection solution**: The LSTM-based IDS will be designed to be scalable and adaptive, meaning it should be capable of detecting known and unknown (zero-day) attacks across various IoT environments. This will be achieved by combining signature-based and anomaly-based detection techniques.

5. **Integrate edge computing principles for real-time detection**: The project will leverage edge computing principles to process network traffic closer to the source, reducing latency and ensuring faster detection and response to threats. This approach will enable the IDS to operate efficiently in distributed IoT networks, providing real-time threat monitoring and mitigation.

The scope of this project is focused on the development and evaluation of a **LSTM-based Intrusion Detection System (IDS)** specifically for IoT networks. The project covers the following areas:

1. **IoT Network Security**: The project is limited to IoT environments and focuses on identifying security risks inherent to these networks. While the methodology may be applicable to other types of networks, the model is optimized for IoT devices' specific challenges, such as limited computational resources and network traffic patterns.

2. **Network Traffic Analysis**: The scope of the project involves analyzing network traffic data, specifically identifying patterns of malicious activity. The IDS will focus on

detecting various types of cyberattacks, including Distributed Denial of Service (DDoS) attacks, data exfiltration, brute-force attacks, and others, as found in the IoT-23 dataset.

3. **LSTM-based Model Development**: The project is centered on the application of Long Short-Term Memory (LSTM) neural networks, which will be used to analyze the temporal dependencies in network traffic data. The LSTM model is chosen for its ability to handle sequential data and its effectiveness in detecting attacks that develop over time.

4. **Real-Time Detection**: A key aspect of the project is real-time intrusion detection. The system will be designed to operate in real-time, ensuring that cyberattacks are detected promptly, thus enabling rapid response and mitigation of security threats.

5. **Evaluation and Testing**: The project will include an evaluation of the LSTM-based IDS using the IoT-23 dataset. Metrics such as detection accuracy, false-positive rate, and system performance will be analyzed to ensure the system's effectiveness in practical IoT environments.

6. **Limitations and Assumptions**: While the project focuses on the security of IoT networks, it does not address other aspects of IoT, such as hardware vulnerabilities or physical layer security. Additionally, the IDS will be designed for network-level detection and may not fully capture device-level security threats. Furthermore, it is assumed that IoT devices in the network communicate using standard protocols, as modeled by the dataset.

The objectives and scope of this project are framed around developing a robust, scalable, and real-time intrusion detection system based on LSTM networks, tailored for the unique requirements of IoT networks. By leveraging the strengths of LSTM in analyzing time-series data and employing edge computing principles, the project aims to enhance the security infrastructure of IoT environments, providing a practical and adaptive solution for threat detection in real-time.

4 Benefits of the project

The development of a **Long Short-Term Memory (LSTM)-based Intrusion Detection System (IDS)** for IoT networks, as outlined in this project, offers several significant benefits for both academic research and practical applications. These benefits are crucial given the growing reliance on IoT devices in various domains and the increasing threats these networks face. The key benefits of this work are as follows:

1. Enhanced Security for IoT Networks

One of the primary benefits of this project is the improvement of security in IoT networks, which are notoriously vulnerable due to their resource constraints, lack of standardized security protocols, and widespread deployment. The LSTM-based IDS provides an advanced mechanism for detecting a wide range of cyberattacks in real-time, allowing for proactive defense measures. By continuously monitoring network traffic, the system can detect abnormal behaviors indicative of attacks, such as Distributed Denial of Service (DDoS), brute force attempts, and malware infections, before they cause significant damage.

2. Real-Time Threat Detection

Real-time detection is critical in mitigating the impact of cyberattacks, especially in IoT environments where attacks can propagate quickly across devices. The LSTM-based IDS, integrated with edge computing principles, allows for faster threat detection and response times by processing data closer to the source. This significantly reduces the time between the identification of malicious activity and the implementation of countermeasures, minimizing potential disruptions and data breaches in IoT systems.

3. Effective Time-Series Analysis

IoT networks generate large volumes of sequential data, making time-series analysis crucial for detecting temporal patterns associated with attacks. LSTM networks are well-suited for this type of analysis, as they can model long-term dependencies in network traffic and detect gradual changes in behavior that may indicate the onset of an attack. This makes the system highly effective at recognizing sophisticated, multi-stage attacks that evolve over time, which might otherwise go undetected by traditional intrusion detection systems.

4. Scalable and Adaptive Solution

The LSTM-based IDS developed in this project is designed to be scalable, making it adaptable to different IoT environments with varying levels of traffic and device density. The system can efficiently handle large-scale IoT networks, ensuring that it remains responsive and effective even as the number of devices grows. Additionally, its adaptive nature, supported by machine learning, allows it to detect both known attack patterns (through signature-based detection) and emerging threats (through anomaly detection), offering a more comprehensive security solution for IoT networks.

5. Reduced False Positives

Intrusion detection systems often suffer from high false-positive rates, where legitimate activities are incorrectly flagged as malicious. This project addresses this challenge by utilizing LSTM's ability to learn from past network behavior and distinguish between normal variations in traffic and actual security threats. Through its sequential learning capabilities, the system can minimize false alarms, which reduces the burden on security teams and allows them to focus on genuine threats.

6. Increased Resilience Through Edge Computing

By leveraging edge computing, this project enhances the resilience and efficiency of the IDS. Processing data at the edge reduces network congestion and central server load, making the system more responsive and less prone to delays. This is particularly important in distributed IoT networks, where latency can compromise security effectiveness. Edge computing also reduces the risk of a single point of failure, increasing the overall robustness of the security system.

7. Contribution to Research in Machine Learning for Cybersecurity

This project contributes to the growing body of research on the application of machine learning, particularly LSTM networks, in cybersecurity. The implementation and evaluation of LSTM for IoT network intrusion detection not only demonstrate its practical effectiveness but also open new avenues for future research in both academia and industry. This work showcases how advanced neural network architectures can be applied to real-world problems, paving the way for further innovations in AI-driven security solutions.

8. Practical Application for Industry

The outcomes of this project can have direct applications in industries that rely heavily on IoT networks, such as healthcare, manufacturing, smart cities, and critical infrastructure. The deployment of an LSTM-based IDS tailored for IoT networks can provide these industries with a more reliable and sophisticated layer of security. This is particularly important for protecting sensitive data, ensuring operational continuity, and preventing financial losses that could arise from cyberattacks on IoT systems.

9. Detecting Zero-Day and Evolving Threats

A significant benefit of the LSTM-based IDS is its ability to detect zero-day attacks and other evolving threats. Traditional signature-based systems are limited to recognizing known attack patterns, but the anomaly detection capabilities of LSTM networks allow the system to identify previously unknown attacks based on deviations from normal network behavior. This proactive approach is essential in today's rapidly changing threat landscape, where attackers constantly develop new tactics to bypass existing defenses.

10. Efficiency in Resource-Constrained Environments

Many IoT devices are resource-constrained, with limited computational power, memory, and battery life. The LSTM-based IDS developed in this project is optimized to operate efficiently in such environments, ensuring that security can be maintained without overburdening the devices or the network infrastructure. This makes the system particularly suitable for widespread deployment in real-world IoT networks, where resource constraints are a common challenge.

This project offers multiple benefits, both in terms of improving IoT network security and contributing to the field of machine learning-based intrusion detection systems. By developing a scalable, adaptive, and efficient LSTM-based IDS, the project addresses the unique challenges posed by IoT networks and provides a robust solution for real-time attack classification. The system’s ability to detect zero-day threats, minimize false positives, and function in resource-constrained environments positions it as a valuable tool for securing the increasingly interconnected world of IoT.

5 Related works

The integration of machine learning, artificial intelligence (AI), and neural networks (NN) has become increasingly prevalent in the realm of IoT cybersecurity. These technologies offer sophisticated tools for detecting and responding to threats in a landscape that is constantly being targeted by cyberattacks.

In their paper titled "Edge Computing for Real-Time Botnet Propagation Detection" Mikhail Gromov, David Arnold, and Jafar Saniie [4] explore the utilization of edge computing coupled with Convolutional Neural Networks (CNNs) for real-time botnet attack detection. Their research leverages the N-BaIoT dataset and proposes a system that operates at the network edge, specifically using a Jetson Nano to perform live traffic analysis. This setup is crucial for stopping botnet propagation swiftly, enhancing the responsiveness of IoT devices to imminent threats. In addition to their existing contributions, the authors have engaged in extensive research involving the application of AI for the detection of malicious attacks [5]. Notably, in their study titled "Utilizing Computer Vision Algorithms to Detect and Classify Cyberattacks in IoT Environments in Real-Time" [6, 7] the authors employ the IoT-23 dataset, the same dataset used in this current research Focusing on LSTM neural networks, the "Deep-IDS: A Real-Time Intrusion Detector for IoT Nodes Using Deep Learning" paper by Sandeepkumar Racherla et al. [8] demonstrates a significant advancement in intrusion detection within IoT systems. This study employs a streamlined LSTM architecture, which includes 64 LSTM units, to detect a variety of common IoT threats such as Denial of Service and Brute Force attacks using the CIC-IDS2017 dataset. Their approach underscores the efficacy of LSTM models in handling real-time data processing, providing a balance between high detection rates and low false alarm rates.

Finally, the paper "Enhanced Intrusion Detection with LSTM-Based Model, Feature Selection, and SMOTE for Imbalanced Data" research [8] study leverages LSTM's capability to model sequential data effectively, alongside SMOTE to address class imbalances prevalent in cybersecurity datasets. The combination of LSTM and sophisticated feature selection strategies enhances the system's accuracy and efficiency, making it a robust tool against diverse network threats.

6 Alternative Analysis

When developing a solution for intrusion detection in IoT networks, selecting the appropriate programming language and tools for packet capture and network traffic analysis plays a critical role in the project's performance and efficiency. This section presents an analysis of two key decisions made during the project: the choice of **Python** over **C++** and the use of **Scapy** instead of **Pyshark** for network packet analysis. Each alternative has its own set of advantages and disadvantages, and these decisions were carefully considered to meet the project's objectives.

6.1 Python vs C++

In this project, Python was chosen as the primary programming language for developing the LSTM-based Intrusion Detection System (IDS). However, it is important to evaluate the potential benefits and drawbacks of using Python versus C++, another widely used language in network security applications.

6.1.1 Python

Python is a high-level, interpreted programming language known for its simplicity, readability, and extensive library support. Its popularity in machine learning, data science, and network programming has made it a go-to language for many security and AI-related projects.

- **Advantages**:
  - **Ease of Use and Rapid Development**: Python's simple syntax and extensive libraries, such as TensorFlow, PyTorch, and Keras for machine learning, allow for rapid prototyping and development of complex models like LSTM networks. This ease of use reduces development time and lowers the learning curve, especially when working with large datasets and neural networks.
  - **Extensive Library Support**: Python has a rich ecosystem of libraries for network analysis (e.g., Scapy, Pyshark), data manipulation (e.g., Pandas, NumPy), and deep learning (e.g., TensorFlow, Keras). This makes it highly versatile for tasks such as network traffic analysis, feature extraction, and model training.
  - **Cross-Platform Compatibility**: Python is cross-platform, which allows the developed system to run on multiple operating systems with minimal changes. This flexibility is particularly useful for projects involving IoT devices deployed in diverse environments.
- **Disadvantages**:
  - **Performance**: One of Python's primary drawbacks is its performance compared to compiled languages like C++. Python is an interpreted language, meaning it executes code slower than C++ in CPU-intensive tasks. For real-time network traffic analysis in high-speed IoT networks, this can be a limitation, though mitigated in this project through careful optimization and the use of efficient libraries.

  - **Memory Management**: Python's automatic memory management through garbage collection can lead to inefficiencies in resource-constrained environments, such as IoT devices with limited memory and processing power.

6.1.2 C++

C++ is a compiled, high-performance programming language widely used in systems programming, game development, and real-time applications. For network security projects, C++ is often favored for its speed and control over system resources.

- **Advantages**:
  - **High Performance**: C++ is significantly faster than Python due to its compiled nature and ability to optimize code at a low level. This makes it ideal for real-time applications, such as high-throughput network traffic analysis, where performance is a critical factor.
  - **Fine-Grained Memory Management**: C++ provides more control over memory allocation and management, which can be a critical advantage in resource-constrained environments like IoT devices. This enables more efficient use of system resources, reducing overhead in real-time systems.
  - **Concurrency Support**: C++ offers better support for multi-threading and parallel processing, which can improve performance when handling large volumes of network traffic or running real-time intrusion detection tasks.
- **Disadvantages**:
  - **Complexity**: C++ is a more complex language with a steeper learning curve compared to Python. The need for manual memory management, pointer handling, and more verbose syntax can increase development time and complexity, particularly for machine learning tasks where Python's ease of use shines.
  - **Limited Machine Learning Ecosystem**: While libraries like TensorFlow have C++ APIs, the machine learning ecosystem in C++ is not as mature or developer-friendly as in Python. This makes implementing complex models, such as LSTM networks, more challenging and time-consuming.

6.1.3 Conclusion

The choice of Python in this project was driven by the need for rapid development, ease of use, and access to a rich set of libraries for machine learning and network analysis. While C++ offers better performance, Python's flexibility and extensive ecosystem made it the preferred choice for building the LSTM-based IDS. The trade-off in performance was considered acceptable for the scope of this project, especially since Python's rich libraries enabled efficient handling of the required tasks.

6.2 Pyshark vs Scapy

In network traffic analysis, the choice of tools is equally important. For this project, **Scapy** was chosen over **Pyshark** for network packet analysis and traffic monitoring. Both tools are widely used in the field, and each has unique strengths and weaknesses.

6.2.1 Scapy

Scapy is a powerful Python-based network analysis tool that allows users to capture, manipulate, and analyze network packets. It is widely used in cybersecurity for tasks such as packet sniffing, crafting, and injecting packets into a network.

- **Advantages**:
    - **Flexibility**: Scapy is highly flexible, allowing users to easily create, modify, and inject network packets. This is particularly useful in testing scenarios where custom packets or specific protocol manipulation is required.
    - **Low-Level Control**: Scapy provides extensive control over network packets and protocols, enabling detailed packet crafting and analysis. This is a key feature when experimenting with different network protocols and simulating specific attack scenarios, as required for this project.
    - **Integration with Python**: Since Scapy is written in Python, it integrates seamlessly with the machine learning and data processing components of the project. This allows for easy extraction of features from network traffic and their direct use in the LSTM model for attack classification.
- **Disadvantages**:
    - **Packet Capture Speed**: While Scapy is powerful, it may not be as fast as dedicated packet capture libraries such as Pyshark (which is built on top of Tshark, a part of Wireshark). Scapy's performance can be a limitation when dealing with large volumes of network traffic, though this was manageable within the project’s scope.

6.2.2 Pyshark

Pyshark is a Python wrapper for Tshark, the command-line interface of Wireshark, which is the industry-standard tool for network protocol analysis. Pyshark allows for packet capture and analysis using Wireshark’s powerful capabilities.

- **Advantages**:
    - **Speed and Efficiency**: Pyshark is built on Tshark, making it highly efficient in capturing and analyzing large volumes of network traffic. This makes it a suitable choice for high-speed network environments where performance is a critical requirement.

  - **Comprehensive Protocol Support**: Since Pyshark leverages Wireshark, it supports a wide range of network protocols, ensuring compatibility with various types of network traffic and enabling in-depth packet analysis.

  - **Ease of Use for Capture**: Pyshark simplifies the process of capturing and dissecting packets, providing a high-level interface to Wireshark's capabilities. For straightforward packet analysis tasks, Pyshark offers a more user-friendly approach compared to Scapy.

- **Disadvantages**:

  - **Limited Flexibility**: Pyshark is primarily designed for packet capture and analysis and lacks the low-level packet crafting and manipulation features that Scapy offers. For projects that require custom packet injection or network protocol testing, Pyshark is less suited than Scapy.

  - **Dependency on Tshark**: Pyshark requires Tshark to be installed, adding an additional dependency to the project. This can complicate deployment in certain environments, especially when trying to keep dependencies minimal.

6.2.3 Conclusion

Scapy was chosen for this project due to its flexibility and the need for detailed packet crafting and manipulation, which were critical in testing specific attack scenarios and simulating network traffic. While Pyshark provides faster packet capture and analysis, the ability to fully control and modify packets in Scapy aligned better with the project's objectives, allowing for deeper experimentation and integration with Python's machine learning tools.

7 Methodology

The methodology of the project will be explained in this section of the report.

7.1. IoT-23 dataset

The IoT-23 dataset [3] is a comprehensive collection of network traffic data, carefully curated to support the development and evaluation of Intrusion Detection Systems (IDS) in Internet of Things (IoT) environments. Developed by the Stratosphere Laboratory, the dataset addresses the growing need for realistic IoT network traffic data to enhance attack detection and classification. The dataset is composed of 23 scenarios, of which 3 scenarios represent benign traffic, and 20 scenarios represent malicious traffic, providing a rich resource for researchers and practitioners in the field of IoT security.

The dataset includes traffic captured from three widely used IoT devices:

1. **Philips Hue smart LED lamp**: A smart lighting system with internet connectivity for remote control and automation.
2. **Amazon Echo**: A voice-activated personal assistant device designed to control smart home devices and interact with cloud services.
3. **Somfy smart door lock**: A smart lock system that enables users to remotely manage physical access via internet connectivity.

These devices were chosen to provide a representative sample of typical IoT devices, each with distinct communication protocols and patterns, thereby providing insight into both normal and malicious IoT network traffic.

The IoT-23 dataset is composed of two main types of files:

1. **PCAP Files**: These packet capture files contain raw network packets recorded from the IoT devices. PCAP files allow for a detailed examination of network traffic, including packet-level data such as headers, payloads, and timestamps. The PCAP files were used in this project to extract network flows and derive features necessary for training the LSTM-based neural network.
2. **Labeled Zeek Flow Files**: These files summarize network connections using the Zeek network monitoring framework. Each flow file includes metadata about network connections, such as duration, packet counts, and byte transfers, as well as labels indicating whether the traffic is benign or malicious. The labeled flow files were instrumental in assigning accurate labels to the flows extracted from the PCAP files.

For this project, **18 PCAP files** and their corresponding Zeek flow files were utilized, comprising **3 benign captures** and **15 malware captures**. The use of these files provided a balanced dataset for training the LSTM model, with a variety of attack types and benign traffic included in the analysis.

To manage the dataset size and ensure efficient processing, only the first **300,000 packets** from each PCAP file were used for analysis. This decision allowed for a balance between computational efficiency and maintaining a diverse set of network traffic data. In total, **1,851,432 network flows** were extracted from the PCAP files. These flows were then used to generate features such as packet size, inter-arrival time between packets, and protocol types, which were crucial for training the LSTM-based IDS model.

The labeled Zeek flow files were used to assign ground truth labels to the extracted network flows, ensuring the accuracy of the data used for training the machine learning model.

The IoT-23 dataset contains **10 distinct labels**, representing both benign and malicious network traffic. These labels were used in the project to train the LSTM-based model for intrusion detection and attack classification. The distribution of labels, along with their corresponding IDs and flow counts, is shown below:

Table I. Labels

Label name	Label ID	Number of Flows
Benign	1	242,133
C&C	2	4,128
C&C-FileDownload	3	13
C&C-HeartBeat	4	575
C&C-Torii	6	14
FileDownload	7	12
Okiru	8	710715
DDoS	9	212
Attack	10	3,729
PartOfHorizontalPortScan	11	889,898

These labels encompass both benign network traffic (Label 1) and various types of malicious activity. A number of malicious labels are associated with activities that contribute to **botnet formation** and other types of cyberattacks. Key examples include:

- **C&C (Command and Control)**: Traffic between a compromised device and a command and control server, used by attackers to manage botnets or issue instructions to infected devices.
- **C&C-FileDownload**: A subtype of C&C traffic where malicious files are downloaded onto compromised IoT devices.
- **Okiru**: A malware family designed to infect IoT devices and use them in botnets to launch large-scale cyberattacks, such as DDoS attacks.
- **PartOfHorizontalPortScan**: A technique used by attackers to scan multiple devices for open ports, often preceding larger attacks, such as malware infections or DDoS attacks.

Several labels, including Okiru, C&C, and PartOfHorizontalPortScan, represent activities that are integral to the formation and management of botnets. Botnets are networks of compromised devices controlled by an attacker, and they are often used to launch large-scale attacks, such as Distributed Denial of Service (DDoS) attacks or data theft.

- **Okiru**: This malware is designed to exploit vulnerabilities in IoT devices, turning them into part of a botnet that can be used for further malicious activities, such as participating in DDoS attacks.

- **C&C Traffic**: Command and Control traffic is vital to understanding how attackers maintain control over botnets. The communication between botnet controllers and infected devices is essential for issuing commands, downloading additional malware, and orchestrating attacks.

- **Port Scanning**: Horizontal port scanning is commonly used to identify vulnerabilities in a range of IoT devices. Once vulnerabilities are discovered, attackers can compromise the devices and add them to a botnet.

The IoT-23 dataset provides an extensive and realistic dataset of IoT network traffic, offering a diverse range of benign and malicious activities for research purposes. By incorporating network traffic from widely used IoT devices and a variety of malware types, the dataset enables researchers to develop and evaluate machine learning models for IoT security. In this project, the dataset was used to train an LSTM-based Intrusion Detection System, leveraging both PCAP and labeled Zeek flow files to identify and classify attacks in IoT networks. The use of real-world attack scenarios and a broad spectrum of malware types ensures that the resulting model is both robust and applicable to modern IoT environments.

7.2 Data preprocessing

The data preprocessing phase was a critical step in preparing the network traffic data captured in the **PCAP (Packet Capture) files** for analysis using the **LSTM neural network**. This phase ensured that raw network data was transformed into a structured format that could be efficiently processed by the machine learning model. For this project, the **Pyshark** library was employed to parse the PCAP files and extract a comprehensive set of features from each network flow, allowing the LSTM model to detect patterns indicative of benign or malicious activity. The preprocessing phase included several key steps: **feature extraction**, **data labeling**, and **encoding and data splitting**.

7.2.1 Feature Extraction

Feature extraction is an essential step in transforming raw network traffic into a format suitable for machine learning. For each flow captured in the PCAP files, various features were extracted that encapsulated the behavior and characteristics of the network traffic. These features were chosen to capture not only basic flow information but also statistical properties that could reveal underlying patterns in the communication between devices. In total, **24**

features were selected to train the LSTM neural network, encompassing aspects of traffic volume, timing, packet statistics, and protocol information.

The table below provides an overview of the features extracted for each network flow:

Table II. Extracted features

Feature Name	Description
Source IP	The IP address of the device that initiated the network flow.
Destination IP	The IP address of the device that received the flow.
Source Port	The port number used by the source device for communication.
Destination Port	The port number used by the destination device.
Protocol	The transport or application layer protocol used (e.g., TCP, UDP, HTTP, DNS).
Packet Size (Mean)	The average packet size in bytes for all packets in the flow.
Packet Size (Median)	The median packet size in bytes.
Packet Size (Standard Deviation)	The standard deviation of packet sizes, indicating variability in packet size.
Packet Size (Min)	The smallest packet size in the flow.
Packet Size (Max)	The largest packet size in the flow.
Inter-Packet Arrival Time (Mean)	The average time between consecutive packets within the flow, measured in seconds.
Inter-Packet Arrival Time (Median)	The median time between consecutive packets.
Inter-Packet Arrival Time (Standard Deviation)	The variability in the timing of packet arrivals.
Inter-Packet Arrival Time (Min)	The shortest time between consecutive packets.
Inter-Packet Arrival Time (Max)	The longest time between consecutive packets.
Flow Duration	The total duration of the flow, measured from the first to the last packet in seconds.

Forward Packet Count	The number of packets sent from the source to the destination device.
Backward Packet Count	The number of packets sent from the destination back to the source device.
Forward Bytes	The total number of bytes sent from the source to the destination.
Backward Bytes	The total number of bytes sent from the destination back to the source.
Connection State	The state of the connection (e.g., initiated, established, reset, terminated).
Origin Bytes	The total number of bytes sent by the source (client) in the flow.
Response Bytes	The total number of bytes sent by the destination (server) in response.
Total Packets	The sum of forward and backward packets in the flow.

Key Features:

1. **Source and Destination IP/Port**: These features uniquely identify each flow by specifying the origin and destination of the communication. They are essential for tracking the direction of the traffic and recognizing patterns related to specific devices or ports. Malicious flows often target specific IPs or use unusual port numbers.

2. **Protocol**: Knowing the protocol used in a communication flow is crucial for differentiating between different types of traffic. For example, malicious actors may use uncommon or non-standard protocols to avoid detection.

3. **Packet Size and Inter-Packet Arrival Time**: These features help capture the size and timing patterns of network traffic. Irregularities in packet size distribution or abnormal timing between packets could indicate suspicious activities like denial-of-service attacks or botnet communication.

4. **Flow Duration**: The total duration of a flow provides insight into whether a connection lasted unusually long or short, which can be a characteristic of certain attacks. For example, a very short connection with high data throughput could be a sign of a malware infection attempting to exfiltrate data.

5. **Packet Counts and Bytes**: These features measure the volume of traffic moving in both directions between devices. In many cases, an abnormal number of packets or bytes (e.g., a large number of requests with no responses) can signify a brute force attack or other anomalous activity.

6. **Connection State**: Monitoring the state of the connection lifecycle (initiation, establishment, reset, termination) helps track irregular patterns in network behavior, such as abrupt connection terminations that could indicate an ongoing attack or compromised device.

In total, **24 features** were extracted and processed for each flow, capturing both the high-level behavior and low-level statistical properties of network traffic. These features were designed to maximize the model's ability to distinguish between benign and malicious flows, while also providing sufficient contextual information about each communication.

7.2.2 Data Labeling

To ensure accurate training and testing of the LSTM model, each network flow needed to be correctly labeled as either benign or malicious. The labeling process was conducted by comparing the IP addresses and ports extracted from the PCAP files with those cataloged in the Zeek flow files. The Zeek files contained pre-labeled data, indicating whether each connection in the network was associated with a known malware type or represented normal, benign behavior.

By matching the IPs, ports, and other flow characteristics with the information from the Zeek flow files, each network flow extracted from the PCAP files was accurately classified. This step was critical in maintaining the integrity of the training and testing datasets, ensuring that the LSTM model was trained on reliable and correctly labeled data. The labels are shown in Table 1.

7.2.3 Encoding and Data Splitting

After the features were extracted and labeled, additional preprocessing steps were applied to prepare the dataset for the LSTM neural network. Specifically, **One-Hot Encoding** was applied to transform categorical features (such as protocols and connection states) into a binary matrix format that could be used as input to the model. One-Hot Encoding is a common technique in machine learning for converting categorical variables into a format that is easy for neural networks to interpret.

Once encoding was complete, the dataset was split into two subsets:

- **Training Set (80%)**: Used to train the LSTM neural network. This set was carefully curated to ensure that the model could learn from both benign and malicious traffic, including a variety of attack types.

- **Testing Set (20%)**: Reserved for evaluating the performance of the trained model. This set allowed for an unbiased assessment of the model's ability to generalize to new, unseen network flows.

This **80/20 split** was selected to balance the need for a robust training process while still maintaining sufficient data for effective model validation.

The entire preprocessing pipeline is summarized in the following flowchart, outlining the steps from raw PCAP file parsing, feature extraction, data labeling, encoding, and data splitting, to the final datasets ready for input into the LSTM neural network.

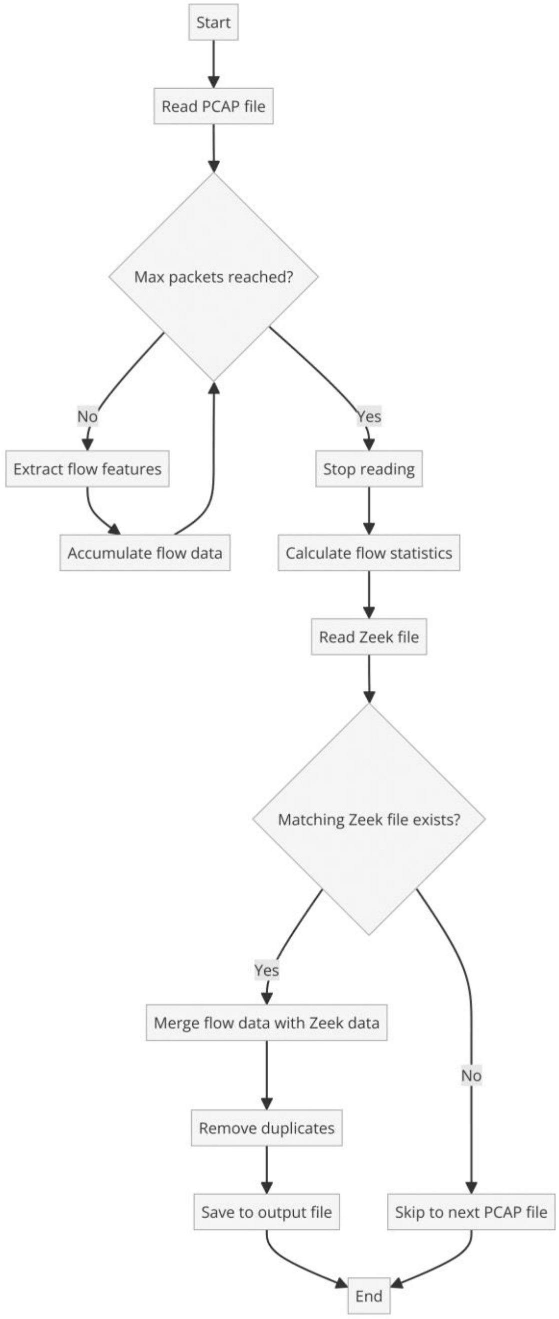


Figure I. DATA PREPROCESSING FLOWCHART

7.3 LSTM Model Architecture

The Long Short-Term Memory (LSTM) model was developed to classify network attacks within IoT environments by leveraging the sequential nature of network traffic data. This architecture, implemented using the Keras library [11], is designed to exploit the temporal dependencies in the data, which is crucial for identifying patterns indicative of benign or malicious activities. LSTM networks are particularly suited for time-series data, as they excel at retaining information over extended sequences, making them highly effective in network intrusion detection tasks.

The following provides a detailed description of the layers and configurations used in the model, which is visually depicted in Figure II.

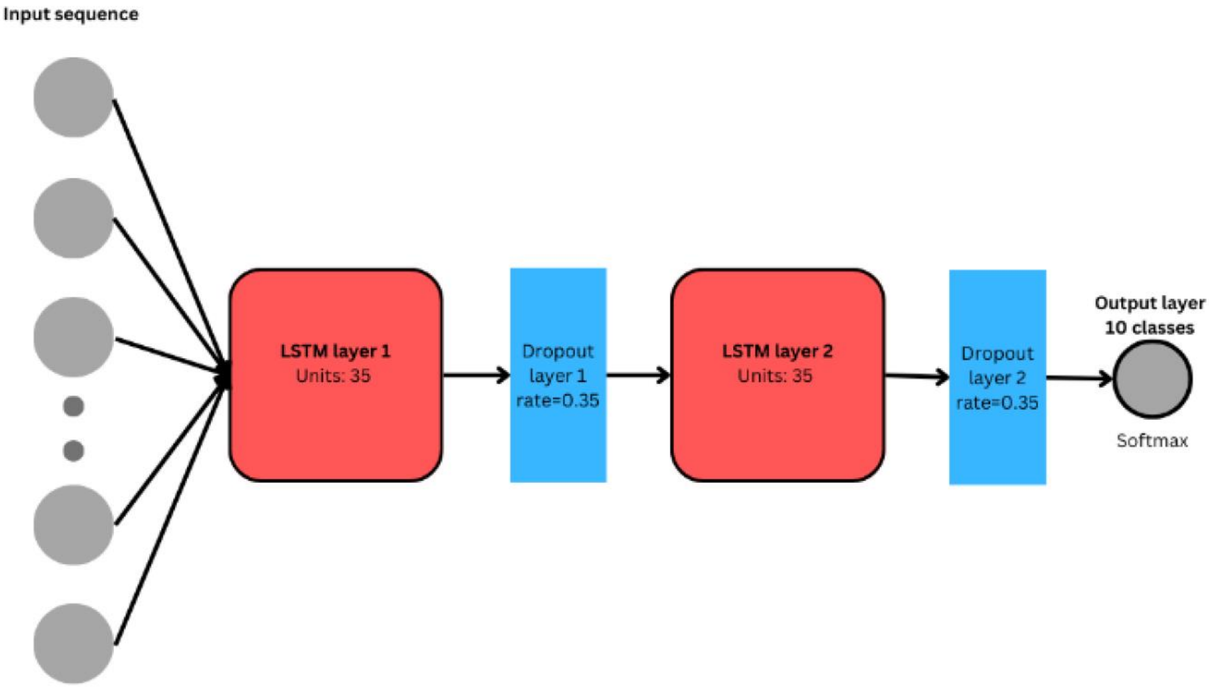


Figure II. LSTM model architecture

1. **Input Layer**:

   The input layer is configured to receive sequences of data corresponding to the 24 extracted features from the IoT network flows. These features include statistics such as packet sizes, inter-packet arrival times, flow durations, and protocol information (as described in the data preprocessing section). Each input sequence is normalized to ensure that the features are on a consistent scale, facilitating faster convergence during model training. The shape of the input data is thus defined by the sequence length (number of network flows in each sequence) and the 24 features.

2. **First LSTM Layer**:

   The first LSTM layer consists of **35 LSTM units (neurons)**. It is configured with return_sequences=True, meaning that it outputs the full sequence of hidden states to the next layer, rather than a single final output. This is essential for capturing long-range dependencies across the input sequence, which is particularly important in network traffic data where the behavior of one packet may influence packets further down the sequence. This layer processes the sequential data while retaining memory of previous events, allowing the model to learn both short-term and long-term patterns in the network flows.

3. **Dropout Layer (1st Dropout)**:

   Following the first LSTM layer, a **Dropout layer** is applied with a dropout rate of **0.35**. Dropout is a regularization technique used to prevent overfitting by randomly setting a fraction of the input units to zero during training. This forces the network to learn more robust features, as it cannot rely on any single neuron being consistently active. The chosen dropout rate was determined through hyperparameter tuning, ensuring an optimal balance between preventing overfitting and maintaining model performance.

4. **Second LSTM Layer**:

   The second LSTM layer also contains **35 LSTM units**, but unlike the first, it is configured to output only the final hidden state (return_sequences=False). This means that rather than returning the entire sequence of hidden states, the layer produces a single output representing the accumulated knowledge from the entire input sequence. This output condenses the learned features and is used as input to the next layer for final classification. The use of two LSTM layers enhances the model's ability to capture both high-level and detailed temporal patterns in the network traffic.

5. **Dropout Layer (2nd Dropout)**:

   Another **Dropout layer** is applied with the same dropout rate of **0.35**, further helping to reduce the risk of overfitting by introducing noise during training. This second dropout layer ensures that the final output from the second LSTM layer is less dependent on specific units, increasing the model’s generalization ability when dealing with unseen network traffic.

6. **Output Layer**:
   The final layer is a **fully connected (dense) output layer**, with **10 output units**, corresponding to the **10 classes** present in the dataset. These classes represent different attack types as well as benign network traffic. The output layer employs a **softmax activation function**, which converts the output into a probability distribution over the possible classes. The softmax function ensures that the sum of all class probabilities equals 1, making it ideal for multiclass classification tasks like this one, where the model must assign an input sequence to one of several attack types or benign traffic.

The model is trained using the Adam optimizer, a popular choice for deep learning models due to its adaptive learning rate and ability to efficiently converge on optimal solutions. Adam combines the advantages of two other extensions of stochastic gradient descent—AdaGrad and RMSProp—allowing for faster convergence and better handling of sparse gradients. In this project, Adam is used to minimize the categorical cross-entropy loss, which is the standard loss function for multiclass classification tasks. Cross-entropy measures the difference between the predicted probability distribution and the true labels, guiding the model in adjusting its weights to minimize this difference during training.

To prevent overfitting and improve the generalization of the model, the Optuna library was used for hyperparameter tuning. Optuna is an open-source library that automates the search for optimal hyperparameters by systematically testing different combinations of parameters such as the number of LSTM units, dropout rates, and learning rates. This method ensures that the model is fine-tuned to perform optimally given the specific characteristics of the IoT-23 dataset. The final configuration of 35 units and a dropout rate of 0.35 was chosen based on these optimization trials.

The LSTM model was trained over 50 epochs, with each epoch representing a complete pass through the training dataset. This number of epochs was determined based on preliminary experiments, which showed that training for 50 epochs provided the best balance between model accuracy and overfitting. Training for fewer epochs resulted in the model not fully learning the complex patterns in the data, while training for more epochs risked overfitting to the training set.

During training, the model adjusted its weights to capture the intricate patterns in the network traffic, learning to distinguish between benign traffic and various types of cyberattacks. The performance of the model was monitored using the accuracy metric, which measures the percentage of correctly classified instances out of the total number of predictions.

This LSTM architecture, with its two LSTM layers and carefully applied dropout regularization, was designed to efficiently process network flow data for attack classification. By leveraging the temporal characteristics of IoT network traffic, the model is capable of identifying patterns indicative of different types of attacks, including botnet activity, DDoS attacks, and other forms of malicious behavior. The use of Optuna for hyperparameter optimization ensured that the model was fine-tuned to achieve the best possible performance on the IoT-23 dataset.

The model's architecture and training process demonstrate a robust approach to detecting anomalies and classifying attacks in real-time, providing a scalable solution for intrusion detection in IoT networks.

7.4 System Methodology and Process Flow

The overall methodology for developing and evaluating the Long Short-Term Memory (LSTM)-based Intrusion Detection System (IDS) follows a structured, step-by-step process. This ensures that the system is built on a solid foundation of clean, well-prepared data and is evaluated accurately to assess its effectiveness in detecting network attacks. The general process is depicted in Figure 3, and the following is a detailed explanation of each step involved in the system:

1. Load Data

The first step in the process involves loading the network traffic data, both **benign** and **malicious**, from the **PCAP files** in the IoT-23 dataset. These files contain raw packet capture data representing various scenarios of network traffic, including regular IoT device communication and traffic generated by malware and other types of cyberattacks. Loading the data is a crucial step, as the PCAP files provide the basis for extracting meaningful features and constructing the dataset for the LSTM model. This step ensures that the system can work with both benign traffic (such as regular device usage) and malicious traffic (such as Distributed Denial of Service (DDoS) or botnet communication), giving the model a diverse and comprehensive set of data to learn from.

2. Preprocess Data

Once the network traffic data is loaded, the next step is to preprocess the data. This step includes two major tasks:

- **Feature Extraction**: Using the Pyshark library, features are extracted from the raw network flows captured in the PCAP files. These features include packet size statistics, inter-packet arrival times, protocol types, flow durations, and more, providing a rich set of attributes that describe each flow. The extraction of these features transforms the raw traffic data into a structured format suitable for machine learning.
- **Data Labeling**: After feature extraction, each flow is labeled as either benign or malicious by comparing the extracted features (such as IP addresses and ports) with the Zeek flow files. The Zeek files contain pre-labeled data, which provides a ground truth reference for distinguishing between normal and malicious traffic. This labeling process is essential for ensuring that the training and testing datasets are properly classified, allowing the LSTM model to learn how to differentiate between benign and malicious behaviors.

3. Train-Test Split

To ensure that the LSTM model is evaluated accurately, the dataset is split into **training** and testing subsets. This step is critical for building a robust machine learning model. Typically, 80% of the data is allocated for training the model, and 20% is reserved for testing. The training subset is used to teach the model how to recognize patterns in the network traffic, while the testing subset is held back to evaluate how well the model performs on unseen data. This split allows the system to avoid overfitting and ensures that the model generalizes well to new traffic patterns.

4. Encode Labels

Since machine learning models, particularly neural networks, require numerical inputs, the labels in the dataset (e.g., benign or different types of malicious traffic) must be encoded into a format suitable for the LSTM model. In this step, One-Hot Encoding is applied to convert the categorical labels into a binary matrix format. This encoding method ensures that each class label is represented as a binary vector, where the vector length equals the number of classes in the dataset, and a 1 indicates the presence of a specific class. For example, a network flow labeled as "benign" might be encoded as [1, 0, 0, ..., 0], while an attack type might be encoded as [0, 0, 1, ..., 0]. This transformation is necessary to train the model effectively.

5. Normalize Features

In this step, the extracted features are normalized to ensure that they are on a consistent scale before being fed into the LSTM model. Normalization is a preprocessing technique that adjusts the scale of the features so that they fall within a common range (e.g., between 0 and 1). This is important because features with larger numerical ranges can disproportionately influence the model's learning process. By standardizing the input data, normalization allows the model to converge more quickly during training and prevents any one feature from dominating the others. Common normalization methods include min-max scaling or z-score standardization.

6. Train Model

Once the data has been preprocessed, the system proceeds to the **training phase**. During this step, the preprocessed and normalized network traffic data is fed into the LSTM model. The model learns by adjusting its internal weights through backpropagation and gradient descent. In this case, the Adam optimizer is used to optimize the learning process, while categorical cross-entropy is employed as the loss function to measure the model's performance during training.

The model is trained over a series of epochs, where each epoch represents one complete pass through the training data. For this project, the LSTM model was trained for 50 epochs, as preliminary experimentation showed that this number of epochs provided an optimal balance between comprehensive learning and avoiding overfitting. During training, the model adjusts its parameters to minimize the loss function and improve its accuracy in classifying the different types of network traffic.

7. Evaluate Model

After the model has been trained, the next step is to evaluate its performance. The evaluation process involves testing the model on the reserved testing subset to assess how well it generalizes to new, unseen data. This evaluation provides insights into the model’s ability to detect various types of attacks and benign traffic, offering several key outputs:

- **Save Model**: Once the model has been trained and evaluated, it is saved for future use. Saving the trained model allows it to be deployed in real-world environments where it can monitor IoT network traffic and detect potential intrusions in real-time. The saved model can also be reloaded for further training or fine-tuning if needed.

- **Generate Predictions**: The trained model generates predictions on the test data, classifying each network flow as either benign or one of the attack types. These predictions are compared against the actual labels to measure the model's accuracy and identify any misclassifications.

- **Plot Training History**: A plot of the model's training history is generated to visualize the learning process over the 50 epochs. This plot typically shows the loss and accuracy metrics for both the training and testing sets, helping to determine whether the model overfitted or underfitted during training.

8. Confusion Matrix

A confusion matrix is then generated to provide a detailed breakdown of the model's predictions. The confusion matrix is a table that compares the predicted class labels to the actual class labels, highlighting how well the model performed for each individual class. It shows true positives, true negatives, false positives, and false negatives, which are essential for understanding the strengths and weaknesses of the model in terms of classification accuracy and misclassification. For instance, the confusion matrix may reveal that the model accurately detects botnet traffic but struggles with detecting certain types of DDoS attacks.

In conclusion, this structured methodology ensures that the LSTM-based intrusion detection system is developed systematically, from data loading and preprocessing through to model training and evaluation. Each step in the process contributes to building a robust and accurate model capable of detecting and classifying a variety of cyberattacks in IoT networks.

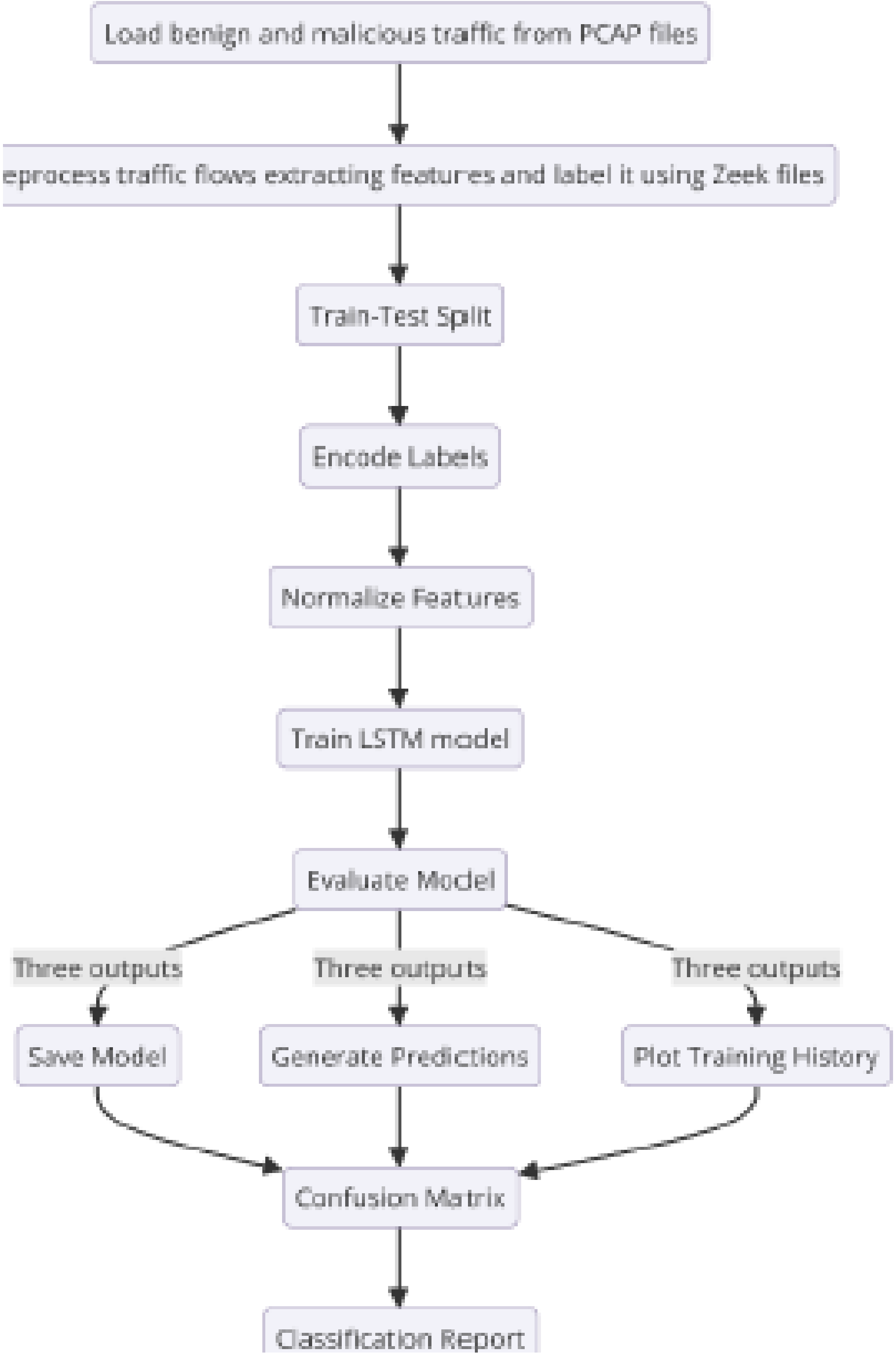


Figure III. System flowchart

8 Results

8.1 Optimal Hyperparameters

To optimize the performance of the **LSTM-based Intrusion Detection System (IDS)**, a systematic hyperparameter tuning process was conducted using the **Optuna library** [12]. Optuna is an advanced framework that automates hyperparameter optimization by efficiently exploring different configurations and identifying those that maximize model performance. Through this process, several combinations of hyperparameters were tested, and the configuration that yielded the best balance between learning efficiency and model complexity was selected.

The optimal configuration identified through the tuning process consisted of the following:

- **LSTM Units**: The model was most effective with **35 LSTM units** in each of the two LSTM layers. This number of units allowed the model to capture the temporal dependencies and patterns in the network traffic data without becoming too complex or overfitting.
- **Dropout Rate**: A **dropout rate of approximately 34.89%** (rounded to 35%) was selected. Dropout is a regularization technique that helps prevent overfitting by randomly omitting a subset of neurons during training. This dropout rate was found to be ideal in forcing the network to learn more generalized and robust features from the input data.
- **Learning Rate**: The **learning rate** was fine-tuned to **0.000741507**, which enabled the model to gradually converge on an optimal solution without oscillating or diverging. This relatively low learning rate allowed for fine adjustments to the model weights, ensuring a stable and steady learning process.

These hyperparameters proved effective in optimizing both the learning speed and the ability of the model to generalize to unseen data. The balance between the number of units, dropout rate, and learning rate ensured that the LSTM model learned efficiently while maintaining an appropriate level of complexity.

8.2 Training and Validation Performance

The training and validation performance of the LSTM model is illustrated in **Figure 4**, which shows the evolution of **training accuracy**, **validation accuracy**, **training loss**, and **validation loss** over the course of **50 epochs**. These charts provide valuable insight into the model's learning behavior and its ability to generalize from the training data to new, unseen data.

8.2.1 Training and Validation Accuracy:

The **accuracy plot** demonstrates a consistent increase in both **training accuracy** and **validation accuracy** over the training period. The training accuracy shows a steady improvement as the model gradually learns from the data, stabilizing at around **98%** by the end of the training process. This indicates that the model effectively learned the patterns and features in the network traffic data, allowing it to correctly classify a large majority of the traffic.

The **validation accuracy**, which measures the model's performance on unseen test data, closely follows the training accuracy throughout the 50 epochs. This close alignment between training and validation accuracy is a strong indicator that the model is generalizing well and not overfitting to the training data. At its peak, the validation accuracy reaches just under **98%**, further demonstrating the model's robustness in handling new network traffic flows and accurately distinguishing between benign and malicious behaviors.

8.2.2 Training and Validation Loss:

The **loss plot** shows the progression of **training loss** and **validation loss** over the same 50 epochs. The training loss decreases steadily as the model continues to learn and adjust its weights, reaching a very low value by the end of training. This reduction in loss indicates that the model is becoming increasingly confident in its predictions and is successfully minimizing the error between its predicted and actual classifications.

The **validation loss**, which measures the error on the test data, also decreases over time, closely mirroring the trend in training loss. However, slight fluctuations are observed in the validation loss, particularly around epochs 30 and 40. These fluctuations are normal and suggest that the model is responding to variations in the validation set. Notably, the validation loss peaks around epoch 41 before quickly returning to a lower value. This spike could be due to specific traffic patterns or anomalies in the validation data, but it did not significantly impact the overall performance of the model.

8.2.3 Model Robustness and Generalizability:

The close tracking of validation accuracy with training accuracy, as well as the similar trends in training and validation loss, highlights the **robustness of the LSTM model**. The model demonstrates a strong ability to generalize, meaning it can accurately classify network traffic that it has not encountered during training. The high accuracy on both training and validation data indicates that the model is not overfitting and is well-suited for deployment in real-world scenarios where it will encounter previously unseen data.

Additionally, the use of dropout layers in the model helped prevent overfitting by introducing regularization. The dropout rate of approximately 35% ensured that the model did not rely too heavily on specific features or neurons, thus encouraging it to learn more generalized patterns from the input data.

8.2.4 Summary of Performance:

- **Accuracy**: The model achieved approximately **98% accuracy** on both the training and validation datasets, demonstrating its effectiveness in distinguishing between benign traffic and a wide range of network attacks.
- **Loss**: Both training and validation loss steadily decreased, confirming that the model learned effectively over the course of 50 epochs and did not suffer from significant overfitting or underfitting.

The high performance of the model across both training and validation datasets highlights its suitability for real-time intrusion detection in IoT networks. The combination of systematic hyperparameter tuning, the use of dropout layers to prevent overfitting, and careful model evaluation ensured that the LSTM model was both accurate and generalizable.

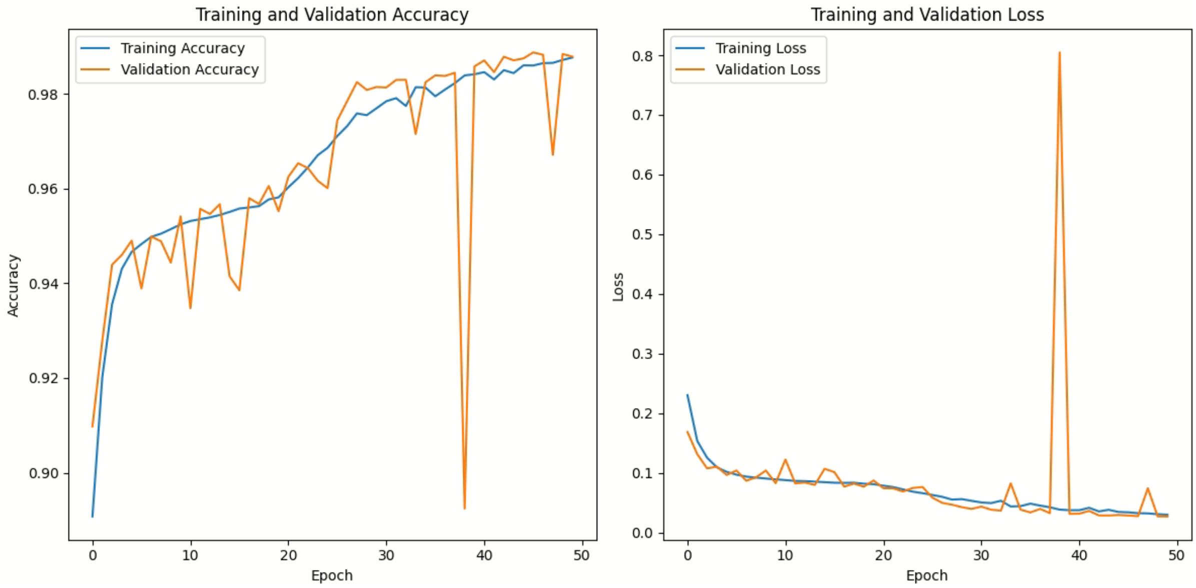


Figure IV. Loss and accuracy graphs

8.3 Test Accuracy

After completing the training and validation phases, the LSTM model was subjected to a comprehensive evaluation using the **test dataset**, which had been reserved for final performance validation. The **test set** consists of network traffic data that the model has never encountered before, making this step a critical measure of the model's true generalization capability and its effectiveness in real-world deployment scenarios.

The evaluation on the test set revealed an impressive **accuracy of 98.79%**. This high level of accuracy reflects the model's exceptional ability to correctly classify both **benign traffic** and various types of **malicious activities** present in IoT networks. A test accuracy close to 99% demonstrates that the model can reliably distinguish between normal behavior and a wide range of network attacks, including botnets, Distributed Denial of Service (DDoS) attacks, and Command and Control (C&C) traffic.

8.3.1 Significance of Test Accuracy

Achieving such a high test accuracy indicates that the LSTM model has successfully learned the temporal dependencies and patterns that distinguish different types of network traffic. It suggests that the model is capable of handling the complexity and diversity of the traffic generated by IoT devices, which often involve various communication protocols and packet structures. The following aspects of the test accuracy are particularly noteworthy:

- **Generalization Capability**: The fact that the model performed so well on the test set, which contains unseen data, signifies that it has generalized effectively from the training data. This generalization is crucial for real-world deployment, where the model will need to analyze live network traffic that is likely to differ from the data it was trained on. The ability to maintain high accuracy on previously unseen data demonstrates the robustness and adaptability of the LSTM architecture.

- **Classification of Diverse Attack Types**: The high test accuracy suggests that the model is not only proficient at identifying benign traffic but also excels at classifying different types of attacks, such as botnet-related traffic, horizontal port scans, file downloads initiated by malware, and various forms of DDoS attacks. Each of these attack types presents unique challenges, but the LSTM model's accuracy indicates that it can reliably recognize these threats based on the patterns and features present in the network flows.

- **Precision in Identifying Malicious Traffic**: In addition to the overall accuracy, the model's high test accuracy implies strong performance in identifying malicious traffic with precision. In intrusion detection systems, a high precision rate is critical, as it minimizes the number of **false positives** (benign traffic incorrectly classified as malicious) and **false negatives** (malicious traffic that goes undetected). The 98.79% accuracy achieved on the test set suggests that the model strikes an excellent balance between sensitivity to attacks and minimizing misclassifications.

8.3.2 Factors Contributing to High Test Accuracy

Several factors contributed to the LSTM model's ability to achieve such a high level of accuracy on the test set:

1. **LSTM’s Capability to Model Temporal Dependencies**: The LSTM architecture is specifically designed to capture the temporal dependencies in sequential data, such as network traffic. By analyzing the sequence of packets and the timing between them, the model can detect patterns that might indicate normal behavior or malicious activity. The use of two LSTM layers ensured that the model was able to capture both short-term and long-term dependencies within the network flows, enhancing its classification performance.

2. **Comprehensive Feature Set**: The set of **24 features** extracted during the data preprocessing phase, including packet size statistics, inter-packet arrival times, and flow durations, provided the model with a rich and diverse dataset for learning. These features captured both the high-level behavior of network flows (e.g., total bytes and packet counts) and the low-level timing characteristics (e.g., variability in packet arrival times), which are critical for identifying subtle differences between benign and malicious traffic.

3. **Dropout for Overfitting Prevention**: The use of dropout layers in the model architecture played a key role in preventing overfitting. By randomly omitting a portion of neurons during each training iteration, the dropout layers forced the model to generalize better, rather than memorizing the training data. This helped ensure that the model remained robust and adaptable when exposed to new traffic patterns in the test set.

4. **Hyperparameter Tuning with Optuna**: The fine-tuning of hyperparameters using the Optuna library contributed to the high test accuracy by optimizing key settings such as the number of LSTM units, dropout rate, and learning rate. These hyperparameters were carefully calibrated to ensure that the model could efficiently learn from the data without becoming too complex or prone to overfitting. The resulting configuration allowed the model to balance learning speed and generalization, achieving optimal performance on the test set.

8.3.3 Practical Implications of High Test Accuracy

Achieving a **98.79% accuracy** on the test set carries significant practical implications for real-world applications of this model in **IoT security**:

- **Real-Time Threat Detection**: In an operational setting, this model can be deployed to monitor live network traffic in real time, detecting and classifying potential threats with a high degree of accuracy. The model's strong generalization ability means it can reliably identify previously unseen attack patterns, providing an effective defense against emerging threats.

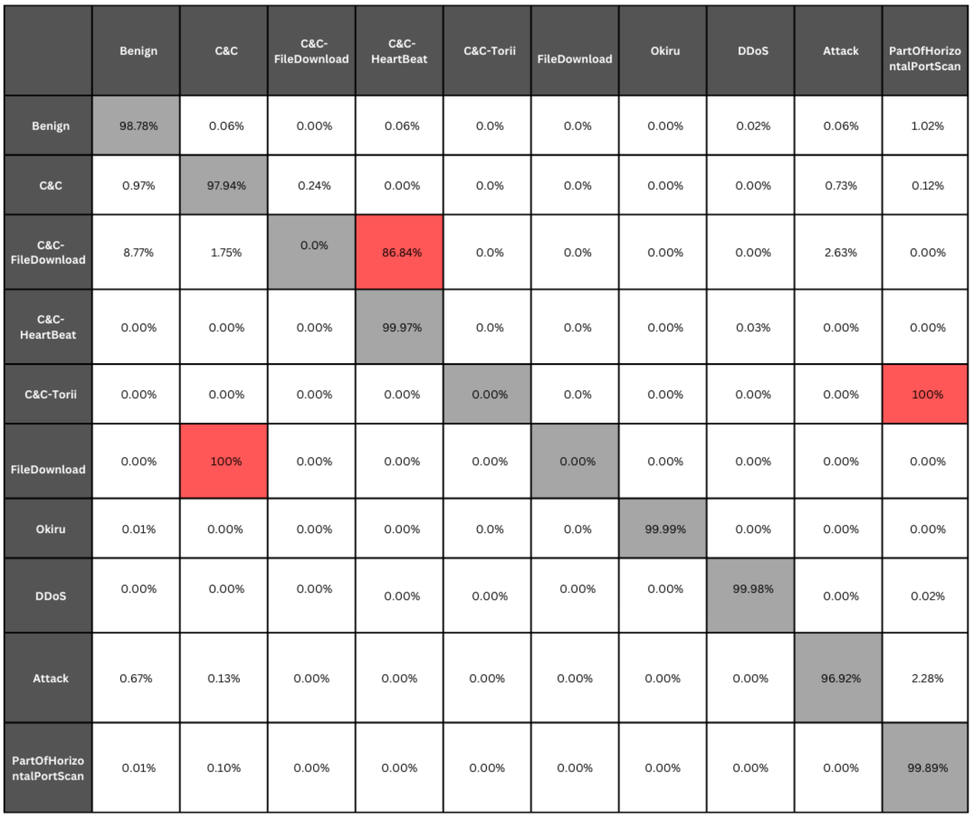
- **Reduction in False Alarms**: A high test accuracy is also indicative of the model's capacity to minimize false positives, which are a common issue in many intrusion detection systems. Reducing the number of false alarms ensures that network security teams can focus their efforts on genuine threats, improving overall efficiency and response times.

- **Scalability to Larger Networks**: The robustness of the model, as demonstrated by its test accuracy, suggests that it can be scaled to larger and more complex IoT networks without significant degradation in performance. This makes the model suitable for deployment in a wide range of IoT environments, from small home networks to large industrial IoT systems.

In conclusion, the **98.79% test accuracy** achieved by the LSTM model is a testament to its ability to accurately detect and classify a variety of network traffic behaviors, both benign and malicious. This high level of accuracy demonstrates the effectiveness of the LSTM architecture, feature engineering, and hyperparameter optimization techniques used in this project, positioning the model as a robust and scalable solution for **real-time intrusion detection** in IoT environments.

6.4 Confusion Matrix

As illustrated in **Figure V**, the LSTM model demonstrates high precision and recall across most of the classes, particularly in identifying benign traffic, Okiru malware, DDoS attacks, and horizontal port scanning traffic. The majority of diagonal cells in the matrix exhibit values close to or above 98%, indicating that the model is highly accurate in distinguishing between different types of network traffic and attack vectors.



	Benign	C&C	C&C-FileDownload	C&C-HeartBeat	C&C-Torii	FileDownload	Okiru	DDoS	Attack	PartOfHorizontalPortScan
Benign	98.78%	0.06%	0.00%	0.06%	0.0%	0.0%	0.00%	0.02%	0.06%	1.02%
C&C	0.97%	97.94%	0.24%	0.00%	0.0%	0.0%	0.00%	0.00%	0.73%	0.12%
C&C-FileDownload	8.77%	1.75%	0.0%	86.84%	0.0%	0.0%	0.00%	0.00%	2.63%	0.00%
C&C-HeartBeat	0.00%	0.00%	0.00%	99.97%	0.0%	0.0%	0.00%	0.03%	0.00%	0.00%
C&C-Torii	0.00%	0.00%	0.00%	0.00%	0.00%	0.0%	0.00%	0.00%	0.00%	100%
FileDownload	0.00%	100%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Okiru	0.01%	0.00%	0.00%	0.00%	0.0%	0.0%	99.99%	0.00%	0.00%	0.00%
DDoS	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	99.98%	0.00%	0.02%
Attack	0.67%	0.13%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	96.92%	2.28%
PartOfHorizontalPortScan	0.01%	0.10%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	99.89%

Figure IV. Confusion matrix

Key observations from the matrix:

- The model achieves **98.78% accuracy** in correctly classifying **benign** traffic, with minimal misclassifications. This is particularly important, as correctly identifying benign traffic reduces the risk of false positives, which are common in many intrusion detection systems.

- The model performs exceptionally well in detecting **C&C (Command and Control) traffic**, achieving **97.94% accuracy** in this class. This indicates that the LSTM model has effectively learned to recognize the distinctive features of C&C communication, which is crucial for identifying compromised devices in a botnet.

- **Okiru** malware is classified with near-perfect accuracy, with **99.99%** of Okiru traffic being correctly identified. This demonstrates the model's strong capability to detect this specific type of malware, which is a common threat in IoT environments.

While the overall classification performance is strong, the confusion matrix highlights a few areas where the model struggles to differentiate between certain classes, particularly those with **limited representation in the dataset**. The **red cells** in the matrix indicate errors, which are mostly concentrated in the following categories:

- **C&C File Download (8.77% misclassified as Benign)**: The model shows significant misclassification of **C&C File Download** traffic, with **8.77%** of instances being incorrectly labeled as benign. This likely stems from the **limited number of examples** in the training set for this specific class, making it harder for the model to learn the distinct features that characterize this type of malicious behavior.

- **C&C Torii (100% misclassified as PartOfHorizontalPortScan)**: The **C&C Torii** traffic is entirely misclassified as **horizontal port scanning** traffic. Again, this error can be attributed to the **small dataset size** for C&C Torii, resulting in insufficient training for the model to learn the nuances of this particular malware type.

- **File Download (misclassified as C&C File Download)**: Similarly, **File Download** traffic is also misclassified as **C&C File Download**, indicating that the model is not able to distinguish between these two closely related attack types. This could be due to the fact that both classes involve similar activities (downloading files), which may have led to overlapping features in the dataset.

These misclassifications emphasize the importance of **balanced datasets** when training machine learning models for intrusion detection. Classes that are underrepresented in the training data tend to be misclassified more often, as the model does not have enough examples to accurately learn the patterns associated with these attack types.

The confusion between certain attack classes, particularly **C&C File Download**, **C&C Torii**, and **File Download**, suggests that these classes share similar feature distributions, making it difficult for the model to distinguish between them based on the available data. This overlap can be attributed to several factors:

1. **Insufficient Data for Rare Classes**: The dataset contains fewer examples of certain attack types, such as **C&C File Download** and **C&C Torii**, which means the model has less opportunity to learn the patterns specific to these classes. Increasing the representation of these attack types in the dataset would likely improve classification accuracy for these classes.

2. **Feature Overlap**: Some attack classes may exhibit similar behavior in terms of network traffic features. For example, both **C&C File Download** and **File Download** involve the transfer of files over the network, which may result in similar flow characteristics (e.g., packet size, flow duration). Introducing more discriminative features or performing feature engineering could help the model better differentiate between these classes.

3. **Model Complexity**: The current LSTM model may benefit from further tuning or additional layers to capture the more subtle differences between certain attack classes. Adding attention mechanisms, for instance, could help the model focus on the most relevant parts of the sequence when making predictions, potentially improving its ability to distinguish between similar classes.

Despite these areas of confusion, the confusion matrix overall paints a picture of a highly effective model. The **precision** and **recall** for the most frequent classes, such as **benign traffic**, **Okiru malware**, and **DDoS attacks**, remain high, indicating that the model can reliably detect common attack patterns and distinguish them from normal network behavior.

However, to further improve the model's performance on underrepresented classes, the following steps could be taken:

- **Data Augmentation**: Increasing the number of samples for rare attack types, either by collecting more real-world data or by augmenting the existing data, could help balance the dataset and improve the model's ability to classify less frequent attacks.

- **Feature Engineering**: Exploring additional features, such as deeper packet-level attributes or temporal relationships between flows, could provide the model with more information to differentiate between closely related attack types.

- **Model Enhancements**: Incorporating more advanced architectures, such as attention mechanisms or adding additional LSTM layers, could allow the model to capture more complex patterns in the data, improving its ability to classify rare or subtle attack types.

In summary, the confusion matrix highlights the model's strong overall performance, particularly in detecting common IoT threats, while also pointing out areas for improvement, particularly in handling underrepresented attack classes. These insights are valuable for guiding future model enhancements and data collection efforts to further improve the accuracy and robustness of the LSTM-based intrusion detection system.

9 Project planification

Project Timeline: 06/11/2023 - 22/07/2024

Phase 1: Initial Research and Dataset Exploration

- **Duration**: 06/11/2023 - 18/12/2023 (6 weeks)
    - **Activities**:
        - Conduct background research on IoT security, intrusion detection systems, and LSTM neural networks.
        - Explore the IoT-23 dataset to understand its structure, features, and labeling.
        - Identify relevant literature and benchmark studies to guide model development.
    - **Milestones**:
        - Research summary and literature review completed.
        - Initial understanding of the IoT-23 dataset and preliminary ideas for feature extraction.

Phase 2: Data Preprocessing and Feature Engineering

- **Duration**: 19/12/2023 - 29/01/2024 (6 weeks)
    - **Activities**:
        - Implement feature extraction processes using Pyshark and Zeek flow files.
        - Experiment with different feature selection techniques (manual, correlation analysis, recursive feature elimination).
        - Prepare the dataset by labeling and normalizing features.
    - **Milestones**:
        - Feature extraction pipeline developed.
        - Labeled and normalized dataset ready for model input.

Phase 3: Model Development (LSTM Architecture Design)

- **Duration**: 30/01/2024 - 18/03/2024 (7 weeks)
    - **Activities**:

        - Design the LSTM architecture based on initial exploration and literature review.
        - Implement the model using Keras, with initial hyperparameters.
        - Train initial models with smaller dataset samples to verify architecture viability.
    - **Milestones**:
        - LSTM model architecture designed.
        - Preliminary model trained and evaluated for initial performance.

Phase 4: Hyperparameter Tuning and Model Optimization

- **Duration**: 19/03/2024 - 22/04/2024 (5 weeks)
    - **Activities**:
        - Use Optuna to optimize key hyperparameters (LSTM units, dropout rate, learning rate).
        - Test different configurations, recording model accuracy and performance metrics.
        - Apply regularization techniques (dropout, etc.) to prevent overfitting.
    - **Milestones**:
        - Optimized hyperparameters determined.
        - Final model architecture and configuration ready for full training.

Phase 5: Full Dataset Training and Testing

- **Duration**: 23/04/2024 - 20/05/2024 (4 weeks)
    - **Activities**:
        - Train the LSTM model using the full dataset.
        - Evaluate the model using train/test split (80/20).
        - Analyze training and validation performance (accuracy, loss).
    - **Milestones**:
        - Full dataset training completed.
        - Model performance validated with testing data.

Phase 6: Evaluation and Results Analysis

- **Duration**: 21/05/2024 - 10/06/2024 (3 weeks)
    - **Activities**:
        - Generate and analyze confusion matrices and classification reports.
        - Compare model performance with state-of-the-art methods from literature.
        - Identify strengths and weaknesses of the model, including areas for potential improvement.
    - **Milestones**:
        - Confusion matrix and classification reports generated.
        - Final results summarized, including test accuracy and error analysis.

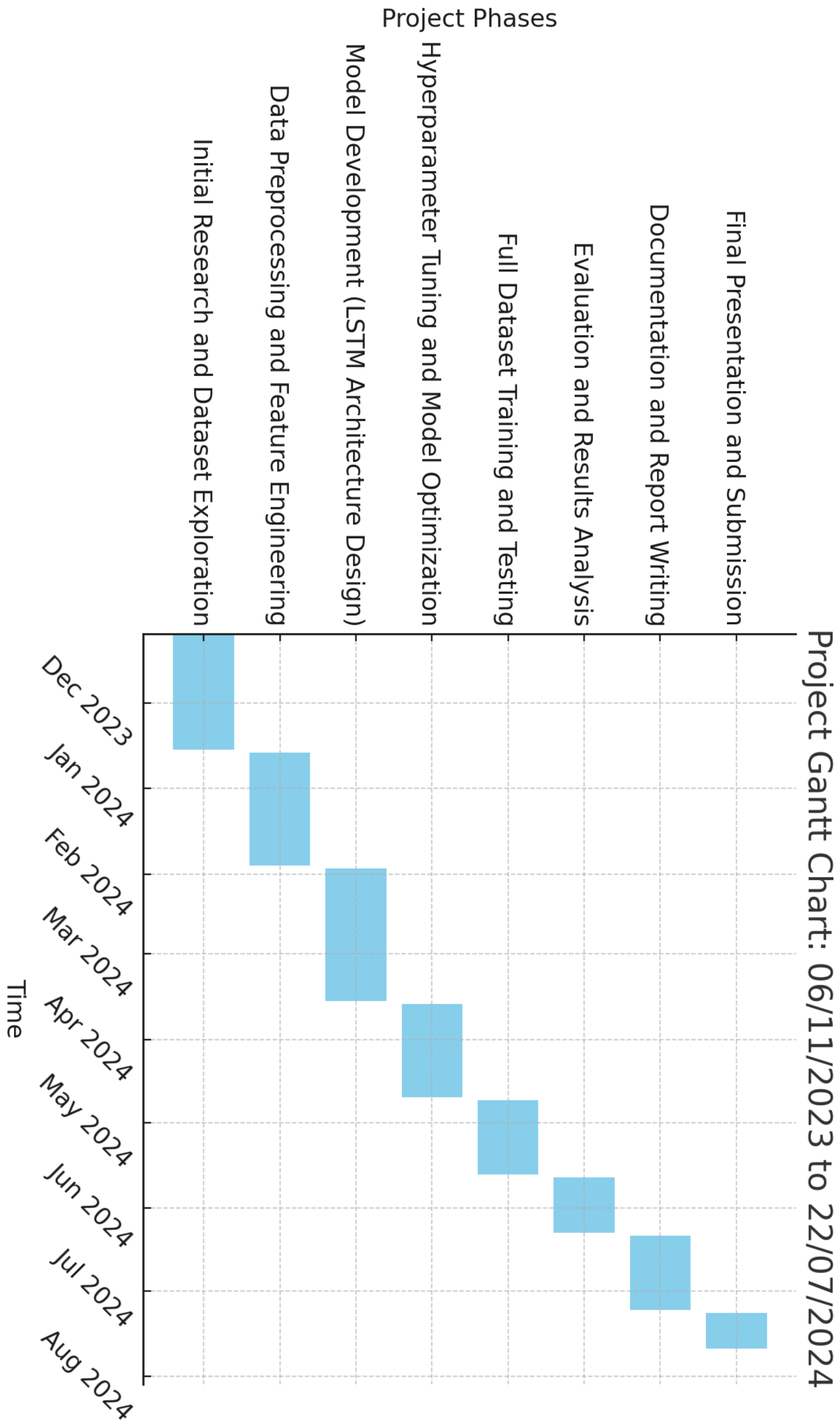
Phase 7: Documentation and Report Writing

- **Duration**: 11/06/2024 - 08/07/2024 (4 weeks)
    - **Activities**:
        - Write the research report, detailing methodology, experiments, and results.
        - Incorporate diagrams, graphs, and visualizations (e.g., confusion matrix, training history plots).
        - Revise the literature review section based on insights gained from model development.
    - **Milestones**:
        - Complete draft of the research report.
        - Peer review and revisions based on feedback.

Phase 8: Final Presentation and Submission

- **Duration**: 09/07/2024 - 22/07/2024 (2 weeks)
    - **Activities**:
        - Prepare the final presentation, including slides and supporting material.
        - Finalize the report with all necessary revisions.
        - Submit the report and project materials.
    - **Milestones**:

- Final presentation delivered.
- Report submitted for evaluation.


Project Gantt Chart: 06/11/2023 to 22/07/2024
Project Phases
Initial Research and Dataset Exploration
Data Preprocessing and Feature Engineering
Model Development (LSTM Architecture Design)
Hyperparameter Tuning and Model Optimization
Full Dataset Training and Testing
Evaluation and Results Analysis
Documentation and Report Writing
Final Presentation and Submission
Time
Dec 2023
Jan 2024
Feb 2024
Mar 2024
Apr 2024
May 2024
Jun 2024
Jul 2024
Aug 2024

10 Faced Challenges

Throughout the development of the LSTM-based Intrusion Detection System (IDS) using the **IoT-23 dataset**, several challenges emerged, requiring careful consideration and innovative solutions. These challenges spanned various aspects of the project, from feature selection and data management to hyperparameter tuning and memory optimization. The following sections highlight the key challenges faced during the project and the strategies implemented to overcome them.

10.1 Feature Selection

One of the primary challenges encountered was identifying the most relevant features from the IoT-23 dataset, which is extensive and contains a wide variety of features representing different aspects of network traffic. The dataset includes numerous attributes related to packet sizes, inter-packet arrival times, protocol types, flow statistics, and connection states, making it difficult to determine which features would provide the most value for detecting malicious activity.

Initially, manual feature selection was attempted based on domain knowledge and intuition. However, this approach proved inadequate due to the **complexity and variability** of the data. Malicious behaviors in network traffic can manifest in subtle and varied ways, making it difficult to identify the key features manually. As a result, the initial models were not as effective as anticipated, and further exploration of automated feature selection techniques became necessary.

To address this challenge, the following methods were explored:

- **Correlation Analysis**: This method was used to identify the relationships between different features and their correlation with the target variable (malicious or benign). Features with low correlation to the target or high correlation with each other were considered for removal, as they either contributed little to predictive power or introduced redundancy.

- **Recursive Feature Elimination (RFE)**: RFE is a feature selection technique that iteratively removes less important features while training a model. By using this method, the project was able to rank the importance of features and identify those that contributed the most to improving model accuracy.

- **Dimensionality Reduction (PCA)**: **Principal Component Analysis (PCA)** was considered as a dimensionality reduction technique to streamline the feature set. However, balancing dimensionality reduction while retaining essential information was a delicate process that required careful experimentation.

While PCA reduced the feature space, there was a risk of losing critical patterns in the data, so a cautious approach was taken.

Ultimately, a combination of correlation analysis and recursive feature elimination was used to refine the feature set. These techniques helped the project focus on the most relevant features while ensuring that unnecessary or redundant information was minimized.

10.2 Memory Constraints and Dataset Size

Another significant challenge arose from the large size of the IoT-23 dataset, which caused memory constraints during data preprocessing and model training. Initially, the project utilized only five PCAP files for training and evaluation. While this allowed for early experimentation, it quickly became clear that this limited dataset was insufficient for achieving the desired level of model accuracy and generalization.

As the project progressed, efforts were made to increase the dataset size by including **18 PCAP files**. This expanded the dataset considerably, providing greater diversity in traffic patterns and attack types. However, this also introduced the problem of managing the increased memory requirements. Handling large amounts of network traffic data, especially during feature extraction and model training, placed a significant burden on system memory.

To overcome these memory challenges, the following strategies were implemented:

- **Data Chunking**: Rather than loading the entire dataset into memory at once, the data was processed in chunks. This allowed for efficient memory usage by loading only small portions of the dataset at a time, ensuring that the system was not overwhelmed.

- **Efficient Data Preprocessing Pipelines**: The data preprocessing pipeline was optimized to minimize redundant calculations and streamline the feature extraction process. This reduced the time and memory required to transform raw network traffic into a usable format for the LSTM model.

- **Data Batching and On-the-Fly Augmentation**: During model training, data batching was used to divide the dataset into smaller, manageable batches. Additionally, on-the-fly data augmentation was employed to dynamically modify the data during training, further reducing memory overhead.

- **Cloud-Based and High-Performance Computing Resources**: Cloud-based resources and high-performance computing environments were explored to support the project’s need for greater computational power and memory. These environments provided the flexibility to handle larger datasets and more complex models without being constrained by local system limitations.

Despite these improvements, memory management remained a critical concern throughout the project. However, the combination of chunking, batching, and cloud resources helped mitigate these issues and allowed the project to scale effectively.

10.3 Hyperparameter Optimization

Determining the optimal hyperparameters for the LSTM model was another significant challenge, as incorrect settings could lead to overfitting or underfitting. Hyperparameters such as the number of LSTM units, dropout rate, and learning rate play a critical role in controlling the model's complexity and its ability to generalize to new data.

Initial models were prone to overfitting when the number of LSTM units was too high or when the dropout rate was too low, causing the model to memorize the training data instead of learning generalized patterns. Conversely, settings that were too conservative led to underfitting, where the model failed to capture the complexity of the network traffic, resulting in poor performance on both the training and test sets.

To address this challenge, the project employed several techniques to optimize hyperparameter settings:

- **Grid Search and Random Search**: Initially, manual grid search and random search techniques were used to explore the hyperparameter space. While these methods provided some improvements, they were time-consuming and computationally expensive, as they required testing multiple combinations of hyperparameters.

- **Automated Hyperparameter Optimization (Optuna)**: To streamline the process, the project implemented an automated hyperparameter optimization process using the Optuna library. Optuna is a powerful tool that uses a trial-based approach to systematically explore different combinations of hyperparameters, identifying the configuration that yields the highest performance. This method significantly reduced the time spent on tuning and led to better overall results.

Although the automated hyperparameter optimization was highly effective in finding the best settings, it also introduced additional computational challenges. The optimization process required numerous trials, each involving full model training and evaluation, which increased the overall computational cost and complexity of the project. Despite this, the use of Optuna helped achieve higher accuracy and stability in the model, making the trade-off worthwhile.

The challenges faced during the development of the LSTM-based IDS were multifaceted, spanning feature selection, memory management, and hyperparameter optimization. Each challenge required creative solutions, from advanced feature selection techniques and efficient data preprocessing to leveraging cloud resources and automated hyperparameter tuning. By overcoming these hurdles, the project was able to develop a highly effective and scalable model for detecting intrusions in IoT networks, ultimately contributing to the robustness and accuracy of the final system.

11 Conclusion

The LSTM model developed in this study demonstrated high effectiveness in classifying network attacks within IoT environments, achieving a test accuracy of 98.79%. This impressive level of accuracy reflects the model's capacity to accurately differentiate between benign and malicious network behavior across a wide range of attack classes. The detailed confusion matrix generated during the evaluation process further underscores the model's robustness and precision, revealing its capability to correctly classify complex patterns of network activity that are characteristic of IoT environments.

The success of the LSTM model can be attributed to the careful selection and optimization of its hyperparameters, including the number of LSTM units, the dropout rate, and the learning rate. Through an extensive hyperparameter optimization process, these elements were finely tuned to enhance the model's performance. The optimal combination of these parameters allowed the model to effectively capture the temporal dependencies, and intricate patterns present in network traffic data, leading to superior classification results.

The study's findings suggest that deep learning, and LSTM networks in particular, are highly suitable for addressing the complex and dynamic nature of network traffic associated with IoT devices. IoT networks are characterized by their heterogeneity and high volume of data, which can vary significantly over time. Traditional machine learning models often struggle to adapt to these complexities, whereas LSTM networks excel due to their ability to process sequential data and maintain information over long periods.

Moreover, the LSTM model's architecture enables it to learn not only from individual packet features but also from the sequences and contexts in which these packets occur. This sequential learning capability is crucial for identifying subtle patterns of malicious activity that might be overlooked by models that do not account for temporal relationships. By analyzing sequences of network traffic, the LSTM model can detect sophisticated attack techniques, such as those involving multi-stage intrusions or coordinated botnet activities.

The practical implications of this study are significant, as IoT devices continue to proliferate across various sectors, including smart homes, healthcare, industrial automation, and transportation. Each of these domains presents unique security challenges, making the need for effective intrusion detection solutions increasingly critical. The LSTM model offers a scalable and adaptive solution that can be integrated into existing security frameworks, providing real-time threat detection and response capabilities.

Additionally, the model's adaptability to diverse IoT architectures will be assessed. This includes evaluating its performance across different device types, communication protocols, and network topologies. By ensuring the model's generalizability, its deployment can be extended to a wide array of IoT environments, each with its unique security requirements and challenges.

Furthermore, the potential for combining LSTM networks with other advanced machine learning techniques, such as ensemble methods or hybrid models, will be investigated. Such combinations may enhance detection accuracy and resilience against novel attack vectors, providing a comprehensive and robust defense mechanism for IoT networks.

In conclusion, the LSTM model developed in this study represents a significant advancement in IoT security, demonstrating the efficacy of deep learning in tackling the challenges posed by complex network traffic. As IoT devices become increasingly integrated into the fabric of modern life, the need for reliable and adaptive security solutions will only continue to grow. This study provides a promising foundation for further exploration and innovation in the field of IoT cybersecurity.

12 Future work

The successful development and application of the **LSTM-based Intrusion Detection System (IDS)** in this project open up a wide array of possibilities for future research, practical applications, and advancements in IoT security. Although the current model demonstrates strong performance in detecting and classifying network attacks within the IoT-23 dataset, there are several areas where further improvements and enhancements can be explored to refine its capabilities and expand its use cases.

12.1 Real-Time Intrusion Detection

A key avenue for future work involves deploying the LSTM model as part of a real-time Intrusion Detection System (IDS) in **IoT networks**. While the current model processes historical network data from the IoT-23 dataset, transitioning to a real-time application requires additional developments to handle **live network traffic**. Real-time IDS systems must continuously monitor traffic and provide immediate responses to detected threats, making low-latency detection critical.

To enable real-time functionality, the following challenges must be addressed:

- **Reducing Detection Latency**: In a real-time environment, the model must quickly process incoming network flows and make instantaneous decisions about potential threats. This could involve optimizing the model architecture, such as reducing the complexity of certain layers or adjusting the number of LSTM units to improve processing speed without sacrificing accuracy.

- **Leveraging Efficient Computational Resources**: Utilizing **Graphics Processing Units (GPUs)** or **Field-Programmable Gate Arrays (FPGAs)** could significantly accelerate the model's inference time. These hardware accelerators are well-suited for parallel processing and would enable the system to analyze large volumes of network traffic in real-time.

- **Integrating with Real-Time Monitoring Tools**: Integrating the model with **network monitoring frameworks** such as **Zeek**, **Snort**, or **Suricata** could facilitate the collection of live traffic and immediate threat analysis. By coupling the LSTM model with these systems, the IDS could act as an active defense mechanism within IoT networks, triggering alerts or countermeasures when malicious activity is detected.

Developing a **real-time IDS** based on the LSTM model will contribute to enhanced **operational security** by enabling proactive threat detection and rapid responses to cyberattacks, thereby minimizing damage and protecting critical IoT infrastructures.

12.2 Model Optimization and Scalability

As the LSTM model transitions to real-time applications, there will be a need to further optimize its performance in terms of both **speed** and **scalability**. This entails fine-tuning various aspects of the model to ensure it can handle large-scale IoT networks, which may consist of thousands or even millions of connected devices.

- **Scalability Across Different Networks**: Future work should test the model's performance in different network environments and under varying conditions, such as fluctuating traffic loads or different communication protocols. By applying the model to diverse datasets, researchers can evaluate its **generalizability** and ensure that it remains effective across a broad range of IoT deployments. Such testing may also reveal the need to adapt the model to specific types of IoT networks, such as **industrial IoT** or **smart cities**, where the types of devices and traffic patterns differ from typical consumer IoT devices.

- **Optimizing for Resource-Constrained Environments**: Many IoT devices operate with limited **computational resources**, memory, and power. Future research could explore **lightweight versions** of the LSTM model that are specifically optimized for deployment on these resource-constrained devices. Techniques such as **model pruning**, **quantization**, and **knowledge distillation** can help reduce the model’s size and computational overhead, making it more feasible for deployment on smaller devices.

- **Distributed Detection Systems**: Another approach to scalability involves developing **distributed IDS systems**, where the LSTM model is deployed across multiple nodes in a network. Each node would monitor traffic in its local area, and the results could be aggregated at a central node for comprehensive network analysis. This would reduce the load on individual devices and improve the system’s ability to scale across large IoT networks.

12.3 Exploring Hybrid and Ensemble Learning Approaches

While the LSTM model shows strong performance in identifying attacks, future work could explore **hybrid approaches** that combine LSTM networks with other **machine learning** or **statistical techniques**. Such approaches could provide greater robustness, improve classification accuracy, and enhance the system’s ability to detect more **sophisticated attack patterns**.

- **Hybrid Models**: A potential direction is to combine the LSTM model with **Convolutional Neural Networks (CNNs)**, which are excellent at detecting spatial patterns in data. While LSTMs excel at capturing temporal relationships in network traffic, CNNs can identify important spatial features (such as packet signatures) that might otherwise be missed. Combining the two architectures could lead to more comprehensive threat detection.

- **Ensemble Methods**: Another promising area of research involves using **ensemble learning techniques** such as **Random Forests**, **Gradient Boosting**, or **XGBoost** in conjunction with the LSTM model. By creating ensembles of models, each specializing in different aspects of network traffic analysis, the system could leverage the strengths of each model to improve overall accuracy and robustness. Ensemble methods are particularly effective in reducing the variance of predictions and mitigating errors from individual models.

- **Adversarial Training**: To increase the model's resilience to evolving threats, future work could explore **adversarial training**, where the LSTM model is trained on adversarial examples designed to confuse it. This process could harden the model against **evasion attacks**, where attackers intentionally modify their traffic patterns to bypass detection.

12.4 Integration with Additional Data Sources

Another area of future research involves expanding the data sources used by the IDS model to enhance its detection capabilities. Currently, the model focuses on network flow features derived from packet captures, but additional contextual data could provide deeper insights into potential attacks.

- **Incorporating Device Metadata**: Integrating **IoT device metadata**, such as device type, firmware version, or historical behavior patterns, could improve the model's ability to detect specific attacks that target device vulnerabilities. By analyzing how different devices behave in the network, the model could detect anomalies that go beyond just packet-level features.
- **Fusion with External Threat Intelligence**: The model could also be integrated with external **threat intelligence feeds** that provide real-time information on new malware strains, known malicious IP addresses, or attack techniques. By combining real-time intelligence with the network traffic analysis performed by the LSTM model, the IDS could become more proactive in detecting emerging threats before they fully manifest.

12.5 Deployment in Real-World IoT Systems

Collaborating with industry partners, **IoT device manufacturers**, and **network operators** presents a valuable opportunity to test and refine the LSTM model in real-world environments. Deploying the IDS in actual IoT systems will help validate its effectiveness under practical conditions, where challenges such as network noise, variable traffic patterns, and device heterogeneity are more prominent.

- **Field Testing in IoT Ecosystems**: Conducting field tests in diverse IoT ecosystems, such as **smart homes**, **industrial IoT systems**, or **smart cities**, would provide insights into how the LSTM-based IDS performs in realistic scenarios. Collaborating with IoT platform vendors and network security teams could ensure that the model is fine-tuned to meet the specific security requirements of these environments.
- **Customization for Different IoT Platforms**: Real-world deployment would also involve customizing the IDS to integrate seamlessly with various IoT platforms. Each IoT system may have its own unique communication protocols and network architectures,

so the model may need to be adapted to ensure compatibility and efficient deployment.

13 References

[1] IoT Security Foundation, “IoT botnets might be the cybersecurity industry’s next big worry”, [Online]. Available: https://iotsecurityfoundation.org/iot-botnets-might-be-the-cybersecurity-industrys-next-big-worry/

[2] IEEE Xplore Digital Library, “Cybersecurity Fortification in Edge Computing”, [Online], Available: https://innovate.ieee.org/innovation-spotlight/cybersecurity-fortification-in-edge-computing/

[3] Sebastian Garcia, Agustin Parmisano, & Maria Jose Erquiaga. (2020). IoT-23: A labeled dataset with malicious and benign IoT network traffic (Version 1.0.0) [Data set]. Zenodo. http://doi.org/10.5281/zenodo.4743746

[4] M. Gromov, D. Arnold and J. Saniie, "Edge Computing for Real Time Botnet Propagation Detection," 2022 IEEE International Conference and Expo on Real Time Communications at IIT (RTC), Chicago, IL, USA, 2022, pp. 13-16, doi: 10.1109/RTC56148.2022.9945060.

[5] S. Kalenowski, D. Arnold, M. Gromov and J. Saniie, "Heterogeneity Tolerance in IoT Botnet Attack Classification," *2023 IEEE International Conference on Electro Information Technology (eIT)*, Romeoville, IL, USA, 2023, pp. 353-356, doi: 10.1109/eIT57321.2023.10187264.

[6] M. Gromov, D. Arnold and J. Saniie, "Utilizing Computer Vision Algorithms to Detect and Classify Cyberattacks in IoT Environments in Real-Time," 2023 IEEE International Conference on Electro Information Technology (eIT), Romeoville, IL, USA, 2023, pp. 300-303, doi: 10.1109/eIT57321.2023.10187263.

[7] M. Gromov, D. Arnold and J. Saniie, "Tackling Multiple Security Threats in an IoT Environment," 2022 IEEE International Conference on Electro Information Technology (eIT), Mankato, MN, USA, 2022, pp. 290-295, doi: 10.1109/eIT53891.2022.9814003.

[8] S. Racherla, P. Sripathi, N. Faruqui, M. Alamgir Kabir, M. Whaiduzzaman and S. Aziz Shah, "Deep-IDS: A Real-Time Intrusion Detector for IoT Nodes Using Deep Learning," in IEEE Access, vol. 12, pp. 63584-63597, 2024, doi: 10.1109/ACCESS.2024.3396461.

[8] Sayegh, Hussein Ridha, Wang Dong, and Ali Mansour Al-madani. 2024. "Enhanced Intrusion Detection with LSTM-Based Model, Feature Selection, and SMOTE for Imbalanced Data" *Applied Sciences* 14, no. 2: 479. https://doi.org/10.3390/app14020479

[9] Pyshark library. [Online]: https://github.com/KimiNewt/pyshark/

[11] Keras library. https://keras.io/

[12] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19). Association for Computing Machinery, New York, NY, USA, 2623–2631.

14 Anex

14.1 DataPreprocessing.py

```
import os
import pandas as pd
import numpy as np
import pyshark
from collections import defaultdict, Counter
from concurrent.futures import ProcessPoolExecutor


# Define the features to extract
FIELDNAMES = ['source_ip', 'source_port', 'destination_ip', 'destination_port', 'protocol',
            'application_protocol', 'packet_count', 'tot_fwd_pkts', 'tot_bwd_pkts',
'byte_count',
            'orig_bytes', 'resp_bytes', 'packet_size_mean', 'packet_size_std',
'packet_size_median',
            'packet_size_min', 'packet_size_max', 'inter_packet_arrival_mean',
'inter_packet_arrival_std',
            'inter_packet_arrival_median', 'inter_packet_arrival_min',
'inter_packet_arrival_max',
            'flow_duration', 'conn_state', 'label']


# Protocol mapping
PROTOCOL_MAP = {'6': 'tcp', '17': 'udp', '1': 'icmp'}


# Label mapping
label_map = {
    '-   benign   -': 1,
    '(empty)   Benign   -': 1,
    '-   Benign   -': 1,
    '(empty)   Malicious   C&C': 2,
    '-   Malicious   C&C': 2,
    '-   Malicious   C&C-FileDownload': 3,
    '-   Malicious   C&C-HeartBeat-FileDownload': 4,
    '(empty)   Malicious   C&C-HeartBeat': 4,
    '-   Malicious   C&C-HeartBeat': 4,
    '-   Malicious   C&C-Mirai': 5,
    '-   Malicious   C&C-Torii': 6,
    '-   Malicious   FileDownload': 7,
    '(empty)   Malicious   Okiru': 8,
    '-   Malicious   Okiru': 8,
    '-   Malicious   DDoS': 9,
    '-   Malicious   Attack': 10,
    '(empty)   Malicious   Attack': 10,
    '(empty)   Malicious   PartOfAHorizontalPortScan': 11,
    '-   Malicious   PartOfAHorizontalPortScan': 11,
}


def extract_features_from_pcap(pcap_file, max_packets=300000):
    """Extract features from the PCAP file using PyShark."""
    try:
        print(f"Processing PCAP file: {pcap_file}")
        flows = defaultdict(lambda: {'timestamps': [], 'sizes': [], 'forward': 0, 'backward': 0, 'total_size': 0, 'orig_bytes': 0, 'resp_bytes': 0, 'protocols': set(), 'conn_states': set()})
        cap = pyshark.FileCapture(pcap_file, only_summaries=False)


        packet_count = 0
        for packet in cap:
            if packet_count >= max_packets:
                break
            if 'IP' in packet:
                src_ip = packet.ip.src
                dst_ip = packet.ip.dst
                transport_layer = packet.transport_layer
                if transport_layer and hasattr(packet, transport_layer.lower()):
                    protocol = PROTOCOL_MAP.get(packet[transport_layer].layer_name.lower(),
packet[transport_layer].layer_name.lower())
                    src_port = int(getattr(packet[transport_layer.lower()], 'srcport', 0))
                    dst_port = int(getattr(packet[transport_layer.lower()], 'dstport', 0))
                    flow_key = (src_ip, src_port, dst_ip, dst_port, protocol)

                    flows[flow_key]['timestamps'].append(float(packet.sniff_time.timestamp()))
                    flows[flow_key]['sizes'].append(int(packet.length))
                    flows[flow_key]['total_size'] += int(packet.length)
                    flows[flow_key]['protocols'].add(packet.highest_layer.lower())
                    if packet.ip.src == src_ip:
                        flows[flow_key]['forward'] += 1
                        flows[flow_key]['orig_bytes'] += int(packet.length)
                    else:
                        flows[flow_key]['backward'] += 1
                        flows[flow_key]['resp_bytes'] += int(packet.length)

                    # Extract connection state
                    if protocol == 'tcp':
                        tcp_flags = packet.tcp.flags
                        if tcp_flags == '0x0002':  # SYN
                            flows[flow_key]['conn_states'].add('S0')
                        elif tcp_flags == '0x0012':  # SYN-ACK
                            flows[flow_key]['conn_states'].add('S1')
                        elif tcp_flags == '0x0010':  # ACK
                            flows[flow_key]['conn_states'].add('S2')
                        elif tcp_flags == '0x0011':  # FIN-ACK
                            flows[flow_key]['conn_states'].add('S3')
                        elif tcp_flags == '0x0014':  # RST
                            flows[flow_key]['conn_states'].add('R')
                        elif tcp_flags == '0x0004':  # RST without SYN
                            flows[flow_key]['conn_states'].add('RSTR')
                        elif tcp_flags == '0x0018':  # ACK + PSH
                            flows[flow_key]['conn_states'].add('SH')
                        else:
                            flows[flow_key]['conn_states'].add('OTH')  # Other TCP states
                    elif protocol == 'udp':
                        flows[flow_key]['conn_states'].add('SF')  # Assume UDP flows are "SYN-FIN"
                    elif protocol == 'icmp':
                        flows[flow_key]['conn_states'].add('SF')  # Assume ICMP flows are "SYN-FIN"
                    else:
                        flows[flow_key]['conn_states'].add('OTH')  # Other protocols

                    packet_count += 1

        cap.close()
        print(f"Extracted {len(flows)} flows from {pcap_file}")
```

```
        return flows
    except Exception as e:
        print(f"Error processing PCAP file {pcap_file}: {e}")
        return {}


def calculate_flow_statistics(flows):
    """Calculate flow statistics from extracted features."""
    flow_data = []
    for key, flow in flows.items():
        src_ip, src_port, dst_ip, dst_port, protocol = key
        packet_count = len(flow['timestamps'])
        tot_fwd_pkts = flow['forward']
        tot_bwd_pkts = flow['backward']
        byte_count = flow['total_size']
        orig_bytes = flow['orig_bytes']
        resp_bytes = flow['resp_bytes']
        packet_size_mean = np.mean(flow['sizes']) if flow['sizes'] else 0
        packet_size_std = np.std(flow['sizes']) if flow['sizes'] else 0
        packet_size_median = np.median(flow['sizes']) if flow['sizes'] else 0
        packet_size_min = np.min(flow['sizes']) if flow['sizes'] else 0
        packet_size_max = np.max(flow['sizes']) if flow['sizes'] else 0
        inter_packet_times = np.diff(flow['timestamps'])
        inter_packet_arrival_mean = np.mean(inter_packet_times) if len(inter_packet_times) > 0 else 0
        inter_packet_arrival_std = np.std(inter_packet_times) if len(inter_packet_times) > 0 else 0
        inter_packet_arrival_median = np.median(inter_packet_times) if len(inter_packet_times) > 0 else 0
        inter_packet_arrival_min = np.min(inter_packet_times) if len(inter_packet_times) > 0
else 0

        inter_packet_arrival_max = np.max(inter_packet_times) if len(inter_packet_times) >
0 else 0

        flow_duration = flow['timestamps'][-1] - flow['timestamps'][0] if
len(flow['timestamps']) > 1 else 0



        application_protocol = Counter(list(flow['protocols'])).most_common(1)[0][0] if
flow['protocols'] else 'N/A'

        conn_state = Counter(list(flow['conn_states'])).most_common(1)[0][0] if
flow['conn_states'] else 'N/A'



        flow_data.append([src_ip, src_port, dst_ip, dst_port, protocol, application_protocol,
packet_count, tot_fwd_pkts, tot_bwd_pkts,

                          byte_count, orig_bytes, resp_bytes, packet_size_mean, packet_size_std,
packet_size_median,

                          packet_size_min, packet_size_max, inter_packet_arrival_mean,
inter_packet_arrival_std,

                          inter_packet_arrival_median, inter_packet_arrival_min,
inter_packet_arrival_max, flow_duration, conn_state])



    df_flows = pd.DataFrame(flow_data, columns=FIELDNAMES[:-1])

    df_flows['label'] = 'N/A'



    return df_flows



def read_zeek_file(zeek_file, max_lines=300000):

    """Read and process the Zeek analysis file."""

    try:

        print(f"Processing Zeek file: {zeek_file}")

        records = []
        with open(zeek_file, 'r') as file:
            for line in file:
                if len(records) >= max_lines:
                    break
                if line.startswith("#") or not line.strip():
                    continue
                fields = line.strip().split("\t")
                if len(fields) >= 21:
                    source_ip = fields[2]
                    source_port = int(fields[3])
                    destination_ip = fields[4]
                    destination_port = int(fields[5])
                    protocol = fields[6].lower()
                    raw_label = fields[20].strip()
                    label = label_map.get(raw_label, 0)  # Default to 0 if not found
                    records.append([source_ip, source_port, destination_ip, destination_port, protocol, label])

        zeek_df = pd.DataFrame(records, columns=['source_ip', 'source_port', 'destination_ip', 'destination_port', 'protocol', 'label'])
        print(f"Zeek DataFrame:\n{zeek_df.head()}")
        return zeek_df
    except Exception as e:
        print(f"Error processing Zeek file {zeek_file}: {e}")
        return pd.DataFrame(columns=['source_ip', 'source_port', 'destination_ip', 'destination_port', 'protocol', 'label'])

def merge_flows_with_zeek(df_flows, zeek_df):
    """Merge PCAP extracted features with Zeek analysis."""
    try:
        # Ensure consistent data types for merging
        for col in ['source_ip', 'destination_ip', 'source_port', 'destination_port', 'protocol']:
            df_flows[col] = df_flows[col].astype(str)
            zeek_df[col] = zeek_df[col].astype(str)

        merge_columns = ['source_ip', 'destination_ip', 'source_port', 'destination_port', 'protocol']

        # Debug: Print sample keys to verify before merging
        print("Sample keys from df_flows:")
        print(df_flows[merge_columns].head())
        print("Sample keys from zeek_df:")
        print(zeek_df[merge_columns].head())

        combined_df = pd.merge(df_flows, zeek_df, on=merge_columns, suffixes=('', '_zeek'))

        # Debug: Print combined DataFrame before dropping columns
        print(f"Combined DataFrame before dropping columns:\n{combined_df.head()}")

        # Drop the original 'label' column and rename 'label_zeek' to 'label'
        combined_df.drop(columns=['label'], inplace=True)
        combined_df.rename(columns={'label_zeek': 'label'}, inplace=True)

        # Remove duplicate flows
        combined_df.drop_duplicates(subset=merge_columns, inplace=True)
        print(f"Combined DataFrame after merging:\n{combined_df.head()}")
        return combined_df
    except Exception as e:
        print(f"Error merging dataframes: {e}")
        return df_flows


def process_file(pcap_file, zeek_file, max_packets=300000, max_lines=300000):
    """Process a single pair of PCAP and Zeek files."""
    try:
        flows = extract_features_from_pcap(pcap_file, max_packets)
        if not flows:
            print(f"No flows extracted from {pcap_file}")
            return pd.DataFrame()
        df_flows = calculate_flow_statistics(flows)
        zeek_df = read_zeek_file(zeek_file, max_lines)
        df_flows_with_labels = merge_flows_with_zeek(df_flows, zeek_df)
        return df_flows_with_labels
    except Exception as e:
        print(f"Error processing files {pcap_file} and {zeek_file}: {e}")
        return pd.DataFrame()


def process_folder(pcap_folder, zeek_folder, output_file, max_packets=300000,
max_lines=300000, n_workers=2):
    """Process all PCAP and Zeek files in a folder and save the combined dataset."""
    all_flows = []


    pcap_files = [f for f in os.listdir(pcap_folder) if f.endswith(".pcap")]
    zeek_files = [f.replace(".pcap", ".labeled") for f in pcap_files]


    with ProcessPoolExecutor(max_workers=n_workers) as executor:
        futures = []
        for pcap_file, zeek_file in zip(pcap_files, zeek_files):
            pcap_path = os.path.join(pcap_folder, pcap_file)
            zeek_path = os.path.join(zeek_folder, zeek_file)
            if os.path.exists(zeek_path):
                futures.append(executor.submit(process_file, pcap_path, zeek_path,
max_packets, max_lines))
            else:
                print(f"Corresponding Zeek file not found for {pcap_file}")


        for future in futures:
            try:
                result = future.result()
                if not result.empty:
                    all_flows.append(result)
            except Exception as e:
                print(f"Error processing file pair: {e}")


    if all_flows:
        combined_df = pd.concat(all_flows, ignore_index=True)
        combined_df.to_csv(output_file, index=False)
        print(f"Combined dataset saved to {output_file}")
        print(f"Total rows in combined dataset: {len(combined_df)}")
        print(f"Label counts:\n{combined_df['label'].value_counts()}")
    else:
        print("No data processed.")


# Paths to the PCAP and Zeek folders
pcap_folder = '/home/VICOMTECH/ialcorta/iot-23/captures'
zeek_folder = '/home/VICOMTECH/ialcorta/iot-23/labels'
output_file = 'attack_dataset.csv'


# Run the processing function
process_folder(pcap_folder, zeek_folder, output_file, max_packets=300000, max_lines=300000, n_workers=2)
```

11.2. LSTM.py

```
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, LSTM, Dropout, Input
from tensorflow.keras.utils import to_categorical
import optuna
from optuna.integration import TFKerasPruningCallback
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import confusion_matrix, classification_report
import ipaddress
import numpy as np
import logging


# Enable logging for Optuna
```

```
optuna.logging.set_verbosity(optuna.logging.INFO)


def preprocess_data(df):
    # Convert IP addresses to integers using .loc to avoid SettingWithCopyWarning
    df.loc[:, 'source_ip'] = df['source_ip'].apply(lambda x: int(ipaddress.ip_address(x)))
    df.loc[:, 'destination_ip'] = df['destination_ip'].apply(lambda x: int(ipaddress.ip_address(x)))

    # Encode application protocols using .loc to avoid SettingWithCopyWarning
    if 'application_protocol' in df.columns:
        app_protocol_encoder = LabelEncoder()
        df.loc[:, 'application_protocol'] =
app_protocol_encoder.fit_transform(df['application_protocol'].astype(str))

    # Encode connection state using .loc to avoid SettingWithCopyWarning
    if 'conn_state' in df.columns:
        conn_state_encoder = LabelEncoder()
        df.loc[:, 'conn_state'] = conn_state_encoder.fit_transform(df['conn_state'].astype(str))

    # Encode protocol using .loc to avoid SettingWithCopyWarning
    protocol_encoder = LabelEncoder()
    df.loc[:, 'protocol'] = protocol_encoder.fit_transform(df['protocol'].astype(str))

    # Ensure label is categorical using .loc to avoid SettingWithCopyWarning
    df.loc[:, 'label'] = df['label'].astype(int)

    return df


# Load your data
```

```
data = pd.read_csv('attack_dataset.csv')

# Count the occurrences of each label
label_counts = data['label'].value_counts()

# Print the counts
print("Label counts before filtering:")
print(label_counts)

# Plot the distribution of labels
plt.figure(figsize=(10, 6))
label_counts.plot(kind='bar')
plt.title('Distribution of Labels in the Dataset')
plt.xlabel('Labels')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()

# Set a threshold for minimum number of instances per label
threshold = 2

# Filter out labels with fewer than the threshold
filtered_data1 = data[data['label'].isin(label_counts[label_counts >= threshold].index)]
# Filter out rows with label 0
filtered_data = filtered_data1[filtered_data1['label'] != 0]

# Print the updated counts
```

```
updated_label_counts = filtered_data['label'].value_counts()
print("Label counts after filtering:")
print(updated_label_counts)

# Apply preprocessing
preprocessed_data = preprocess_data(filtered_data)

# Remove any rows with remaining non-numeric placeholders
preprocessed_data.replace('-', np.nan, inplace=True)
preprocessed_data.dropna(inplace=True)

# Select numeric features for the model
features = preprocessed_data.drop(columns=['label'])
labels = preprocessed_data['label']

# Normalize features
scaler = StandardScaler()
features = scaler.fit_transform(features)

# Print shapes of features and labels
print(f"Features shape: {features.shape}")
print(f"Labels shape: {labels.shape}")

# Save the preprocessed features to a new CSV
preprocessed_df = pd.DataFrame(features, columns=preprocessed_data.columns.drop('label'))
preprocessed_df['label'] = labels.values
preprocessed_df.to_csv('preprocessed_features.csv', index=False)
```

```
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42, stratify=labels)


# Encode the labels
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)


# One-hot encode the labels
y_train_categorical = to_categorical(y_train_encoded, num_classes=len(label_encoder.classes_))
y_test_categorical = to_categorical(y_test_encoded, num_classes=len(label_encoder.classes_))


# Print the size of the training and testing sets
print(f"Training set size: {X_train.shape[0]} samples")
print(f"Testing set size: {X_test.shape[0]} samples")


# Print shapes of training and testing sets
print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train_categorical.shape}")
print(f"y_test shape: {y_test_categorical.shape}")


# Define your LSTM model
def create_model(num_units=50, dropout_rate=0.2, learning_rate=0.001):
    model = Sequential()
    model.add(Input(shape=(X_train.shape[1], 1)))
    model.add(LSTM(num_units, return_sequences=True))
    model.add(Dropout(dropout_rate))
    model.add(LSTM(num_units))
    model.add(Dropout(dropout_rate))
    model.add(Dense(len(label_encoder.classes_), activation='softmax'))

    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# Define the objective function for Optuna
def objective(trial):
    num_units = trial.suggest_int('num_units', 32, 128)
    dropout_rate = trial.suggest_float('dropout_rate', 0.2, 0.5)
    learning_rate = trial.suggest_float('learning_rate', 1e-4, 1e-2, log=True)

    print(f"Trial {trial.number}: num_units={num_units}, dropout_rate={dropout_rate},
learning_rate={learning_rate}")

    model = create_model(num_units, dropout_rate, learning_rate)

    # Use a small number of epochs for quick testing
    history = model.fit(X_train, y_train_categorical,
                        validation_split=0.2,
                        epochs=10,
                        callbacks=[TFKerasPruningCallback(trial, 'val_accuracy')],
                        verbose=1)
    val_accuracy = history.history['val_accuracy'][-1]
    print(f"Trial {trial.number} finished with validation accuracy: {val_accuracy}")
    return val_accuracy


# Create an Optuna study and optimize
#study = optuna.create_study(direction='maximize')
#study.optimize(objective, n_trials=5)


# Get the best parameters
#best_params = study.best_params
#print("Best parameters: ", best_params)


# Train the model with the best parameters
best_model = create_model(num_units=57,
                          dropout_rate=0.36678015982771744,
                          learning_rate=0.0016592879780433026)
history = best_model.fit(X_train, y_train_categorical, epochs=50, validation_split=0.2, verbose=1)


# Save the model
model_path = 'best_lstm_model.h5'
best_model.save(model_path)
print(f"Model saved to {model_path}")


# Evaluate the model on the test set
test_loss, test_accuracy = best_model.evaluate(X_test, y_test_categorical, verbose=0)
print(f'Test accuracy: {test_accuracy}')
```

```
# Generate predictions
y_pred = best_model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# Confusion matrix
conf_matrix = confusion_matrix(y_test_encoded, y_pred_classes)
print("Confusion Matrix:")
print(conf_matrix)

# Classification report
class_report = classification_report(y_test_encoded, y_pred_classes, target_names=label_encoder.classes_)
print("Classification Report:")
print(class_report)

# Plot the training history
def plot_training_history(history):
    plt.figure(figsize=(12, 6))

    # Plot accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    # Plot loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

plot_training_history(history)
```