Network Intrusion Detection

Tamara Soltész
Master Student, Faculty of Informatics, Eötvös Loránd University Budapest
i0kwa3@inf.elte.hu

Abstract

The spread of internet-connected devices in homes and industrial environments has heightened the need for robust cybersecurity solutions. Network Intrusion Detection Systems (NIDS) offer essential protections by monitoring network traffic for signs of cyberattacks, which have evolved into sophisticated threats, often evading traditional detection methods. This study examines the cybersecurity vulnerabilities of IoT devices and industrial systems under the Industry 4.0 paradigm, where increased connectivity introduces novel risks. Machine learning (ML) techniques, such as supervised and unsupervised learning models, are highlighted as critical for enhancing NIDS efficacy. Specifically, Random Forest, Support Vector Machine (SVM), and Gradient Boosting algorithms demonstrate promise in detecting diverse attack patterns with high accuracy. Autoencoders further support anomaly detection by learning typical patterns in network traffic. Experimental results, derived from a dataset collected using a Raspberry Pi-based network monitoring system, validate the potential of ML-enhanced NIDS in mitigating cyber threats. This research contributes to the development of low-resource, cost efficient, tinyML, adaptable cybersecurity frameworks for protecting critical infrastructure against increasingly complex cyberattacks. To facilitate reproducibility and further research, the source code and datasets used in this study are publicly available at GitHub Repository.

Keywords: NIDs, Machine Learning, Anomaly Detection, Cybersecurity, Industry 4.0

1 Introduction

Detecting network attacks has become critically important due to the rapid spread of the internet and IoT devices. Our homes and workplaces are increasingly surrounded by smart devices that are constantly connected to the internet. These devices generate and share vast amounts of data,

significantly increasing cybersecurity risks. Attackers can easily exploit these vulnerabilities to access personal data or disrupt system operations. Moreover, in today's conflicts, cyberattacks have emerged as a new form of warfare, aimed at crippling critical infrastructures and compromising IT systems, thus granting significant strategic advantages to attackers.

In IoT and industrial environments, the challenge is even greater due to devices' limited computing and storage capacities. Therefore, low-resource-demand solutions that operate efficiently without overloading the systems are particularly important.

A Network Intrusion Detection System (NIDS) is an external security technology tool that monitors network traffic, detects suspicious activities, and sends alerts if it identifies signs of potential cyberattacks or unauthorized access. The goal of NIDS is to protect the network, maintain data security, and minimize potential damage.

Detecting and preventing network attacks is essential. However, attacks are becoming increasingly sophisticated and harder to detect, often remaining hidden from traditional detection methods used by these systems.

1.1 Cybersecurity Risks in Industry 4.0

The digitalization of industrial systems and manufacturing environments has significantly increased the connectivity of industrial equipment with corporate networks and the internet in recent years, i.e., in the industry 4.0 era. However, this heightened connectivity also raises the risk of cyber threats, as industrial robots connected to production networks are also vulnerable. Robot control systems are often inadequately secured, and if network security is weak, attackers can remotely manipulate these robots. This can lead to defective products or equipment malfunctions, causing severe economic damage.

Industrial environments are increasingly using IoT devices to continuously collect and transmit data. These devices frequently lack adequate security measures, making them easy targets for cyberattacks. Compromising even a single IoT device can jeopardize the security of the entire industrial system.

In industrial settings, weak cybersecurity can lead to production halts, theft of industrial secrets, and attacks on critical infrastructure that may endanger public safety [1].

1.2 Early life of Attacks

The roots of cybersecurity date back to the 1970s when the U.S. Department of Defense initiated the Ware Report [2]. This report established fundamental security guidelines recommended for the comprehensive protection of computer systems. The first computer virus, the Creeper, appeared in 1971, leaving a warning message in its wake. This was followed by Reaper, the first

antivirus program, designed to neutralize Creeper. In the 1980s, the development of ARPANET into the internet enabled broader dissemination of viruses and worms, such as the infamous Morris Worm attack in 1988.

In the 1990s, viruses like the Melissa virus introduced new challenges, especially regarding email system crashes and data security. During this period, numerous antivirus companies were founded to address increasingly severe threats. However, antivirus programs demanded significant resources and frequently generated false alarms, complicating the identification of actual threats.

In the 2000s, new technologies, including polymorphic and metamorphic viruses, introduced more complex cyber threats. Mid-decade, Stuxnet emerged as the first-known cyberweapon, demonstrating that cyberattacks could also be used for espionage purposes. Stuxnet was a notorious computer worm specifically designed to disrupt industrial equipment, particularly the programmable logic controllers (PLCs) of uranium enrichment centrifuges in Iran. The attackers targeted Siemens PLCs, which were used to control the uranium enrichment process. Stuxnet revealed that industrial systems could become direct targets of cyberattacks, carrying severe implications for modern geopolitical conflicts. In the 2010s, the proliferation of IoT (Internet of Things) and cloud-based systems introduced new challenges in cybersecurity. Cyberattacks became more targeted, specifically aiming to acquire business secrets or cripple services. The NotPetya (2016) and WannaCry (2017) ransomware attacks caused global damage, demonstrating that attacks on digital systems could even trigger international crises [3].

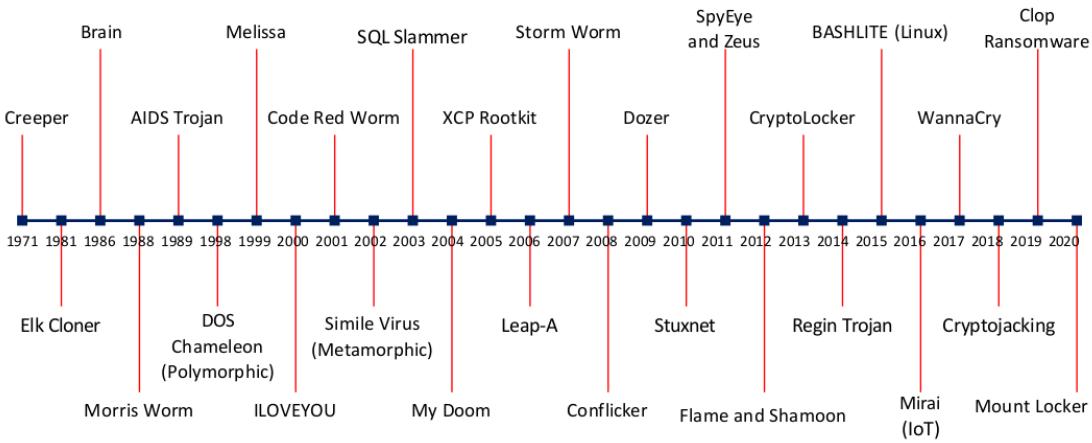

Figure 1: Timeline of CyberAttacks [2]

In 2017, another significant industrial attack was identified when the safety system of a chemical plant in the Middle East was targeted. TRITON, also known as TRISIS or HatMan, was one of the first malware programs specifically aimed at Safety Instrumented Systems (SIS). SIS represents the last line of defense in industrial processes, designed to prevent critical incidents such as explosions and fires. During the attack, the malware targeted Triconex safety controllers, manipulating the plant's safety systems and creating hazardous conditions, thereby increasing the risk of severe industrial incidents.

The significance of TRITON lies not only in its ability to directly attack SIS systems but also in its exposure of vulnerabilities within modern industrial control systems. This incident prompted industrial stakeholders to enhance their defensive mechanisms, particularly by introducing improved monitoring tools and simulation techniques to detect similar attacks early [4].
Furthermore, the attackers utilized artificial intelligence and machine learning techniques to increase the effectiveness of their attacks.
The key response to this step is the emergence of various artificial intelligence and machine learning methods in the defense line for recognizing attack patterns.

1.3 Machine Learning in Cybersecurity

Machine learning plays a vital role in cybersecurity by improving Intrusion Detection Systems, which detect malicious network activities. Traditional signature-based IDS rely on known attack patterns, making them vulnerable to novel threats. However, anomaly-based IDS, using ML techniques, can recognize unusual patterns even if they don't match pre-existing signatures [5]. The primary advantage is its adaptability in identifying zero-day attacks, which are previously unknown vulnerabilities [6].
Key machine learning techniques used in cybersecurity include supervised, unsupervised, and hybrid models. These models are applied to analyze network behavior, detect malware, prevent data breaches, and manage real-time threat responses. Supervised learning is trained on labeled datasets to identify known threats, while unsupervised learning models work with unlabeled data, making them suitable for anomaly detection. Hybrid models combine both, improving detection accuracy and reducing false alarms [7] [8].

DL techniques are highly effective in cybersecurity applications, as they can automatically learn from large and complex datasets, such as those used in network traffic or malware analysis. Although the use of DL applications in cybersecurity environments is growing, it also presents challenges, such as ensuring real-time applicability, robustness [9], and the lack or outdated nature of suitable public datasets related to cybersecurity data. Several prior related studies have highlighted [8, 10, 11] that up-to-date, modern datasets are essential for the successful detection of today's advanced attacks. This is especially important in industrial applications, as it helps minimize false alarms and quickly adapt to new threats, which is critical for the safety and uninterrupted operation of industrial systems. Additionally, modern datasets support the effective training of advanced machine learning models and help ensure compliance with security regulations.

2 Methods

2.1 Data Collecting

2.1.1 Relevant Network Data According to Industrial Machines

The objective of monitoring network traffic between industrial equipment such as CNC machines, 3D printers, and welding robots, is to establish a current and relevant dataset that enables the identification and prevention of potential attacks. The collected data have been categorized according to the following primary data groups:

- **Network Metada:** To identify the devices involved in communication and analyze connection quality, we recorded the source and destination IP addresses as well as the ports used. Additionally, the protocols employed (such as TCP, UDP, MQTT, Modbus, OPC UA) and timestamps were recorded, as these are essential for analyzing time-based patterns and anomalies.
- **Packet-Level Data:** The packet size and packet count were recorded for each data packet, as these metrics facilitate the identification of abnormal data transmissions. Additionally, we monitored TCP connection flags (such as SYN, ACK, FIN), which help in detecting scanning attempts and connection anomalies.

Data collected through network traffic monitoring are of critical importance in structuring the database, as this information enables comprehensive tracking and analysis of network events. Such data include communication patterns between devices, packet sizes, connection durations, and protocol-specific information, which collectively facilitate the identification of the system's normal operational baseline. Additional correlations observed within the traffic flow provide further insights, enhancing the precision and relevance of the database structure.

- **Statistical Characteristics of Data Flows:** Measuring the average and maximum data transfer rates, as well as communication duration, aids in the early detection of threats such as DoS and DDoS attacks. Monitoring the volume and timing characteristics of data traffic is essential for maintaining network stability and preventing potential congestion.
- **Logging Data:** Error messages, login attempts, and authentication events from devices complement the data collection, particularly as access and configuration attempts in industrial systems are frequent targets for cyberattacks.

- **Device-Specific Data:** By analyzing the frequency of communication and command exchanges between devices, we obtained a consistent view of typical operational patterns for industrial equipment. Commands and configuration changes that deviate from these norms may indicate unauthorized access, making their monitoring a priority.

- **Anomaly-Based Features:** Patterns of unusual network activity, including temporally compressed, repetitive requests or atypical inter-device communications, were also recorded. This approach enables the identification of rapidly changing or potentially hazardous patterns.

- **Protocol-Specific Information:** Recording the message structures used by industrial protocols such as Modbus, OPC UA, and others allows for the identification of communication that deviates from normal patterns. Monitoring command sequences enables adherence to predefined patterns, providing alerts in the event of deviations that may indicate suspicious activity.

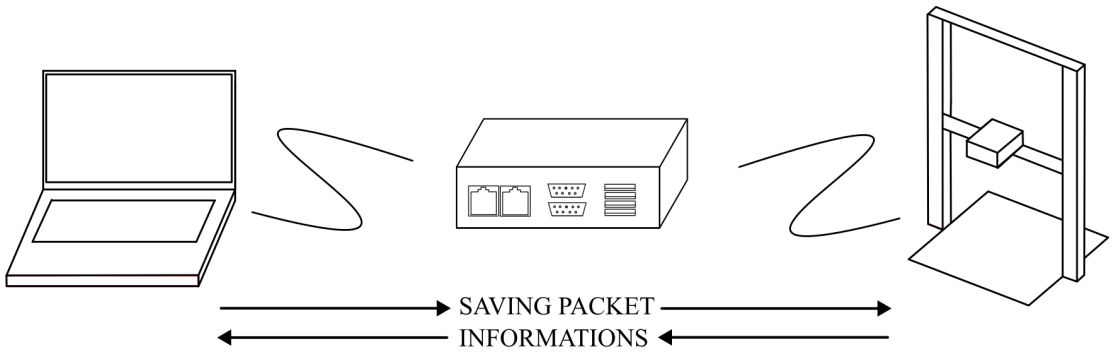

Figure 2: Concept of Collecting Data

These details facilitate the construction of a more precise database, which supports the training of machine learning models, thereby enhancing the detection of potential cyber threats and attacks, such as DoS attempts or data exfiltration. Consequently, the comprehensive analysis of monitored network traffic contributes to strengthening network security, reducing attack risks, and establishing a reliable database architecture.

2.1.2 Network Monitoring with Raspberry PI

To monitor the network traffic between a CNC machine and a computer, we designed and implemented a Raspberry Pi-based data collection system, utilizing a Python program and SQLite database, which is stored in the Raspberry PI's SD card. The system aims to continuously

gather and analyze network metadata, packet-level data, device-specific information, statistical characteristics of data flows, protocol-specific details, anomaly-based features, and logging data.

Installation of necessary libraries

To initiate the network monitoring project, it is essential to install all necessary software and Python libraries to ensure the program operates smoothly and can handle network traffic as designed. This installation process includes downloading and configuring required packages, libraries, and dependencies, especially those associated with network monitoring, packet capturing, and database management.

```
sudo apt update
sudo apt install python3-pip
pip3 install scapy
```

Once all required components are installed, the network monitoring program can be initiated. To start the program, the following command must be executed in the terminal with superuser privileges to access network interfaces and create database entries effectively:

```
sudo python3 network_monitoring.py
```

This command instructs the Raspberry Pi to execute the script, which initiates the network monitoring process. Running the script with sudo privileges is necessary, as network packet capturing and monitoring operations typically require elevated permissions to interact directly with network interfaces and write data to storage. Ensuring that the program is launched with appropriate privileges enables the system to perform its functions comprehensively, including packet capturing, data processing, and secure storage.

Implementation

The monitoring system is based on a Raspberry Pi platform, which uses a Python program to capture network traffic through the interfaces (eth0 and eth1). After recording the data, it forwards the traffic between the interfaces. The program employs the scapy library to process packets and retransmit data, while SQLite provides the data storage structure. The data mentioned in the section 2.1.1 are captured and analyzed.

```
import sqlite3
from scapy.all import sniff, IP, TCP, UDP, Raw, sendp
from datetime import datetime
```

Source Code 1: Import the necessary packages

The implementation utilizes several Python libraries to facilitate network traffic monitoring, data storage, and packet forwarding on the Raspberry Pi. The `sqlite3` library, a built-in Python module for SQLite database management, enables the storage of network traffic data in a local database on the device. Using `sqlite3`, the program creates and manages database tables, allowing the efficient insertion and retrieval of recorded packet information for subsequent analysis.

The `scapy` library is a powerful packet manipulation and analysis tool employed for capturing, analyzing, and forwarding network packets. The `sniff` function monitors incoming network traffic on the specified interface, while `IP`, `TCP`, and `UDP` classes allow access to protocol-specific data fields, supporting detailed analysis of the captured packets. The `Raw` module is used to extract the raw payload content of packets, capturing essential data within each transmission. Additionally, `sendp` facilitates packet forwarding on a secondary network interface, allowing the Raspberry Pi to relay traffic after recording it.

Finally, the `datetime` library, specifically the `datetime.now()` function, records the precise timestamp of each packet, supporting accurate temporal analysis within the stored dataset. Together, these libraries enable comprehensive real-time network monitoring, data management, and traffic forwarding, forming a robust monitoring system on the Raspberry Pi.

```
conn = sqlite3.connect('/home/pi/network_monitoring.db')
cursor = conn.cursor()

cursor.execute('''
CREATE TABLE IF NOT EXISTS traffic (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp TEXT,
    src_ip TEXT,
    dst_ip TEXT,
    protocol TEXT,
    src_port INTEGER,
    dst_port INTEGER,
    length INTEGER,
    flags TEXT,
    message_content TEXT
)
''')
conn.commit()
```

Source Code 2: Create and connect to database

The path `/home/pi/network_monitoring.db` is used by this code to establish an SQLite database connection on the Raspberry Pi. The line `conn = sqlite3.connect(...)` initiates the connection to the database, while `cursor = conn.cursor()` creates a cursor that enables the execution of SQL commands.

The `CREATE TABLE IF NOT EXISTS` statement creates a new table named `traffic` in the database if it does not already exist. This table includes several columns, such as `timestamp` (recording the time of the entry), `src_ip` and `dst_ip` (source and destination IP addresses), `protocol` (the protocol type), along with additional attributes that store specific details of the network traffic. Finally, the `conn.commit()` statement saves these changes to the database, ensuring the table is created or updated as needed.

```
def process_packet(packet):
    if IP in packet:
        src_ip = packet[IP].src
        dst_ip = packet[IP].dst
        length = len(packet)
        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

        if TCP in packet:
            protocol = "TCP"
            src_port = packet[TCP].sport
            dst_port = packet[TCP].dport
            flags = packet[TCP].flags
        elif UDP in packet:
            protocol = "UDP"
            src_port = packet[UDP].sport
            dst_port = packet[UDP].dport
            flags = None
        else:
            protocol = "OTHER"
            src_port = None
            dst_port = None
            flags = None

        if Raw in packet:
            message_content = str(packet[Raw].load)
        else:
            message_content = None

        cursor.execute('''
        INSERT INTO traffic (timestamp, src_ip, dst_ip, protocol, src_port,
            dst_port, length, flags, message_content)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        ''', (timestamp, src_ip, dst_ip, protocol, src_port, dst_port,
            length, flags, message_content))
        conn.commit()

        sendp(packet, iface="eth1", verbose=False)
```

Source Code 3: Data extraction with function

The `process_packet` function is designed to extract, store, and forward network packet data in a structured process. Initially, the function checks for the presence of an IP layer in the packet; if available, it retrieves essential information such as the source and destination IP addresses (`src_ip`, `dst_ip`), packet length (`length`), and a timestamp (`timestamp`) marking the exact capture time. Subsequently, the function verifies the protocol type, distinguishing between TCP and UDP. For TCP packets, it further extracts the source and destination ports (`src_port`, `dst_port`) and any TCP flags present, while marking non-TCP/UDP packets as "OTHER" for classification purposes.

If the packet contains raw payload data (`Raw`), the function captures it as `message_content`.

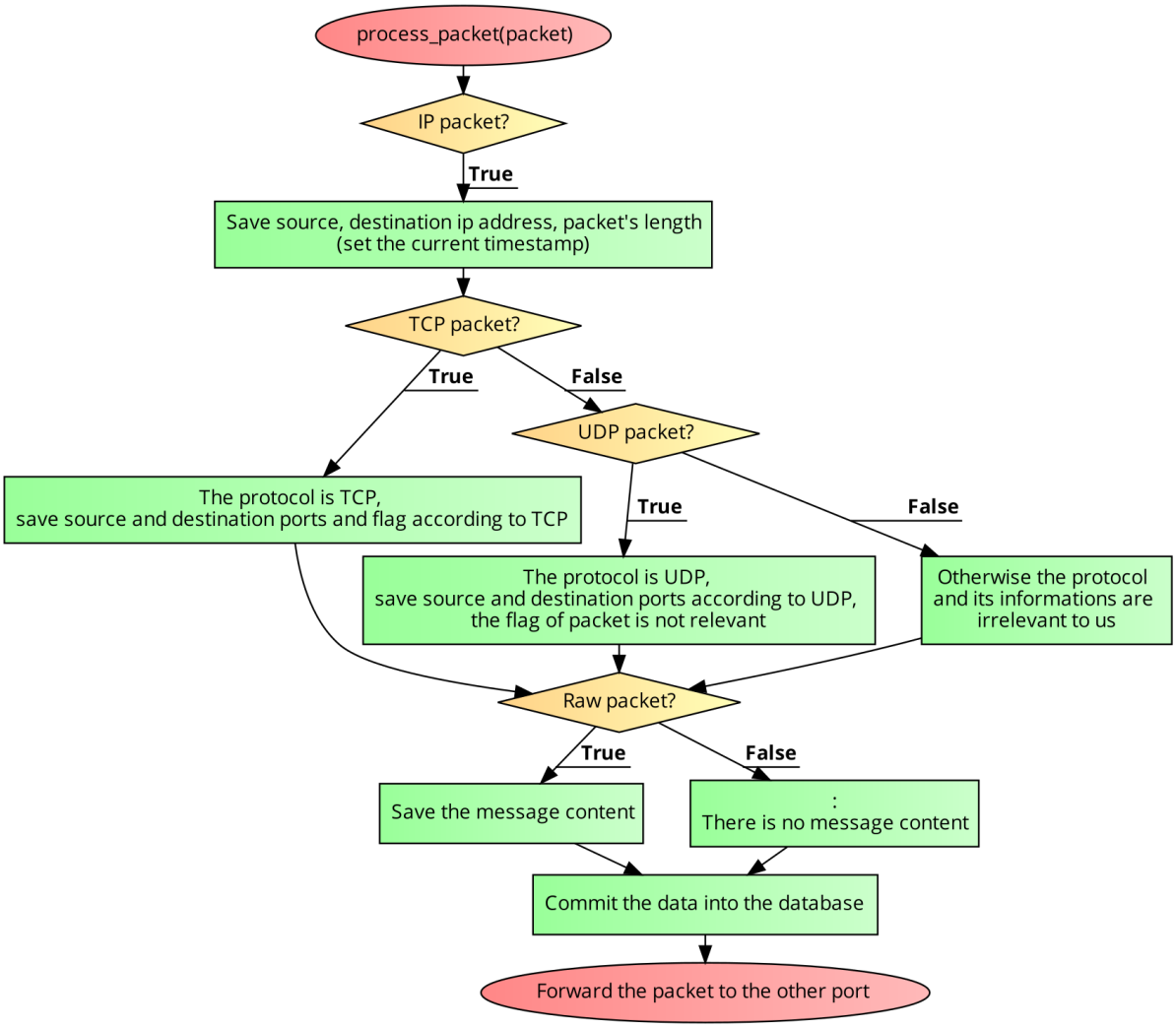

Figure 3: Flowchart of `process_packet()` function

All retrieved data are then stored in the `traffic` database table via an SQL `INSERT` command, with changes committed to the database to ensure data persistence. Finally, the function forwards the packet through a secondary network interface using the `sendp` function. This process ensures comprehensive capture, storage, and relay of network traffic data, facilitating real-time monitoring and analysis.

```
sniff(iface="eth0", prn=process_packet, store=0)
conn.close()
```

Source Code 4: Start the monitoring, and close the connection

The code snippet performs two key operations for efficient network traffic monitoring and data management. The `sniff(iface="eth0", prn=process_packet, store=0)` function, from the `scapy` library, captures network traffic on the specified interface, `eth0`. This parameter designates `eth0` as the monitored interface, capturing all incoming packets. For each packet, the `sniff` function invokes the `process_packet` function, which processes, stores, and forwards the packet according to predefined procedures. Setting `store=0` ensures that the captured packets are not stored in memory, optimizing memory usage and improving performance by directly processing packets without retaining them.
The `conn.close()` command subsequently closes the database connection established earlier with `sqlite3.connect(...)`. This step ensures that all open database resources are properly released, safely concluding the database session. Together, these operations facilitate real-time network monitoring, packet processing, and secure data handling, supporting a robust and efficient monitoring system.

2.1.3 Network Monitoring with Arduino

To implement a network traffic monitoring and logging system on an Arduino platform, specific hardware and software resources are necessary to accommodate the device's capabilities and ensure optimal functionality within the constraints of embedded systems.

Hardware and Software Requirements

The Arduino controller has to be able to connect via Ethernet. Ethernet functionality is critical for enabling direct network communication between devices, allowing the Arduino to act as an intermediary for packet forwarding and logging. To enable Ethernet connectivity on an Arduino, an Ethernet shield is required.
To achieve persistent data storage, an SD card module is integrated with the Arduino via the SPI interface. This module allows the device to log network traffic data in real-time to an external storage medium, enabling efficient data handling and post-processing capabilities. The SD card also provides a cost-effective and accessible method for extending storage capacity beyond the limitations of the Arduino's onboard memory. To handle this hardware requirements correctly, we need some software utilities.
The Arduino Ethernet library is utilized to establish and manage network connections between the Arduino and other networked devices. This library provides essential functions to set up a

server, handle client connections, and relay data, thus facilitating bidirectional communication between the CNC machine and the PC.

The SD library, included in the Arduino IDE, enables interaction with the SD card module for creating and managing files. It allows the Arduino to store network traffic logs in a structured format on the SD card, thus supporting later analysis of recorded data.

Figure 4: Arduino Shield

Implementation

The Arduino requires a predefined network configuration, including a unique MAC address and a static IP address, to communicate effectively over the network. Configuring these parameters enables the Arduino to establish consistent communication channels for capturing and relaying network traffic.

```
#include <SPI.h>
#include <Ethernet.h>
#include <SD.h>

#define SD_CS_PIN 4
#define BUFFER_SIZE 512

byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 1, 201);
EthernetServer server(80);
File logFile;
```

Source Code 5: Predefining the libraries and variables

This code segment provides the core setup for a network-enabled Arduino using the SPI, Ethernet, and SD libraries essential for embedded systems to record network traffic.

Table 1: Arduino Libraries and Definitions

Library/Definition	Description
`SPI.h`	The Serial Peripheral Interface (SPI) library allows the Arduino to communicate with external components, including Ethernet and SD card modules. This interface is crucial for fast data transfers in network applications.
`Ethernet.h`	The Ethernet library provides functions to manage network connections, handle communication protocols, and set up an Ethernet server on the Arduino. This library transforms the Arduino into a web server or client, allowing it to interact with other devices on the network.
`SD.h`	The SD library manages data storage on an SD card, offering file system operations such as reading, writing, and creating files. This functionality is critical for data logging applications, where large volumes of data need to be stored persistently.
`SD_CS_PIN 4`	This line defines the pin number used to select the SD card module. In this context, the `SD_CS_PIN` variable specifies the *chip select (CS)* pin, which allows the Arduino to communicate directly with the SD card over the SPI interface. Pin 4 is typically used with the Arduino Ethernet Shield.
`BUFFER_SIZE 512`	The buffer size for packet data is set to 512 bytes, defining the maximum amount of data that can be temporarily stored in memory as the Arduino processes incoming network packets. This buffer size is a compromise between memory limitations and the need to handle adequate data payloads effectively.

```
void initSDCard() {
    if (!SD.begin(SD_CS_PIN)) {
        Serial.println("SD card initialization failed!");
        while (true);
    }
    Serial.println("SD card initialized.");
}

void logDataToSD(String data) {
    logFile = SD.open("network_log.txt", FILE_WRITE);
    if (logFile) {
        logFile.println(data);
        logFile.close();
    } else {
        Serial.println("Error opening log file!");
    }
}
```

Source Code 6: Functions for handling the SD Card

This code segment demonstrates the initialization and data logging procedures for an SD card in an embedded system context, specifically within an Arduino environment. These functions are essential for data persistence in networked embedded devices, where real-time logging and retrieval of information are critical for analysis and system integrity.

The `initSDCard()` function initializes the SD card and checks for successful connectivity before proceeding with logging operations. It serves as a foundational routine to ensure that the data logging system is functional.

The `logDataToSD(String data)` function manages data storage on the SD card by appending each data entry to a log file, `network_log.txt`. By creating and maintaining a record of network activity, this function enables persistent data storage, which is crucial for systems that require historical data for later analysis.

```
void processPacket(char *buffer, int length, IPAddress srcIP, IPAddress
    dstIP, String protocol) {
     String timestamp = String(day()) + "/" + String(month()) + "/" + String
         (year()) + " " + String(hour()) + ":" + String(minute()) + ":" +
         String(second());


     String logEntry = timestamp + ", SRC IP: " + srcIP.toString() + ", DST
         IP: " + dstIP.toString() + ", Protocol: " + protocol + ", Length: "
         + String(length);

     Serial.println(logEntry);
     logDataToSD(logEntry);
}
```

Source Code 7: Functions for process the Package

The function `processPacket()` processes network packet data by constructing a structured log entry, which it subsequently outputs to both the serial monitor and an SD card for persistent storage. This approach is particularly valuable in embedded network monitoring applications, where resource-efficient data processing and logging are essential for ongoing system performance and data integrity.

```
void setup() {
     Serial.begin(9600);
     Ethernet.begin(mac, ip);
     server.begin();
     initSDCard();
}
```

Source Code 8: The `setup()` function

The `setup()` functions initialize the key components of the Arduino program. This routine is integral to the system's startup sequence, ensuring that each subsystem (communication, data storage, and serial output) is configured correctly before the device begins its monitoring operations.

```
void loop() {
    EthernetClient client = server.available();

    if (client) {
        char buffer[BUFFER_SIZE];
        int index = 0;

        IPAddress srcIP = client.remoteIP();
        IPAddress dstIP = Ethernet.localIP();
        String protocol = "TCP";

        while (client.connected() && client.available() && index <
            BUFFER_SIZE) {
            buffer[index++] = client.read();
        }

        if (index > 0) {
            processPacket(buffer, index, srcIP, dstIP, protocol);

            EthernetClient relayClient = server.available();
            if (relayClient) {
                relayClient.write((uint8_t*)buffer, index);
            }
        }

        client.stop();
    }
}
```

Source Code 9: The `loop()` function

The `Serial.begin(9600)` command initializes communication at a baud rate of 9600, establishing a connection between the Arduino and a connected computer. This channel is essential for debugging and real-time feedback, enabling users to monitor system status directly on the serial monitor. Serial communication offers researchers a simple yet effective means to verify device states, observe initialization sequences, and receive runtime diagnostics, which are invaluable during both prototyping and deployment phases.

The Ethernet setup follows with `Ethernet.begin(mac, ip)` and `server.begin()`, which initialize network communication by assigning a unique MAC address (`mac`) and a static IP address (`ip`). This configuration allows the Arduino to establish consistent network identity

and stability, critical for seamless device interactions. The Ethernet server is then started on port 80, enabling the Arduino to function as a network server that can handle incoming traffic, supporting networked interactions like those between a CNC machine and a PC. The configuration of `mac` and `ip` ensures stable communication channels that are particularly valuable in cybersecurity studies, network traffic analysis, and IoT research, where reliable and structured data flow is fundamental.

The SD card is initialized with the `initSDCard()` function, which verifies proper connection to the SD card, allowing the Arduino to engage in persistent data logging. This setup provides ample storage capacity to handle large amounts of network traffic data without overloading the Arduino's limited memory. Persistent logging is crucial in research that requires longitudinal data, supporting anomaly detection in cybersecurity and network performance analysis, where stored data aids in identifying patterns and trends over time.

Due to the limitations in processing power and memory capacity, the Arduino cannot monitor network data as extensively as a Raspberry Pi. As a simple microcontroller-based device, the Arduino lacks the capability to efficiently handle complex network protocols and large data packets. As a result, it can only record basic information, such as source and destination IP addresses and packet size, but detailed data (including packet content, headers, or specific protocol fields) cannot be observed or analyzed comprehensively. The Arduino's limitations make it suitable primarily for simpler tasks, while the Raspberry Pi enables more complex and detailed network data collection and analysis.

2.2 Data Pre-processing

To ensure the construction of a well-rounded dataset suitable for testing a deep learning-based intrusion detection system, a baseline collection of approximately 1,500 benign traffic samples was initially assembled. As no malicious activity was included in this dataset, the inclusion of attack-related samples became necessary to support both training and evaluation. The presence of labeled data is considered essential for supervised deep learning models, as these systems rely on accurately annotated examples of both "normal" and "attack" traffic to effectively learn and distinguish between different traffic types.

To address the absence of malicious samples, two complementary strategies were tried.

2.2.1 Synthetically generated attacks

Randomly generated data points were first created to represent plausible intrusion scenarios. These data points were labeled as "attack," while the original, collected samples were retained as "normal." This approach ensured that representative attack patterns were included in the

dataset without relying solely on external sources.

To achieve our goal, a simple Python program was utilized, enabling the rapid and efficient generation of data as well as the automation of the required labeling process. This approach not only reduced the time required but also ensured the consistency of the dataset, which is essential for subsequent analysis and modeling.

```
import pandas as pd
import random
from datetime import datetime, timedelta
```

Source Code 10: Import the necessary packages

The `pandas` library is utilized to manage the original traffic data in a structured DataFrame format, supporting data manipulations such as filtering unique IP addresses, ports, and protocols, as well as augmenting and exporting the dataset. The `random` module provides the foundation for introducing randomness, which is crucial for simulating realistic attack patterns; for instance, it generates random timestamps, protocols, or ports to produce diverse data. Furthermore, the combination of the `datetime` and `timedelta` modules enables precise timestamp manipulations, which are critical for constructing time-based attack patterns.

```
def random_timestamp(start_date, end_date, f="%Y-%m-%d %H:%M:%S"):
    start = datetime.strptime(start_date, f)
    end = datetime.strptime(end_date, fo)
    random_time = start + timedelta(seconds=random.randint(0, int((end -
        start).total_seconds())))
    return random_time.strftime(f)
```

Source Code 11: Function for generate random timestamps

The `random_timestamp` function generates a random timestamp within a specified date range. This function is essential in the context of synthetic data generation, as it ensures that the generated attack data contains realistic and diverse temporal patterns. By assigning timestamps within a defined period, the function adds temporal variability, which is critical for simulating real-world scenarios where network activity and attacks occur over time. This capability makes the synthetic dataset more robust and suitable for training machine learning models or testing intrusion detection systems.

```python
def generate_synthetic_attacks(data, num_attacks=1500):
    data = pd.DataFrame(data)
    synthetic_data = []
    unique_ports = range(min(data['src_port']), (max(data['src_port'])))
    protocols = data['protocol'].unique()
    for _ in range(num_attacks):
        length = random.randint(min(data['length']), (min(data['length'])))
        if random.random() < 0.3:
            length = length + random.randint(100, 500)
        protocol = random.choice(protocols)
        flag = ''
        if (protocol == "TCP"):
            flag = random.choice(['SYN', 'ACK', 'FIN'])
        filtered = data[data['flags'] == flag]
        if not filtered.empty:
            src_port=random.choice(filtered['src_port'].dropna().tolist())
            dst_port=random.choice(filtered['dst_port'].dropna().tolist())
        else:
            src_port = '192.168.0.100'
            dst_port = '192.168.0.138'
        synthetic_data.append({
            'id': len(data) + len(synthetic_data) + 1,
            'timestamp': random_timestamp("2024-10-14 00:00:00",
                "2024-11-20 00:00:00"),
            'src_ip': random.choice(ips),
            'dst_ip': random.choice(ips),
            'protocol': protocol,
            'src_port': src_port,
            'dst_port': dst_port,
            'length': length,
            'flags': flag,
            'message_content': random.choice(data['message_content']),
            'label': 'attack'
        })
    return pd.DataFrame(synthetic_data)
```

Source Code 12: Function for generate synthetic attacks

The `generate_synthetic_attacks` function is designed to create synthetic network attack data by augmenting the original dataset. It takes as input a dataset of normal network traffic and a parameter specifying the number of synthetic attacks to generate. The function begins by converting the input data into a structured pandas DataFrame and extracting essential attributes such as unique destination IP addresses, a range of possible source ports, and the protocols present in the dataset. An empty list is initialized to store the synthetic attack records. For each attack, the function randomly determines key characteristics such as packet length, protocol,

TCP flags (if applicable), and source and destination ports. Many of these attributes are deliberately sampled from the existing dataset to ensure that the synthetic records closely resemble real-world traffic patterns. This approach not only adds realism to the synthetic attacks but also makes them harder to distinguish from normal traffic, effectively preventing simple detection based on superficial anomalies.

The function also assigns a random timestamp to each attack using the `random_timestamp` function and selects message content from the existing data. Each record is labeled as an attack and appended to the synthetic data list. The final output is a pandas DataFrame containing the generated synthetic attack data.

By basing much of the synthetic data on the original dataset, the function ensures that the generated attacks are realistic and diverse, enhancing the robustness of the augmented dataset. This strategy allows machine learning models trained on the dataset to develop a deeper understanding of complex patterns in network traffic, ultimately leading to more accurate predictions and better performance in intrusion detection and anomaly detection tasks.

```
original_data = pd.read_csv('traffic.csv')
original_data['label'] = 'normal'
synthetic_attacks = generate_synthetic_attacks(original_data)
augmented=pd.concat([original_data, synthetic_attacks], ignore_index=True)
augmented.to_csv('traffic_with_synthetic_attacks.csv', index=False)
```

Source Code 13: The main part of the file generate_synthetic_attacks.py

The main part of the code begins by reading the collected network traffic dataset from a CSV (Comma-separated values) file. Although the code running on the Raspberry Pi initially stored the collected data in a SQLite database (`.db`) file, exporting the tables into CSV format was deemed advantageous to facilitate subsequent processing and analysis. Python-based processing routines generally read and handle data more efficiently and straightforwardly from CSV files than from database files. To this end, the contents of the `.db` file can be readily extracted using a simple database management tool (such as the `sqlite3` command-line utility), thereby providing swift and convenient access to the corresponding CSV files for further analyses and experimentation.

The data is loaded into a pandas DataFrame for structured handling and manipulation. A new column named `label` is added to the DataFrame, with the value `normal` assigned to all rows, indicating that the existing dataset represents legitimate network traffic.

Next, the `generate_synthetic_attacks` function is called to create synthetic attack records based on the structure and attributes of the original dataset. The function is configured to generate approximately equal number of attack records as the normal values. These attacks are then concatenated with the original dataset using the `pd.concat` function, resulting in a single augmented dataset containing both normal traffic and labeled attack data.

Finally, the combined dataset is saved to a new CSV file, which can be used in the supervised

learning methods. Since the code operates with random data, the outcome of the program will differ with each execution, ensuring that the generated synthetic data is never identical across runs.

Table 2: Some features of the generated dataset

src_ip	dst_ip	protocol	src_port	dst_port	length	flags	label
192.168.1.100	192.168.1.138	UDP	64593	62868	30		attack
192.168.1.100	192.168.1.138	UDP	54248	56980	30		attack
192.168.1.100	192.168.1.100	TCP	54669	22	271	ACK	attack
192.168.1.100	192.168.1.138	UDP	52100	57504	30		attack
192.168.1.100	192.168.1.100	TCP	58933	22	30	SYN	attack
192.168.1.100	192.168.1.100	TCP	65168	443	167	SYN	attack
192.168.1.138	192.168.1.100	UDP	58960	161	123		normal
192.168.1.138	192.168.1.100	UDP	63219	123	192		normal
192.168.1.138	192.168.1.100	UDP	51947	53	178		normal
192.168.1.100	192.168.1.138	TCP	64770	443	34	FIN	normal
192.168.1.100	192.168.1.100	TCP	65341	80	107	ACK	normal
192.168.1.138	192.168.1.100	TCP	63823	443	32	SYN	normal
192.168.1.138	192.168.1.100	UDP	57303	123	30		normal
192.168.1.138	192.168.1.100	TCP	50647	443	144	FIN	normal
192.168.1.138	192.168.1.100	TCP	59546	80	137	SYN	normal
192.168.1.138	192.168.1.100	TCP	64973	46	46	RST	normal
192.168.1.138	192.168.1.100	TCP	57848	59	59	FIN	normal
192.168.1.138	192.168.1.100	TCP	59557	161	74		normal

2.2.2 Real attacks from an existing database

Another approach to augmenting the dataset with attack data is to extract records from an existing database containing real attacks, which may be available online and publicly accessible. These extracted records can then be integrated into the collected data after careful preprocessing. This method not only saves time compared to manually generating attack patterns but also ensures that the augmented dataset is based on real attack events, thereby enhancing the accuracy and applicability of model development.

It is important to note that the two datasets must contain the same stored parameters to ensure comparability in subsequent analyses and maintain data compatibility. This is particularly critical for the naming conventions, formats, and value ranges of the columns, as any discrepancies between the datasets could lead to data processing errors or inaccurate performance of machine learning models. Achieving this may require preprocessing steps, such as renaming columns, recoding data values, or reformatting the data to align the structure of the datasets.

To support the development and evaluation of intrusion detection systems and machine learning models, several publicly available network traffic datasets provide a diverse range of real-world and synthetic data. Below are examples of commonly used datasets:

1. **CICIDS2017 Dataset**

   Developed by the Canadian Institute for Cybersecurity, this dataset includes labeled network traffic data with both normal and attack patterns such as Distributed Denial of Service (DDoS), brute force, and botnet attacks. CICIDS2017 is frequently used in machine learning research for evaluating intrusion detection systems.

Table 3: CICIDS2017 Dataset [12]

SNo	Feature Name	SNo	Feature Name	SNo	Feature Name
1	Flow ID	29	Fwd IAT Max	57	ECE Flag Count
2	Source IP	30	Fwd IAT Min	58	Down/Up Ratio
3	Source Port	31	Bwd IAT Total	59	Average Packet Size
4	Destination IP	32	Bwd IAT Mean	60	Avg Fwd Segment Size
5	Destination Port	33	Bwd IAT Std	61	Avg Bwd Segment Size
6	Protocol	34	Bwd IAT Max	62	Fwd Avg Bytes/Bulk
7	Timestamp	35	Bwd IAT Min	63	Bwd Avg Bytes/Bulk
8	Flow Duration	36	Fwd PSH Flags	64	Fwd Avg Packets/Bulk
9	Total Fwd Packets	37	Bwd PSH Flags	65	Bwd Avg Packets/Bulk
10	Total Backward Packets	38	Fwd URG Flags	66	Fwd Avg Bulk Rate
11	Total Length of Fwd Packets	39	Bwd URG Flags	67	Bwd Avg Bulk Rate
12	Total Length of Bwd Packets	40	Fwd Header Length	68	Subflow Fwd Packets
13	Fwd Packet Length Max	41	Bwd Header Length	69	Subflow Fwd Bytes
14	Fwd Packet Length Min	42	Fwd Avg Bytes/Bulk	70	Subflow Bwd Packets
15	Fwd Packet Length Mean	43	Fwd Packets/s	71	Subflow Bwd Bytes
16	Fwd Packet Length Std	44	Bwd Packets/s	72	Init_Win_Bytes_forward
17	Bwd Packet Length Max	45	Min Packet Length	73	Init_Win_Bytes_backward
18	Bwd Packet Length Min	46	Max Packet Length	74	act_data_pkt_fwd
19	Bwd Packet Length Mean	47	Packet Length Mean	75	min_seg_size_forward
20	Bwd Packet Length Std	48	Packet Length Std	76	Active Mean
21	Flow Bytes/s	49	Packet Length Variance	77	Active Std
22	Flow IAT Mean	50	FIN Flag Count	78	Active Max
23	Flow IAT Std	51	SYN Flag Count	79	Active Min
24	Flow IAT Max	52	RST Flag Count	80	Idle Mean
25	Flow IAT Min	53	PSH Flag Count	81	Idle Std
26	Fwd IAT Total	54	ACK Flag Count	82	Idle Max
27	Fwd IAT Mean	55	URG Flag Count	83	Idle Min
28	Fwd IAT Std	56	CWE Flag Count	84	

2. **NSL-KDD Dataset**

An improved version of the original KDD'99 dataset, NSL-KDD removes redundant records and corrects issues present in the original. It includes labeled data for both normal traffic and various types of attacks, making it suitable for evaluating intrusion detection models.

Table 4: NSL-KDD Dataset [13]

Num	Function	Num	Function
1	Duration	22	is_guest_login
2	protocol_type	23	Count
3	Service	24	srv_count
4	Flag	25	serror_rate
5	src_bytes	26	srv_serror_rate
6	dst_bytes	27	rerror_rate
7	Land	28	srv_rerror_rate
8	wrong_fragment	29	same_srv_rate
9	Urgent	30	diff_srv_rate
10	Hot	31	srv_diff_host_rate
11	num_failed_logins	32	dst_host_count
12	logged_in	33	dst_host_srv_count
13	num_compromised	34	dst_host_same_srv_rate
14	root_shell	35	dst_host_diff_srv_rate
15	su_attempted	36	dst_host_same_src_port_rate
16	num_root	37	dst_host_srv_diff_host_rate
17	num_file_creations	38	dst_host_serror_rate
18	num_shells	39	dst_host_srv_serror_rate
19	num_access_files	40	dst_host_rerror_rate
20	num_outbound_cmds	41	dst_host_srv_rerror_rate
21	is_host_login		

3. **USTC-TFC Dataset**

The USTC-TFC2016 is a publicly available network traffic dataset developed by the University of Science and Technology of China (USTC). The dataset aims to facilitate network traffic classification and the detection of malicious activities. It contains traffic samples categorized into two main groups: benign and malicious. The benign traffic includes data from applications such as BitTorrent, Facetime, FTP, Gmail, MySQL, Outlook, Skype, SMB, Weibo, and World of Warcraft. The malicious traffic comprises samples from Cridex, Geodo, Htbot, Miuref, Neris, Nsis-ay, Shifu, Tinba, Virut, and Zeus.

The USTC-TFC2016 dataset has been widely utilized in research, particularly for the classification of encrypted network traffic and the detection of malicious activities. For instance, a 2023 study introduced a higher-order graph neural network model tested on the USTC-TFC dataset, demonstrating improved accuracy in classifying encrypted traffic

[14]. Another study from the same year employed a one-dimensional convolutional neural network using the USTC-TFC2016 dataset, achieving an average accuracy of 98.8% in identifying network traffic categories [15].

4. **UNSW-NB15 Dataset**

   This dataset, developed by the Australian Centre for Cyber Security, provides a modern and realistic representation of network traffic, including normal behavior and nine types of cyberattacks (e.g., DoS, backdoors, reconnaissance). It contains 49 features extracted from raw network traffic and is widely used in intrusion detection and machine learning research.

Table 5: UNSW-NB15 Dataset [16]

ID	Feature	ID	Feature	ID	Feature
1	attack_cat	16	dloss	31	response_body_len
2	dur	17	sinpkt	32	ct_srv_src
3	proto	18	dinpkt	33	ct_state_ttl
4	service	19	sjit	34	ct_dst_ltm
5	state	20	djit	35	ct_src_dport_ltm
6	spkts	21	swin	36	ct_dst_sport_ltm
7	dpkts	22	stcpb	37	ct_dst_src_ltm
8	sbytes	23	dtcpb	38	is_ftp_login
9	dbytes	24	dwin	39	ct_ftp_cmd
10	rate	25	tcprtt	40	ct_flw_http_mthd
11	sttl	26	synack	41	ct_src_ltm
12	dttl	27	ackdat	42	ct_srv_dst
13	sload	28	smean	43	is_sm_ips_ports
14	dload	29	dmean		
15	sloss	30	trans_depth		

The aforementioned datasets are all suitable for attack detection; however, it is a critical consideration that the selected dataset must include the data collected during prior work. Comparing the datasets, UNSW-NB15 was chosen as it contains the required features and eliminates the need for additional computations.

The merging of datasets was also accomplished using a Python script, which is nearly identical to the one used for generating synthetic data.

For the previous program code, we also create an additional function named `get_traffic()`.

```python
def get_traffic(org, num=1500, t="attack"):
    nb15 = pd.read_csv('unsw_nb15.csv')
    nb15 = nb15[['ct_src_dport_ltm', 'ct_dst_sport_ltm', 'proto', 'sbytes',
        'attack_cat','state']]
    if t != "attack":
        traffic = nb15[nb15['attack_cat'] == 'Normal'].sample(n=num)
    else:
        traffic = nb15[nb15['attack_cat'] != 'Normal'].sample(n=num)
    traffic['id'] = [i for i in range(len(org), len(org) + len(traffic))]
    traffic['timestamp'] = [random_timestamp("2024-10-14 00:00:00",
        "2024-11-20 00:00:00") for i in range (len(traffic))]
    traffic['src_ip'] = ['192.168.0.138' for i in range (len(traffic))]
    traffic['dst_ip'] = ['192.168.0.100' for i in range (len(traffic))]
    traffic['message_content'] = [random.choice(org['message_content'])
        for i in range (len(traffic))]
    traffic['label'] = [t for i in range (len(traffic))]

    traffic['src_port'] = traffic['ct_src_dport_ltm']
    traffic['dst_port'] = traffic['ct_dst_sport_ltm']
    traffic['protocol'] = traffic['proto']
    traffic['length'] = traffic['sbytes']
    traffic['flag'] = traffic['state']

    traffic = traffic[['id','timestamp','src_ip','src_port','dst_ip',
        'dst_port','message_content','protocol','length','label']]

    return pd.DataFrame(traffic)
```

Source Code 14: Function for get data from the UNSW-NB15 dataset and merge the original

The `get_traffic` function is designed to generate attack data that can be added to an existing dataset. This function enables the augmentation of an existing dataset, either by filling in missing attack samples or by increasing the dataset size to enhance its utility for analysis. The function accepts three parameters: the original dataset, the number of samples to generate, and the type of traffic data to create (attack or normal).

The original dataset plays a critical role in ensuring the continuity of unique identifiers (`id`) for the newly generated samples. The UNSW-NB15 dataset is used as the source for necessary network attributes, along with other properties that are omitted during this analysis since they are not required for the current task. The dataset contains various attack types (e.g., DDoS or other attack categories), while the `Normal` label indicates the absence of an attack. The function simplifies this classification by focusing solely on distinguishing between attack and normal traffic samples. Based on this, the dataset is filtered, and a specified number of elements are randomly selected according to the input parameters.

The attributes of the generated data are configured in subsequent steps following filtering and random sampling. The `id` field is created based on the unique identifiers of the original dataset, ensuring a continuously increasing sequence. The `timestamp` values are randomly generated within a specific time range, while `message_content` values are randomly sampled from the original dataset. The source and destination IP addresses (`src_ip` and `dst_ip`) are assigned fixed values, reflecting the assumption that the attacks originate from a specific device (`192.168.0.138`). The `label` attribute is set based on the input parameter, enabling the generation of both attack and normal traffic labels.

Several attributes, such as the source port (`src_port`), destination port (`dst_port`), protocol (`protocol`), packet size (`length`), and state (`flag`), are extracted directly from the corresponding columns in the UNSW-NB15 dataset. These attributes are renamed to align with the naming conventions of the generated traffic dictionary, ensuring compatibility between the new dataset and the original structure. After the necessary keys are created, irrelevant attributes are removed, and the final transformed dataset is returned.

```
original_data = pd.read_csv('traffic.csv')
original_data['label'] = 'normal'
attacks = get_traffic(original_data)
augmented_data = pd.concat([original_data, attacks], ignore_index=True)
output_path = 'with_unsw_nb15_attacks.csv'
augmented_data.to_csv(output_path, index=False)
```

Source Code 15: The main part of the code has been modified

Only a minor adjustment is required in the main part of the program: the process of reading and labeling the original data remains unchanged, but the generation of attack data is now handled using the `get_traffic` function. Subsequently, the original data and the generated attack samples are merged into a unified dataset. The `with_unsw_nb15_attacks.csv` file contains these data.

To enhance the efficiency of model training, we aim to diversify the dataset by incorporating not only attack data but also normal traffic samples from the UNSW-NB15 dataset. To achieve this, the main part of the program needs to be extended as follows:

```
normals = get_traffic(original_data,1000,t= "normal")
augmented_data = pd.concat([augmented_data, normals], ignore_index=True)
output_path = 'augmented_with_unsw_nb15.csv'
augmented_data.to_csv(output_path, index=False)
```

Source Code 16: Added part to the main code

In this case, 1000 normal samples are extracted from the public dataset, resulting in approximately 60% more normal data compared to attack data. To maintain homogeneity within the dataset, it is necessary to increase the number of attack samples accordingly.

Before selecting the appropriate deep learning model, it is crucial to preprocess the dataset to ensure its suitability for the modeling task. This process begins by loading the data and transforming categorical fields or text-based columns into numerical representations, This transformation is essential for enabling machine learning models to interpret and process the categorical information effectively.

Subsequently, the most relevant features from the dataset should be identified and selected for use in the model. This includes determining the input features and the target labels, which serve as the basis for the model's learning and prediction tasks. Proper feature selection reduces the dimensionality of the data, minimizes noise, and focuses the model on the most informative aspects of the dataset.

Once the features and labels are defined, the dataset must be divided into training and testing subsets. This split ensures that the model is evaluated on unseen data, providing a reliable measure of its generalization capability. Typically, the dataset is partitioned so that a majority is used for training, while a smaller portion is reserved for testing. In our case, the dataset is partitioned with an 80:20 ratio, allocating 80% of the data for training and 20% for testing.

```
file_path = 'augmented_traffic_with_synthetic_attacks.csv'
data = pd.read_csv(file_path)

label_encoder = LabelEncoder()
data['label'] = label_encoder.fit_transform(data['label'])
data['protocol'] = label_encoder.fit_transform(data['protocol'])
data['flags'] = label_encoder.fit_transform(data['flags'])

features = ['src_port', 'dst_port', 'length', 'flags', 'protocol']
X = data[features]
y = data['label']

X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

Source Code 17: Pre-processing the data in python

2.3 Models

Deep learning models, inspired by the structure and function of the human brain, play a critical role in modern artificial intelligence applications due to their ability to capture complex patterns and representations in high-dimensional data. Their use is particularly prominent in network intrusion detection systems, where they excel at identifying anomalous traffic patterns and distinguishing between normal and malicious activity.

In Python, the implementation and utilization of deep learning methods have been greatly sim-

plified through the development of specialized libraries. These libraries provide user-friendly interfaces and pre-built functionalities for a wide range of machine learning and deep learning techniques. For instance, the widely used `scikit-learn` library includes implementations of several foundational machine learning methods, as well as tools for preprocessing, evaluation, and optimization. To use these methods, the corresponding classes and modules must be imported into the working environment. This step ensures that all the essential functionalities for the implementation and evaluation of our models are readily available. These imports form the backbone of the computational workflow, facilitating the seamless integration of deep learning techniques into the analytical pipeline.

```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.svm import OneClassSVM
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.metrics import classification_report, f1_score, make_scorer
```

Source Code 18: All of the necessary imports from scikit-learn

The `LabelEncoder` converts class labels into numerical values to make them processable for machine learning models, while the `StandardScaler` standardizes the data by centering it around zero with unit variance, thereby enhancing numerical stability. The `GridSearchCV` performs hyperparameter tuning with cross-validation to optimize model performance. The `RandomForestClassifier` is an ensemble learning algorithm that improves classification by utilizing a collection of decision trees, whereas the `GradientBoostingClassifier` is a gradient boosting algorithm that iteratively enhances classification using a series of weak learners. The `SVC` (Support Vector Classifier) learns the optimal hyperplane to separate different classes, and the `OneClassSVM` is an unsupervised learning model that identifies anomalies based on the learned distribution of normal samples. The `accuracy_score` is a metric for calculating classification accuracy, representing the proportion of correctly classified samples, while the `confusion_matrix` displays the relationship between true and predicted classes in a matrix format. The `classification_report` provides a detailed summary of multiple classification metrics, including precision, recall, and f1-score, and the `f1_score` measures the harmonic mean of precision and recall. Finally, the `make_scorer` enables the definition and application of custom metrics for model evaluation. These tools are essential for executing machine learning and data processing tasks effectively.

2.3.1 Supervised Learning

Supervised learning is a foundational approach in machine learning, where models are trained on labeled datasets to learn the mapping between input features and corresponding target outputs. This paradigm is widely utilized in applications requiring predictive accuracy, including classification, regression, and time-series forecasting tasks The expanded and labeled datasets created in section 2.2 were specifically prepared to align with the requirements of this methodology.

The effectiveness of a model can be evaluated using classification accuracy, which is a metric that calculates the ratio of correctly classified samples to the total number of samples. This is a straightforward yet crucial metric, as it indicates the percentage of correct predictions made by the model on the test dataset. The formula for accuracy is given by:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

While classification accuracy is widely used, it is not suitable for handling imbalanced classes, as it may underestimate the impact of minority classes. In the present study, care was taken to ensure that the dataset contained approximately balanced class distributions, mitigating this limitation.

Another important evaluation method is the confusion matrix, which provides a detailed analysis of classification results. The confusion matrix consists of the following four components:

- **True Positives (TP):** Cases where the model correctly predicts the positive class.
- **True Negatives (TN):** Cases where the model correctly predicts the negative class.
- **False Positives (FP):** Cases where the model incorrectly predicts the positive class.
- **False Negatives (FN):** Cases where the model incorrectly predicts the negative class.

The structure of the confusion matrix depends on the number of classes, but in the case of binary classification, it is represented as:

$$\begin{bmatrix} \text{TP} & \text{FP} \\ \text{FN} & \text{TN} \end{bmatrix}$$

The confusion matrix enables a detailed examination of the model's errors, which is particularly valuable for imbalanced datasets. By breaking down predictions into these components, the matrix provides clear insights into the error rates for each class, offering a comprehensive perspective on the model's performance.

Random Forest

The concept of random forests was introduced by Leo Breiman in his 2001 paper, ”Random Forests,” published in the journal Machine Learning [17]. In this seminal work, Breiman combined the principles of ”bagging” (bootstrap aggregating) and random feature selection to construct a collection of decision trees with controlled variance, thereby improving classification accuracy and robustness. In a random forest, each tree is built using a random subset of the training data and a random subset of features at each split, promoting diversity among the trees. This randomness leads to a model whose generalization error converges as the number of trees in the forest becomes large, effectively mitigating the risk of overfitting.

Random Forest is widely utilized in IDS for its robustness and ability to handle high-dimensional data [5]. This algorithm builds multiple decision trees and aggregates their outputs to improve accuracy. In cybersecurity, RF is popular for its ability to handle diverse data types and detect complex attack patterns effectively [7]. RF models excel in balancing high accuracy with computational efficiency, making them suitable for real-time applications.

Recent studies have shown that RF can achieve accuracy levels of 99.95% on certain datasets, such as CIC-IDS2017, and effectively differentiate between normal and malicious traffic [18]. Its capability to perform feature selection also aids in reducing dimensionality, thus enhancing detection speed without sacrificing precision. This makes RF especially beneficial for large-scale network monitoring [6].

```
param_grid = {
    'n_estimators': [10, 30, 50, 100, 200],
    'max_depth': [None, 10, 20, 30, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2', None]}
rf_model = RandomForestClassifier(random_state=42)
grid_search = GridSearchCV( estimator=rf_model, param_grid=param_grid,
                            scoring='accuracy', cv=5,
                            verbose=2, n_jobs=-1)
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
best_score = grid_search.best_score_
optimized_rf_model = RandomForestClassifier(**best_params, random_state=42)
optimized_rf_model.fit(X_train, y_train)
y_pred = optimized_rf_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
```

Source Code 19: Random Forest Classifier in Python

The hyperparameter optimization of a Random Forest classifier is critical for maximizing the efficiency and generalization ability of machine learning models. Proper tuning of hyperparameters directly impacts the model's performance, as they determine its structure and behavior. For instance, an insufficient number of decision trees (`n_estimators`) may result in a lack of flexibility, while too many trees can lead to increased computational costs. Similarly, an inappropriate selection of the `max_depth` parameter can result in overfitting or underfitting. Addressing these factors during optimization is essential to ensure the model's stability and accuracy.

The `GridSearchCV` systematically evaluates all possible combinations of hyperparameters on the training dataset while performing 5-fold cross-validation. The use of cross-validation during optimization minimizes the risk of overfitting by testing the model on different subsets of the data rather than solely on the training set. Consequently, the selected hyperparameters yield a model that performs well not only on the training set but also on unseen data. This is particularly critical in scientific research and industrial applications, where the ability of the model to generalize to new data is indispensable. Once the hyperparameter search is completed, a new Random Forest model is instantiated using the optimal parameters and is referred to as the optimized model. This model is retrained on the full training dataset. Subsequently, predictions are generated on the test dataset. The optimized model facilitates the evaluation of predictions using various metrics, such as accuracy or confusion matrices, ensuring a robust and reliable assessment of model performance.

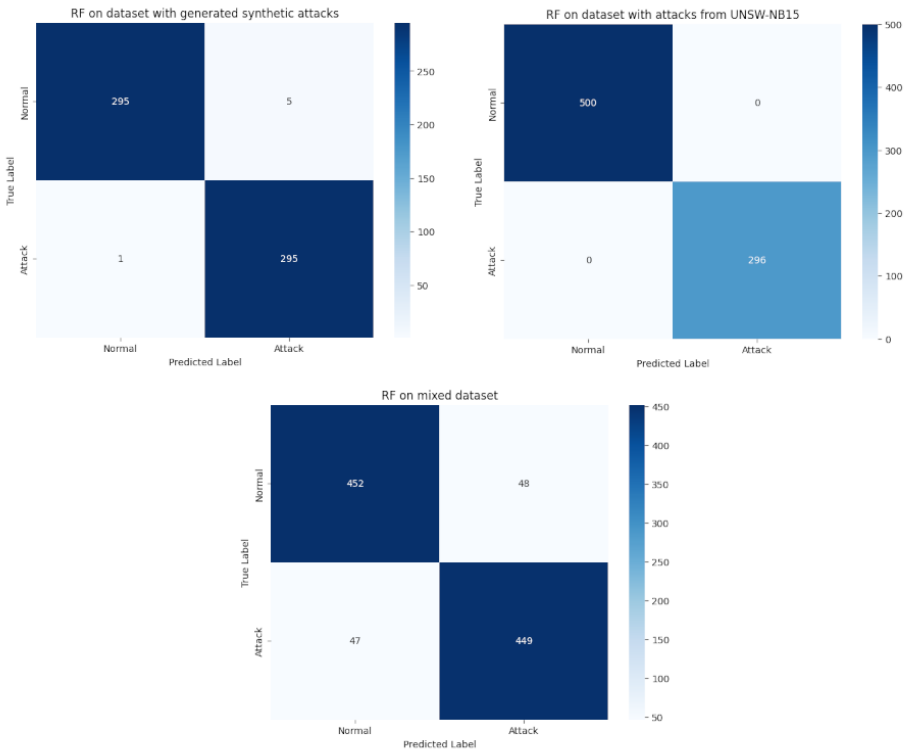

Figure 5: Confusion Matrices of the Random Forest models on each dataset

Support Vector Machine

Support Vector Machine is a classification-based ML algorithm often used in cybersecurity for detecting intrusions. SVM is highly effective for binary classification problems and performs well on small to medium-sized datasets. In IDS, it is used to distinguish between normal and malicious network activities by creating a hyperplane that separates data points into classes. SVM performs well in high-dimensional spaces, though it may struggle with large datasets due to computational complexity [19]. SVM has shown significant promise in reducing false positives, an essential feature in cybersecurity. SVM's performance is often augmented by combining it with feature reduction techniques like PCA to manage the high-dimensional data typically encountered in IDS [20].

Support Vector Machines aim to identify a hyperplane in the input space that separates data points belonging to different classes with the largest possible margin, thereby maximizing the model's generalization ability. This margin-based optimization allows SVMs to achieve robust performance even in complex classification scenarios.

One of the most significant features of SVMs is the kernel trick, which enables the implicit mapping of data into higher-dimensional spaces without explicitly computing the transformed coordinates. This property allows SVMs to effectively handle non-linear classification problems. Commonly used kernels include the linear, polynomial, radial basis function (RBF), and sigmoid kernels, each suited to different types of data distributions and problem characteristics.

The theoretical foundation of SVMs is closely tied to statistical learning theory, particularly the concept of Vapnik–Chervonenkis (VC) dimension [21]. The VC dimension quantifies the capacity of the model and provides insights into its generalization ability. Building on these theoretical principles, SVMs demonstrate strong generalization performance, even with relatively small training datasets. This capability makes SVMs a powerful tool in a wide range of machine learning applications.

```
svm_model = SVC(kernel='linear', random_state=42)
svm_model.fit(X_train, y_train)

y_pred = svm_model.predict(X_test)
```

Source Code 20: Support Vector Machine in Python

In the first step, an SVC model is initialized with a linear kernel. The linear kernel implies that the model attempts to distinguish between classes in the input space using a straight line or a plane. This choice is particularly suitable for linearly separable data, where the separation between different classes is well-defined.

In the next step, the model learns the hyperplane that separates the classes by training on the dataset (`X_train`) and its corresponding labels (`y_train`).

During the training process, the model identifies the support vectors, which determine the posi-

tion of the hyperplane, and calculates the maximum possible margin between the classes. This margin maximization enhances the model's generalization capability, which is particularly critical when the model encounters new, unseen data points.

Finally, the model makes predictions on the test dataset (`X_test`) based on the learned decision boundary.

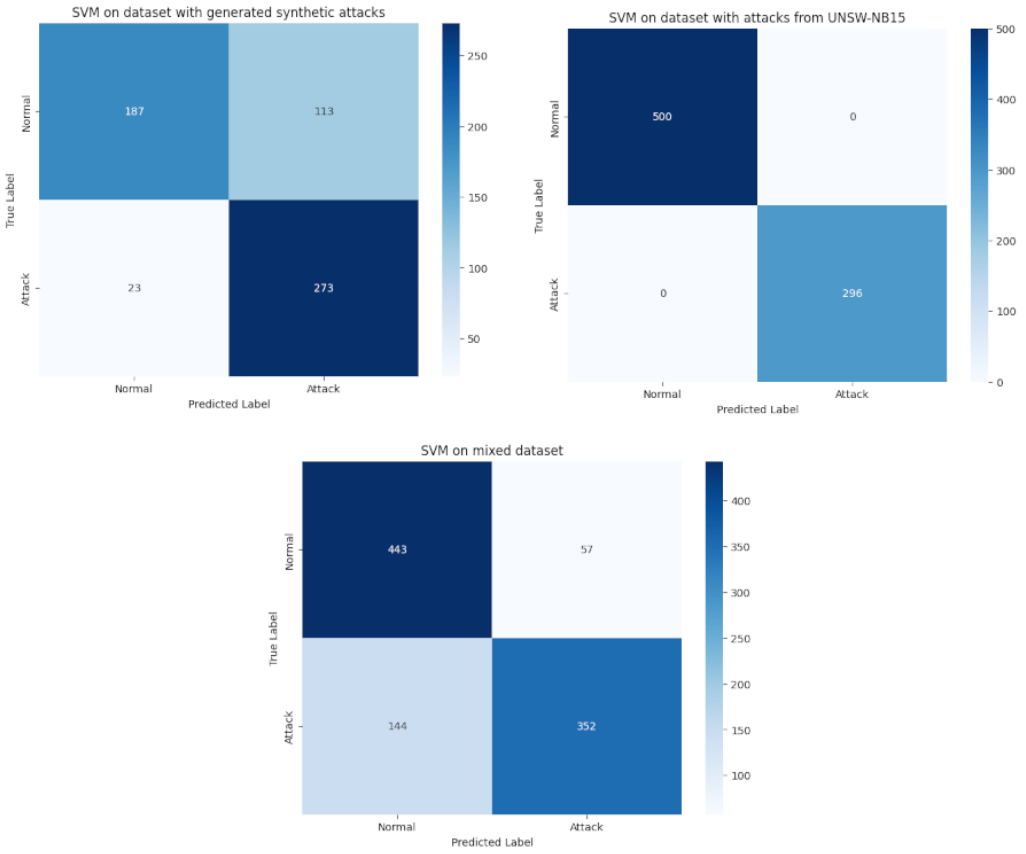

Figure 6: Confusion Matrices of the Support Vector Machine models on each dataset

Gradient Boosting Machine

The Gradient Boosting Machine (GBM) is a supervised learning algorithm that iteratively builds a strong predictive model as a linear combination of weak learners, typically decision trees. At each iteration, GBM attempts to correct the errors of previous models by fitting a new model in the direction of the negative gradient of the loss function. This approach enables the effective capture of complex, non-linear patterns in data.

Friedman (2001) provided a detailed introduction to the theoretical foundations and practical applications of GBM, highlighting its flexibility in employing different loss functions and weak learners [22]. Furthermore, Natekin and Knoll (2013) offered a comprehensive tutorial on the practical application of GBM, including parameter tuning and various use cases [23].

```
param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}

gbm = GradientBoostingClassifier(random_state=42)

grid_search = GridSearchCV(
    estimator=gbm,
    param_grid=param_grid,
    scoring='accuracy',
    cv=5,
    verbose=2,
    n_jobs=-1)

grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
best_score = grid_search.best_score_

print("Best Parameters:", best_params)
print("Best Cross-Validation Accuracy:", best_score)

optimized_gbm_model = GradientBoostingClassifier(**best_params,
   random_state=42)
optimized_gbm_model.fit(X_train, y_train)

y_pred = optimized_gbm_model.predict(X_test)
```

Source Code 21: Gradient Boosting Machine e in Python

This code snippet demonstrates the application of hyperparameter tuning for a Gradient Boosting Classifier using `GridSearchCV`, a systematic method for optimizing machine learning models, similar to the approach used for Random Forest classifiers. The `param_grid` dictionary defines a search space for the hyperparameters, including the number of estimators (`n_estimators`), learning rate (`learning_rate`), maximum tree depth (`max_depth`), minimum number of samples required to split a node (`min_samples_split`), and the minimum number of samples required to be at a leaf node (`min_samples_leaf`).

The `GridSearchCV` systematically evaluates all combinations of these hyperparameters on the training dataset using 5-fold cross-validation (`cv=5`). It uses accuracy as the evaluation metric to determine the optimal hyperparameter combination that maximizes the model's per-

formance. The parallelization of the search process (`n_jobs=-1`) ensures efficient computation.

The best hyperparameters and the corresponding cross-validated accuracy score are retrieved using `grid_search.best_params_` and `grid_search.best_score_`, respectively. These optimal parameters are then used to initialize a new Gradient Boosting Classifier, referred to as the optimized model. This model is trained on the entire training dataset. In the last step, the optimized model makes predictions on the test dataset (`X_test`) using `y_pred = optimized_gbm_model.predict(X_test)`.

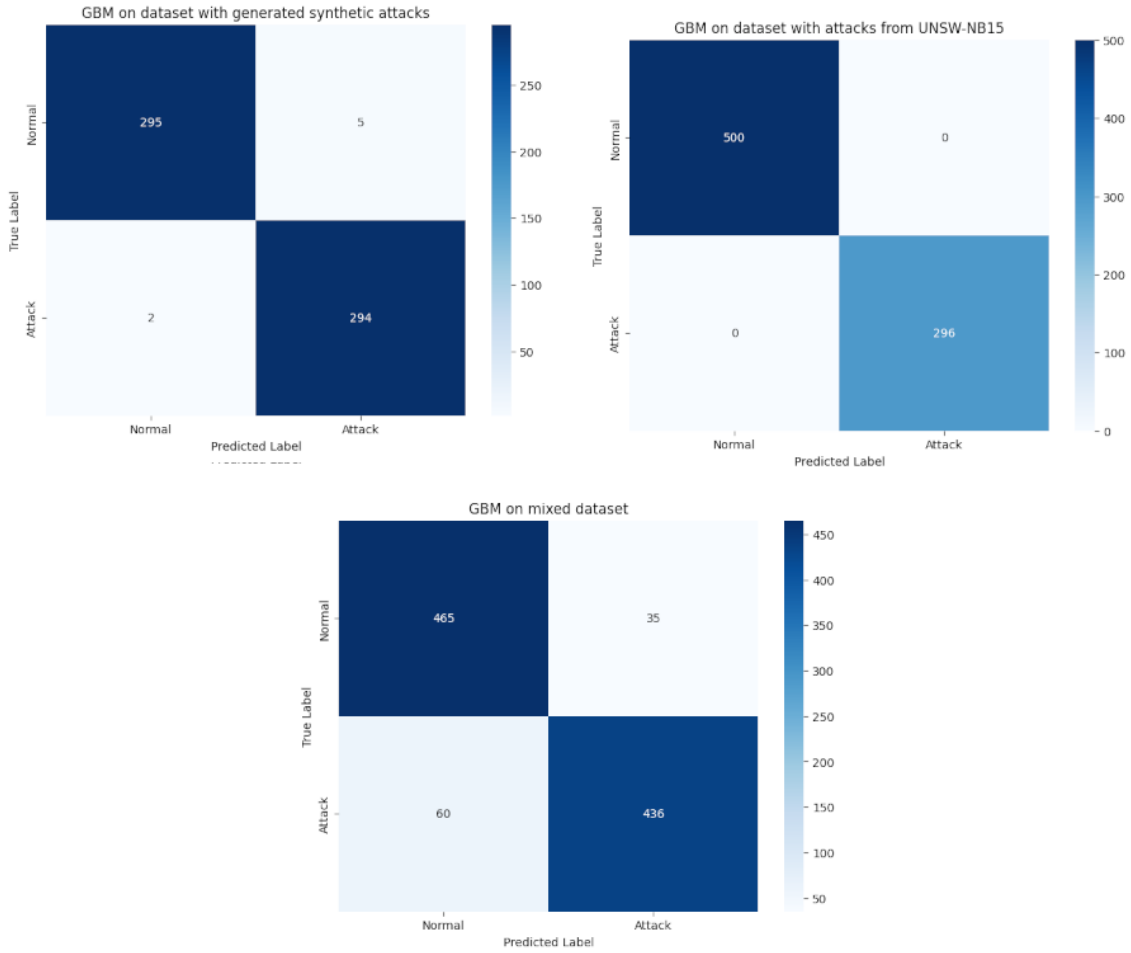

Figure 7: Confusion Matrices of the Gradient Boosting Machine models on each dataset

2.3.2 Semi-Unsupervised Learning

Semi-unsupervised learning is a hybrid machine learning paradigm that combines elements of both supervised and unsupervised learning [24]. Unlike traditional supervised learning, which relies entirely on labeled data, and unsupervised learning, which utilizes only unlabeled data, semi-unsupervised learning leverages a small amount of labeled data along with a large set of unlabeled data. This approach is particularly effective when labeled data is scarce or expensive

to obtain, but unlabeled data is abundant.

The primary goal of semi-unsupervised learning is to enhance the model's performance by leveraging the structure and distribution of the unlabeled data to inform the learning process [25]. Techniques in this domain often involve clustering, self-training, or pseudo-labeling, where the model generates its own labels for the unlabeled data during training. These methods help in capturing the underlying patterns and improving the generalization capabilities of the model.

Semi-unsupervised learning is widely applied in fields such as natural language processing, image recognition, and bioinformatics, where labeled datasets are often incomplete or imbalanced. Its ability to utilize both labeled and unlabeled data makes it a powerful tool for addressing real-world machine learning challenges.

This type of learning model is particularly suitable for our use case because the data we have collected consists exclusively of normal samples. If our goal is to work strictly with such data, it is essential to familiarize ourselves with these types of models to ensure effective and accurate analysis.

One-Class Support Vector Machine

The One-Class Support Vector Machine (OCSVM) is a specialized machine learning algorithm designed primarily for the detection of anomalies, novelties, or outliers. The objective of OCSVM is to identify data points that significantly deviate from normal samples, making it highly effective in domains such as fraud detection, fault diagnosis, and network intrusion detection. The algorithm learns the characteristics of normal data and constructs a decision boundary that separates normal samples from potential anomalies. Consequently, OCSVM is capable of detecting unusual or suspicious data that deviates from established patterns [26].

The approach begins with data preprocessing, where categorical features such as protocol and flags are encoded into numerical values to ensure compatibility with the machine learning algorithm [27]. Features like source and destination ports, packet length, and protocol-specific attributes are selected for model training, while missing values are removed to maintain data integrity.

Standardization is applied to the selected features to center them around zero and scale them to unit variance. This step is critical, as Support Vector Machines are sensitive to the scale of input features, and standardization ensures that all features contribute equally to the model.

The dataset is split into training and testing sets, with the training set comprising 80% of the data. To simulate anomalies in the test set, synthetic data points are generated within the range of the training data, mimicking potential deviations from normal patterns. Labels are assigned to differentiate between normal samples (labeled as 1) and anomalies (labeled as -1).

Hyperparameter tuning is conducted using GridSearchCV, a systematic method for optimizing model parameters. In this case, the parameters include gamma, which controls the influence

of individual data points in the RBF kernel, and nu, representing the expected proportion of anomalies in the data. The evaluation of parameter combinations is performed using 3-fold cross-validation to ensure robust selection. The scoring metric for this process is the F1 score, chosen for its balanced consideration of precision and recall, particularly important in imbalanced datasets like those encountered in anomaly detection.

The best-performing model, identified through the grid search, is used to predict anomalies in the test data. A detailed classification report evaluates the model's performance, providing metrics such as precision, recall, and F1 score for both normal and anomalous classes.

To effectively demonstrate the model's performance, it is essential to identify the two most influential features that significantly impact the model's predictions. By focusing on these features, we can gain deeper insights into the model's decision-making process and better interpret its behavior. For this purpose, the Python library SHAP (SHapley Additive exPlanations) can be utilized.

SHAP provides a unified framework for interpreting machine learning models by quantifying the contribution of each feature to a specific prediction. It is based on game theory and calculates Shapley values to attribute the impact of each feature on the model's output. Using SHAP, we can identify which features are most critical for the model's performance and how they influence the predictions. This allows for a transparent evaluation of the model and aids in uncovering underlying patterns in the data.

Visualizing the SHAP values can further enhance our understanding of the model. By plotting the feature importance and interaction effects, we can pinpoint the most impactful features and their relative contributions to the decision boundary. This process not only supports the evaluation of model efficiency but also provides a basis for feature selection, enabling the refinement of the dataset to improve model performance.

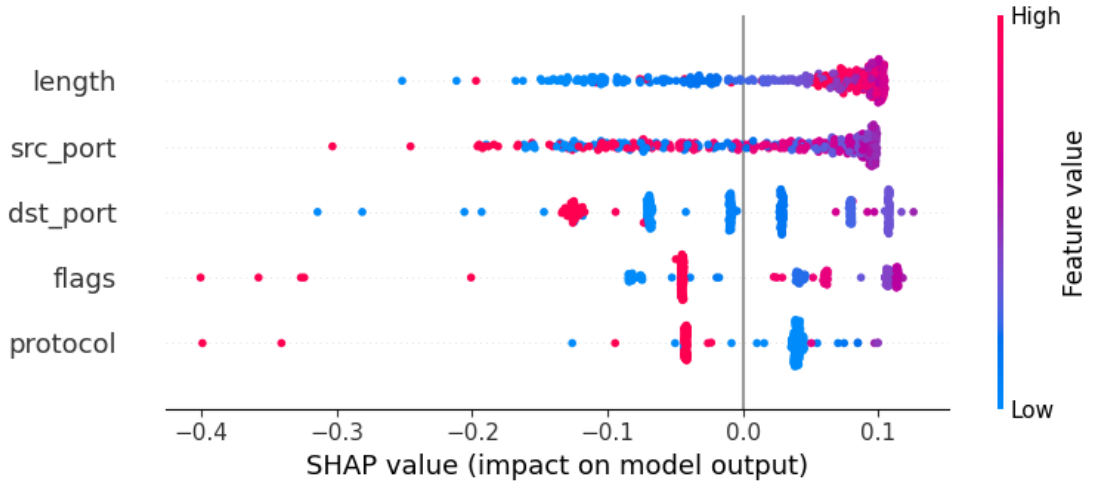

Figure 8: Feature Importance of the OCSVM model

From the plot, it is evident that the ”length” feature has the highest influence on the model’s predictions, as it exhibits the widest distribution of SHAP values. This suggests that variations in packet length significantly affect the model’s decision-making process. Additionally, ”src_port” and ”dst_port” also play a considerable role in influencing the model’s predictions, as indicated by their relatively broad SHAP value distributions.

The features ”flags” and ”protocol” have a comparatively smaller impact, as their SHAP value ranges are narrower. This implies that while they contribute to the model’s output, their influence is less critical compared to the primary features.

The color gradient, ranging from blue (low feature values) to red (high feature values), provides further insights into how feature values influence the SHAP values. For instance, in the ”length” feature, higher values (red points) tend to have a positive impact on the model’s predictions, while lower values (blue points) exert a negative influence. Similar patterns can be observed for ”src_port” and ”dst_port,” though their effects are less pronounced.

By selecting the two most influential features, ”length” and ”src_port”, identified through the SHAP analysis, the model is retrained using only these features to focus on the most impactful attributes. This approach ensures that the model concentrates on the features that contribute most significantly to its predictions, potentially improving its interpretability and computational efficiency.

After retraining, the model’s performance is evaluated, and its effectiveness is demonstrated through visual representations. These visualizations provide insights into the model’s behavior, highlighting how the selected features influence the decision-making process and validating the model’s capability to maintain high predictive performance with a reduced feature set. Such an approach not only refines the model but also aligns with best practices in feature selection, which aim to simplify the model while preserving or enhancing its accuracy.

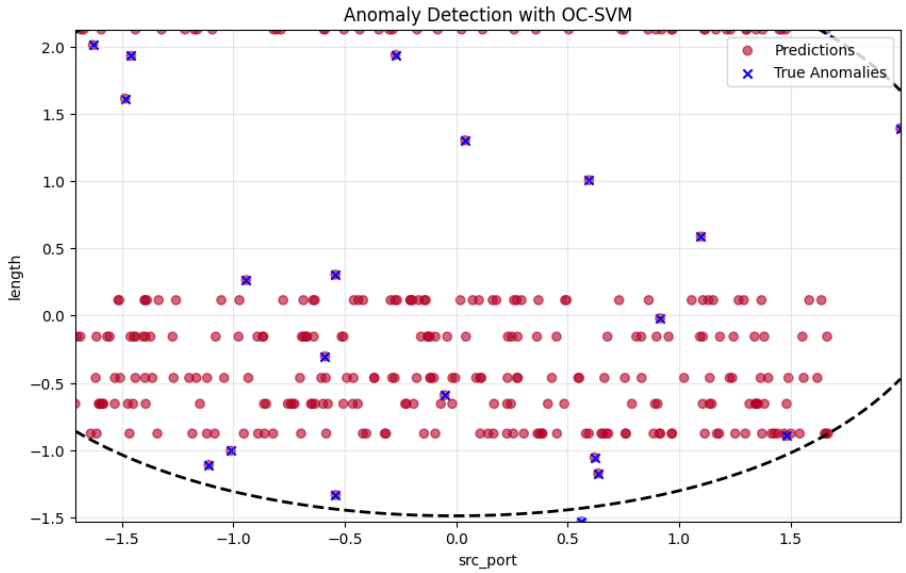

Figure 9: Visualization of the predicted anomalies

The black dashed curve represents the decision boundary learned by the OCSVM model. This boundary separates the normal data points, which fall within the curve, from potential anomalies, which lie outside of it. Points positioned outside the boundary are flagged as anomalies by the model.

From the visualization, it is evident that the model correctly identifies a significant proportion of true anomalies, as indicated by the blue crosses outside the boundary. However, some anomalies remain undetected, suggesting room for improvement in the model's sensitivity. The density and distribution of normal data points within the decision boundary further validate the model's ability to generalize and learn the patterns of the normal data effectively.

This analysis underscores the utility of the OCSVM for anomaly detection, particularly when focused on the most influential features, while highlighting the need for further optimization to improve detection rates for subtle anomalies.

2.3.3 Unsupervised Learning

Unsupervised learning is a machine learning paradigm that focuses on uncovering hidden patterns, structures, or relationships within unlabeled data. Unlike supervised learning, which relies on labeled datasets for training, unsupervised learning works solely with input data, making it particularly valuable in scenarios where labeled data is scarce or unavailable.

The primary objectives of unsupervised learning include clustering, where data points are grouped based on their similarities, and dimensionality reduction, which simplifies high-dimensional datasets while preserving essential information. These techniques are widely employed in various fields such as natural language processing, bioinformatics, and image processing, where they help to discover insights, detect anomalies, or preprocess data for further analysis.

Unsupervised learning is pivotal in real-world applications where the understanding of complex, unlabeled data is required. It provides a foundation for exploratory data analysis and can serve as a precursor to supervised learning tasks by generating pseudo-labels or reducing computational complexity. This adaptability makes unsupervised learning a cornerstone of modern data science and artificial intelligence research.

For the implementation of this technique, we utilized and imported the TensorFlow library, a widely adopted Python package designed for numerical computation and deep learning applications.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
```

Source Code 22: Importing tensorflow in Python

Autoencoders

Autoencoders are artificial neural networks designed to encode input data into a compressed latent representation and then reconstruct the original input from this representation, thereby capturing the intrinsic structures of the data [28]. They consist of two main components: an encoder, which maps the input to a lower-dimensional space, and a decoder, which reconstructs the original input from this compact representation. This process facilitates dimensionality reduction and the extraction of meaningful features.
In deep learning, autoencoders are used for anomaly detection by reconstructing data. When there is a large discrepancy between the original and reconstructed data, the instance is flagged as suspicious. Autoencoders are particularly effective for unsupervised learning in IDS, as they learn normal patterns and can detect deviations as potential threats [10].

The autoencoder model is constructed with an encoder-decoder architecture. The encoder maps the input data into a compressed latent representation, while the decoder reconstructs the input from this representation. The bottleneck layer in the middle captures the most essential features of the data, effectively reducing its dimensionality and highlighting the underlying structure. The model is trained using the mean squared error (MSE) loss function and the Adam optimizer, optimizing it to minimize the reconstruction error for normal data.
After training, the model's performance is evaluated by reconstructing the validation and test datasets. The reconstruction error for each data point is calculated as the mean squared difference between the original and reconstructed inputs. A threshold is determined based on the 95th percentile of reconstruction errors in the validation set, distinguishing normal data points from anomalies. Data points with reconstruction errors exceeding the threshold are classified as anomalies.
The reconstruction error distribution is visualized using histograms for both validation and test datasets. These plots illustrate the separation between normal data and potential anomalies, with a red dashed line indicating the decision threshold. This approach demonstrates the effectiveness of autoencoders in anomaly detection by leveraging the learned patterns of normal data and identifying deviations as anomalies.

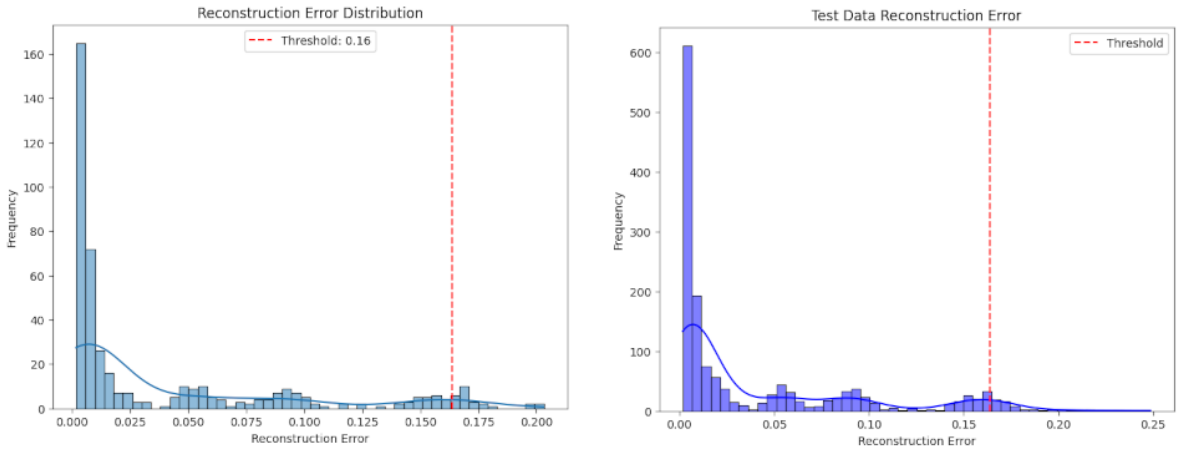

Figure 10: Reconstruction Error

The presented figures illustrate the reconstruction error distributions for the validation dataset (left) and the test dataset (right), as derived from the autoencoder model. The reconstruction error quantifies the difference between the original input and the reconstructed output produced by the autoencoder, with lower errors indicating that the input data aligns closely with the model's learned representation of normal data.

In the validation dataset, the histogram shows a significant concentration of data points with low reconstruction errors, indicating that the majority of the validation data is well-represented by the model. The red dashed line denotes the decision threshold, set at 0.16, based on the 95th percentile of the reconstruction errors from the validation data. This threshold effectively separates the majority of normal data points from potential anomalies. A small proportion of data points exhibit reconstruction errors exceeding this threshold, marking them as potential anomalies.

Similarly, the test data reconstruction error distribution displays a comparable pattern, with a high density of data points exhibiting low reconstruction errors. This similarity between the validation and test distributions demonstrates the model's ability to generalize well to unseen data. The same decision threshold (0.16) is applied to the test data, and points exceeding this value are classified as anomalies. The majority of the test data, however, remains below the threshold, confirming their classification as normal.

These results emphasize the autoencoder's effectiveness in capturing the patterns of normal data and its capability to distinguish anomalous data points based on reconstruction error. The clear separation around the threshold highlights the robustness of the model in anomaly detection tasks. The model demonstrates a strong ability to generalize from training data to validation and test datasets, as evidenced by the consistent reconstruction error distributions. To further evaluate the model's performance, quantitative metrics such as precision, recall, and F1-score can be used to provide a comprehensive assessment of its anomaly detection capabilities.

3 Results

3.1 The dataset

According to 2.1.1, we collected the following data in our database, these data was used to analyze the network between devices and train the models:

Table 6: Monitored Data

Field	Description
`id`	It is an identification number, which increases automatically when a new data line is added to the table. It is also a key value in our database to identify each element uniquely. This auto-increment feature ensures that each entry has a distinct ID, eliminating the risk of duplication and simplifying data retrieval and management. By using this identifier, we can establish relationships between different tables, enabling efficient cross-referencing and ensuring data integrity throughout our system.
`timestamp`	This identifier serves as an effective means of classification; however, its potential for duplication poses a risk of error, precluding its designation as a primary key. Instead, it is valuable for tracking events, such as indicating whether an attack occurred at a specific timestamp. In scenarios where a machine learning model operates within temporal data structures, incorporating this identifier becomes essential. By aligning the data with precise time points, the model can effectively analyze sequences and patterns over time, facilitating enhanced predictive accuracy in temporal analysis.
`src_ip`	The source IP address identifies the sender's logical IP address, which serves as essential information during an attack. Identification of the IP address enables the rapid localization of the attacking device or network, facilitating immediate response actions and allowing for appropriate countermeasures, such as blocking the attack or strengthening network security measures.

Continued on next page

Field	Description
dst_ip	The destination IP address indicates the logical address of the intended recipient of the packet. In our case, this information is relevant for two reasons: first, it allows us to identify the target device of the attack within the database, and second, if the packet is valid, it enables the device to forward the message to its intended recipient. Thus, the destination IP address plays a crucial role both in identifying the target of the attack and in ensuring the smooth handling of data traffic within the network.
protocol	The protocol type specifies the communication rules by which devices should process the message. This information not only supports anomaly detection but also assists in precisely identifying which specific vulnerability the potential attack exploits, enabling a faster and more targeted response. By identifying the protocol, the risk level of the affected communication channel can be more accurately assessed, and the effectiveness of the deployed defensive mechanisms can be evaluated.
src_port	The source protocol port number indicates the specific gateway through which communication occurs. Certain protocols may apply additional sub-rules, for which the use of port numbers has been introduced to define and manage these rules effectively. The use of port numbers is therefore crucial for targeted management of network traffic and differentiation between protocols, which is especially significant in security analysis.
dst_port	The destination protocol port number designates the specific gateway of the target device through which communication arrives. Similar to the source port number, this port number is part of the protocol's rule set; however, its validity is established upon the arrival of the packet. The destination port number thus plays a crucial role in receiving and directing the packet to the appropriate device, enabling more efficient and secure management of network traffic.
length	The size of a packet is a crucial factor in network traffic analysis. Suspiciously small or unusually large packets often indicate anomalies, which may suggest potential attacks. Therefore, monitoring packet size is essential for the timely identification and mitigation of possible network threats.

Continued on next page

Field	Description
`flags`	This message-specific identifier precisely designates the type of packet and, in the event of an error, signals the type and status of the issue. This allows for accurate monitoring and management of data transmission, thereby facilitating seamless error detection and resolution.
`message_content`	The content of the packet, or message, plays a critical role in subsequent stages of data processing. During post-processing, a thorough and precise analysis of this message enables the efficient execution of further data management operations and the extraction of essential information.

Table 7: Collected data with Raspberry PI - 1

id	timestamp	src_ip	dst_ip	protocol
281	2024.10.28 15:54	192.168.1.138	192.168.1.100	UDP
282	2024.10.28 15:54	192.168.1.138	192.168.1.100	UDP
283	2024.10.28 15:54	192.168.1.138	192.168.1.100	UDP
284	2024.10.28 15:54	192.168.1.138	192.168.1.100	TCP
285	2024.10.28 15:54	192.168.1.138	192.168.1.100	TCP
286	2024.10.28 15:54	192.168.1.138	192.168.1.100	UDP

Table 8: Collected data with Raspberry PI - 2

src_port	dst_port	length	flags	message_content
62792	161	147		G-code: G1 X100 Y100 Z0.2 ; Move to next layer (Data packet received)
62791	123	156		G-code: G1 X100 Y100 Z0.2 ; Move to next layer (Data packet received)
60779	123	184		G-code: G1 X100 Y100 Z0.2 ; Move to next layer (Data packet received)
50897	443	49	SYN	G-code: G28 ; Home all axes (Handshake initiated)
60488	443	173	SYN-ACK	G-code: G28 ; Home all axes (Handshake initiated)
62500	123	167		G-code: G1 X100 Y100 Z0.2 ; Move to next layer (Data packet received)

The communication presented in the tables outlines the initial steps involved in 3D printing a simple pyramid shape. It includes defining the necessary communication parameters to initiate the printing process, as well as methods for addressing potential issues that may arise in the early stages of printing. The program installed on the device transmits the packet between devices by defining these data parameters. In the future, particularly in scenarios involving data

stream monitoring, this program can be further developed to operate as an NID system under the influence of deep learning models. This would enable it to filter out unwanted data and notify the user of any unauthorized access attempts.

3.2 Comparison of the models

In Section 2.3, we constructed various models belonging to the supervised learning category. To make these methods applicable, the data had to be labeled and supplemented with attack data, enabling a binary classification (normal or attack). Once the attack data were integrated, it was necessary to select the features that could significantly influence the models. For each case, the selected features included src_port, dst_port, length, protocol, and flag. Using the same features across all models ensured a fair comparison of their performance.

The next step involved transforming text-based attributes into numerical representations using LabelEncoder, allowing the machine learning models to process them effectively. The datasets were divided into four subsets: two main parts—training and testing sets—and, within these, separate subsets for the features and their corresponding labels.

Following this, the models were initialized and trained. Where applicable, hyperparameter tuning was employed to optimize their performance. Once trained, the models were evaluated using various metrics to assess their effectiveness. Most evaluations were conducted using accuracy score as the primary metric, complemented by the confusion matrix, whose calculation methodology is detailed in 2.3.1. This systematic approach ensured a rigorous and reproducible evaluation of the models.

Table 9: Accuracy score of the models

Model	Synthetic Generated	Attacks from UNSW-NB15	Mixed with UNSW-NB15
RF	0.9899	1.0	0.9046
SVM	0.7718	1.0	0.7981
GBM	0.9882	1.0	0.9046

The table clearly demonstrates that the Random Forest Classifier and Gradient Boosting Machine performed exceptionally well across all three datasets, particularly on the synthetic and mixed datasets, where their results were remarkably similar. In contrast, the Support Vector Machine underperformed compared to the other two models on the synthetic and mixed datasets, suggesting that it may be less robust in handling these types of data.

However, a closer examination of the table reveals a potential issue with the models' performance on the dataset where normal data were collected from our custom data sources while attack data were sourced from the UNSW-NB15 database. All models exhibited extremely high performance on this dataset, which raises concerns about overfitting or excessive distinguisha-

bility between the normal and attack data. This phenomenon suggests that the models may be overly specialized in differentiating between the two classes, potentially compromising their generalization capabilities on unseen, more realistic data distributions.

To address this issue, it is necessary to delve deeper into the inner workings of these models to identify the factors contributing to this behavior. Specifically, we should analyze which features exert the most influence on the models' decision-making processes. By examining the feature importance and their impact on the predictions, we can gain valuable insights into the reasons behind the models' overly efficient performance and evaluate potential steps for mitigating these challenges, such as feature engineering, data augmentation, or adjusting the models' hyperparameters.

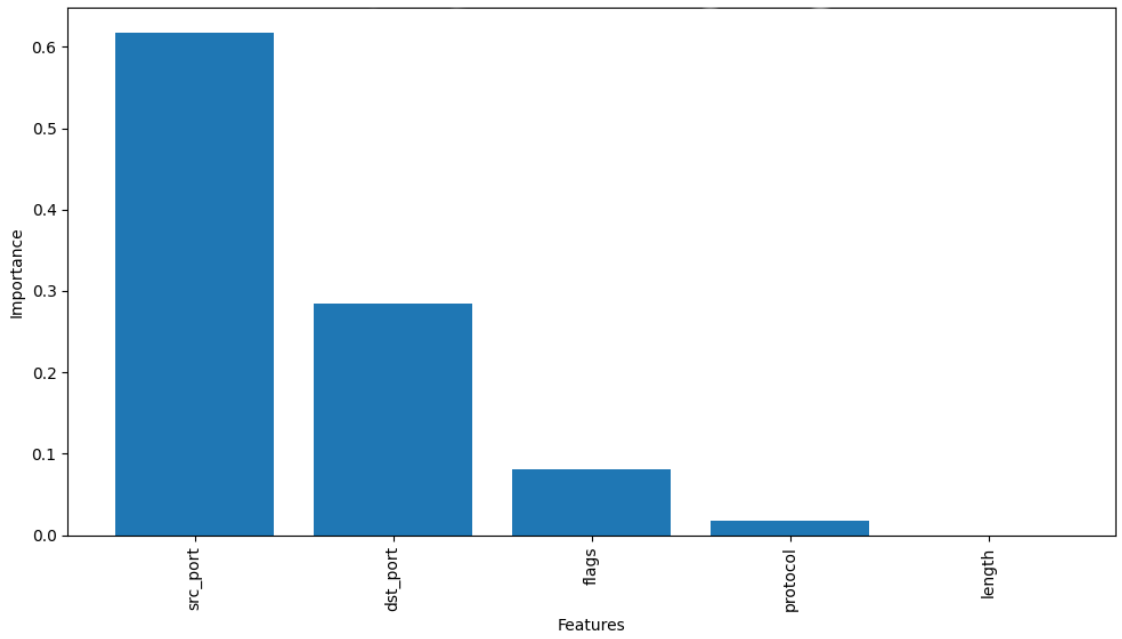

Figure 11: Feature importance of the dataset where attacks were from the UNSW-NB15 database

The analysis of the graph reveals that the most influential factor for the model is the src_port attribute associated with each traffic flow, which exerts twice the influence compared to the next most significant feature. This indicates that the source ports within the dataset play a crucial role in distinguishing between normal and attack traffic. The high discriminative power of this feature suggests a strong correlation between specific source port values and the type of traffic, enabling the model to effectively differentiate between the two categories.

This observation implies that the dataset's source port distribution is not uniform and contains patterns or anomalies that are highly indicative of traffic type. For instance, if two randomly selected traffic flows with different labels (normal and attack) are compared, the src_port attribute is likely to account for a significant portion of the variation between them. Such a finding underscores the importance of this feature in the model's decision-making process and highlights its role as a key driver of classification accuracy.

Table 10: Randomly choosed source ports and these labels

id	label	src_port
1480	normal	63210
1481	attack	25

To understand this phenomenon, it is essential to examine the port numbers associated with each instance of network communication. Port numbers play a crucial role in networking, acting as logical endpoints that help differentiate between various applications or services running on a single device. They allow multiple types of traffic to coexist and communicate efficiently within a network.

Port numbers are defined within a 16-bit range, spanning from **0 to 65535**, and are categorized into three main groups:

- **Well-Known Ports (0–1023)**: These ports are reserved for commonly used services and protocols, such as HTTP (port 80), HTTPS (port 443), FTP (port 21), and SSH (port 22). They are standardized and globally recognized, ensuring consistent behavior across different systems and networks.
- **Registered Ports (1024–49151)**: These ports are used by specific applications or services that are registered with the Internet Assigned Numbers Authority (IANA). Examples include database systems like MySQL (port 3306) or gaming servers. While not as strictly regulated as well-known ports, these are often associated with specific applications.
- **Dynamic or Private Ports (49152–65535)**: Often referred to as ephemeral ports, these are dynamically allocated by operating systems for temporary or client-side communication. For instance, when a user accesses a website, the browser may use an ephemeral port to establish the connection.

From this, we can infer that the 3D printer used in our setup communicates through private ports, while attackers tend to initiate their actions via open, well-known ports to ensure the success of their attacks. However, given the continuous evolution of attackers and their methods, it cannot be assumed that they only exploit well-known ports. Advanced attackers are increasingly leveraging unconventional approaches, including targeting private ports, making it critical to develop comprehensive defenses that account for a variety of attack vectors.

In terms of enhancing the dataset, we have already experimented with incorporating normal data from the UNSW-NB15 dataset. Another potential avenue for improvement involves the synthetic generation of additional attack data to supplement the existing samples. Moreover, a more advanced approach could involve conducting controlled attacks on the 3D printer itself to gather realistic and diverse attack data. This would provide a more robust foundation for training machine learning models capable of identifying sophisticated threats.

The types of attacks that could be simulated are numerous and vary in scope. At one end, attacks might aim to render the 3D printer inaccessible to other devices, effectively isolating it from the network. On the other end of the spectrum, attacks could escalate to severe actions, such as causing physical damage to the printer. Examples include:

- **Denial of Service (DoS) attacks:**: Flooding the printer's communication channel with excessive requests, making it unresponsive to legitimate commands.
- **Unauthorized firmware modification:**: Introducing malicious code to alter the printer's behavior, potentially leading to flawed outputs or hardware malfunctions.
- **Sabotage of critical components:**: Manipulating motor controls or heating elements to overheat or damage the printer's internal systems.

These types of attacks can significantly disrupt manufacturing processes, resulting in operational delays and financial losses for the factory. As the manufacturing industry increasingly relies on connected devices, the potential impact of such threats cannot be overstated. Understanding and simulating these scenarios is crucial for developing resilient security measures and ensuring the integrity of the production process.

Among the models we developed, those trained on synthetically generated datasets demonstrated the highest suitability for the classification tasks at hand. However, an important question arises: would these models perform equally well in real-world scenarios, where actual attacks are carried out on networked systems? This remains an open area of exploration, as our current study lacks access to real-world attack data collected from live systems. Addressing this limitation could represent a valuable direction for future research, wherein attack data could be recorded in a controlled, simulated environment. Such an approach would provide a more accurate and realistic basis for evaluating the practical applicability of these models.

In the context of semi-unsupervised and unsupervised learning methodologies, it is noteworthy that these approaches do not require labeled attack data for training. Instead, they rely on the patterns present in normal traffic to learn a baseline model of typical behavior. However, testing and evaluating such models necessitate the availability of attack data, as these are required to assess the model's ability to detect anomalies effectively. In our study, the attack data were also generated synthetically to simulate abnormal behavior. This synthetic generation was applied consistently across both semi-unsupervised and unsupervised models to ensure standardization. While the synthetic generation of attack data provides a useful proxy for evaluation, it introduces certain limitations. Synthetic data may fail to capture the complexity and diversity of real-world attack patterns, potentially leading to overoptimistic results in controlled environments. Thus, transitioning from synthetic datasets to real-world attack data is crucial for advancing the robustness and reliability of anomaly detection models. This transition could involve designing experimental setups where networked devices, such as 3D printers or industrial equipment, are

intentionally subjected to controlled attacks to collect realistic data.

Furthermore, semi-unsupervised and unsupervised models present unique advantages in environments where labeled attack data are scarce or infeasible to obtain. By leveraging only normal data for training, these models align well with real-world scenarios where anomalous events are rare and poorly documented. Nonetheless, as the models are validated and tested, the availability of well-curated attack data becomes indispensable for evaluating their effectiveness under diverse and realistic conditions. Therefore, future research should focus on the generation, collection, and use of real-world attack data to complement the synthetic approaches employed in this study. Such efforts will enhance the practical utility of these models in identifying sophisticated threats and protecting critical systems.

4 Conclusions

This study emphasizes the critical role of data collection in enhancing the effectiveness of Network Intrusion Detection Systems in increasingly interconnected environments. By monitoring network traffic between industrial devices, such as CNC machines and IoT-enabled robots, we have constructed a comprehensive dataset that captures essential parameters, including IP addresses, protocol types, packet size, and anomalous flags. The Raspberry Pi-based network monitoring system enabled efficient real-time data acquisition, leveraging protocols specific to industrial communication. These data are instrumental for training machine learning models that detect and respond to network anomalies, thereby bolstering defenses against cyber threats. The precision and diversity of the collected dataset are essential for developing machine learning models capable of identifying zero-day vulnerabilities and complex attack patterns. This data-driven approach to cybersecurity allows for timely detection and response to potential threats, minimizing risks to critical infrastructure. In future developments, expanding the dataset with more complex features and integrating additional industrial protocols will further enhance the accuracy and applicability of machine learning-enhanced NIDS. The study demonstrates that robust, resource-efficient data collection processes are foundational to modern cybersecurity frameworks, particularly in safeguarding industrial systems in an era of advanced cyber threats.

This study also emphasizes the pivotal role of machine learning models in enhancing Network Intrusion Detection Systems. Through the integration of data from synthetic attack generation and publicly available datasets like UNSW-NB15, the project effectively demonstrated the classification capabilities of multiple supervised learning algorithms. The Random Forest classifier consistently showcased exceptional performance across various datasets, achieving near-perfect accuracy, especially when trained on synthetic and mixed datasets. This result highlights RF's robustness and adaptability to diverse data characteristics, making it a strong candidate for real-world deployment. The Gradient Boosting Machine performed comparably

to RF, reinforcing its utility in capturing complex non-linear relationships within network traffic data. Its results validate its suitability for environments where precision and adaptability are crucial. Conversely, the Support Vector Machine displayed moderate performance, particularly excelling in datasets with balanced attack and normal traffic distributions. However, its limitations in scalability with large datasets underline the need for further optimization or hybridization with other methods. Overall, the experimental results underline the viability of these machine learning models in real-time NIDS applications. Future research should focus on validating these models with real-world attack scenarios and exploring deep learning techniques to further enhance detection capabilities. Expanding the dataset and incorporating additional industrial communication protocols will be critical to addressing emerging cyber threats and ensuring robust industrial system security.

Acknowledgments

The work was prepared for the courses Digital Factory Lab I (IPMSEK-22fi4DFLAB1) and Digital Factory Lab II (IPMSEK-22fi4DFLAB2), under the supervision of Béla J. Szekeres, PhD, and Mátyás Andó, PhD.

References

[1] Janusz Pochmara and Aleksandra Świetlicka. Cybersecurity of industrial systems—a 2023 report. *Electronics*, 13:1191, 03 2024.

[2] Shahid Alam. Cybersecurity: Past, present and future, 07 2022.

[3] Seth E. Smith. An analysis of significant cyber incidents and the impact on the past, present, and future. Technical report, Old Dominion University, December 2021. Cybersecurity Undergraduate Research, Fall 2021.

[4] Alessandro Di Pinto, Younes Dragoni, and Andrea Carcano. Triton: The first ics cyber attack on safety instrument systems. In *Black Hat USA 2018 - Research Paper*. Nozomi Networks, 2018.

[5] Halil İbrahim Coşar, Çağrı Arısoy, and Hasan Ulutaş. Intrusion detection on cse-cic-ids2018 dataset using machine learning methods. *Artificial Intelligence Theory and Applications*, 4(2):143–154, 2024.

[6] Chaofei Tang, Nurbol Luktarhan, and Yuxin Zhao. An efficient intrusion detection method based on lightgbm and autoencoder. *Symmetry*, 12:1458, 09 2020.

[7] Md. Alamin Talukder, Md. Manowarul Islam, Md Ashraf Uddin, Khondokar Fida Hasan, Selina Sharmin, Salem A. Alyami, and Mohammad Ali Moni. Machine learning-based network intrusion detection for big and imbalanced data using oversampling, stacking feature embedding and feature extraction, 2024.

[8] Man Ni. A review on machine learning methods for intrusion detection system. *Applied and Computational Engineering*, 27:57–64, 12 2023.

[9] Vinayakumar Ravi, Mamoun Alazab, Soman Kp, Sriram Srinivasan, Sitalakshmi Venkatraman, Viet Pham, and Simran Ketha. Deep learning for cyber security applications: A comprehensive survey, 10 2021.

[10] Miel Verkerken, Laurens D'Hooge, Tim Wauters, Bruno Volckaert, and Filip De Turck. Unsupervised machine learning techniques for network intrusion detection on modern data. 11 2020.

[11] Nour Moustafa and Jill Slay. The significant features of the unsw-nb15 and the kdd99 data sets for network intrusion detection systems. pages 25–31, 11 2015.

[12] Naveen Bindra and Manu Sood. Evaluating the impact of feature selection methods on the performance of the machine learning models in detecting ddos attacks. *Romanian Journal of Information Science and Technology*, 3:250–261, 01 2020.

[13] Raja Raja Mahmood, AmirHossien Abdi, and Masnida Hussin. Performance evaluation of intrusion detection system using selected features and machine learning classifiers. *Baghdad Science Journal*, 18:0884, 06 2021.

[14] Zulu Okonkwo, Ernest Foo, Zhe Hou, Qinyi Li, and Zahra Jadidi. Encrypted network traffic classification with higher order graph neural network. In Leonie Simpson and Mir Ali Rezazadeh Baee, editors, *Information Security and Privacy*, pages 630–650, Cham, 2023. Springer Nature Switzerland.

[15] Yan Zhou, Huiling Shi, Yanling Zhao, Wei Ding, Jing Han, Hongyang Sun, Xianheng Zhang, Chang Tang, and Wei Zhang. Identification of encrypted and malicious network traffic based on one-dimensional convolutional neural network. *Journal of Cloud Computing*, 12(1):53, April 2023.

[16] Hoang Thanh and Lang Tran. An approach to reduce data dimension in building effective network intrusion detection systems. *EAI Endorsed Transactions on Context-aware Systems and Applications*, 6:162633, 07 2018.

[17] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[18] Faisal Nabi and Xujuan Zhou. Enhancing intrusion detection systems through dimensionality reduction: A comparative study of machine learning techniques for cyber security. *Cyber Security and Applications*, 2:100033, 2024.

[19] Yasmeen Almutairi, Bader Alhazmi, and Amr Munshi. Network intrusion detection using machine learning techniques. *Advances in Science and Technology Research Journal*, 16:193–206, 07 2022.

[20] Marzia Zaman and Chung-Horng Lung. Evaluation of machine learning techniques for network intrusion detection. pages 1–5, 04 2018.

[21] Alexey Ya. Chervonenkis. *Early History of Support Vector Machines*, pages 13–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[22] Jerome H Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[23] Alexey Natekin and Alois Knoll. Gradient boosting machines, a tutorial. *Frontiers in Neurorobotics*, 7:21, 2013.

[24] Jesse E. Van Engelen and Holger H. Hoos. A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440, 2020.

[25] Xiaojin Yang, Zhiqiang Song, Irwin King, and Zenglin Xu. A survey on deep semi-supervised learning. *IEEE Transactions on Neural Networks and Learning Systems*, 32(12):5425–5446, 2021.

[26] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alexander J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1471, 2001.

[27] Tamara Soltész. Network intrusion detection (github repository), 2024.

[28] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *arXiv preprint arXiv:2003.05991*, 2020.