Federated Certificate- and Behavior-Based Malware Traffic Detection



Isaac Silva
Master in Information Security
Department of Computer Science
Faculty of Sciences of the University of Porto
2025

Federated Certificate- and Behavior-Based Malware Traffic Detection

Isaac Silva

Dissertation carried out as part of the Master in Information Security

Department of Computer Science

2025

Supervisor

Bernardo Luís Fernandes Portela, Assistant Professor, Faculty of Sciences of the University of Porto

Acknowledgements

I am extremely grateful to my thesis supervisor, Bernardo Portela, whose guidance and expertise elevated this work and made the process both enriching and enjoyable. I would also like to sincerely thank Carlos Novo for his invaluable support throughout this thesis, which was fundamental to its successful development.

Secondly, I am grateful to the Faculty of Sciences for facilitating and supporting the development of this thesis, including assistance with the underlying infrastructure.

Thank you to my mother, father, and brother for your unwavering love and encouragement. Thank you, Cláudia, my girlfriend, whose trust helped me through the difficult times. And to my closest friends, Manel, Pedro Santos, Francisco and Pedro Vilar, and everyone else who's been there for me, thank you.

Resumo

Os ataques cibernéticos têm-se tornado cada vez mais prevalentes, acompanhados por um aumento significativo da sua sofisticação. Em particular, tem-se observado que malware recorre a infraestruturas de *Command and Control* para estabelecer comunicações persistentes entre o sistema comprometido e a infraestrutura maliciosa. Este tipo de comunicação tem sido frequentemente estabelecida através de protocolos criptográficos, como o TLS 1.2 e o TLS 1.3, tornando a sua deteção mais complexa. Os sistemas de deteção de intrusões têm como objetivo identificar padrões indicativos de atividade maliciosa. Esses sistemas são, em geral, construídos a partir de modelos de aprendizagem automática (machine learning) treinados com dados que, predominantemente, se encontram armazenados em infraestruturas centralizadas. Não obstante, tal abordagem pode constituir uma limitação da capacidade de deteção, dado que certos padrões relevantes podem não estar refletidos nos dados de treino, uma vez que diferentes entidades observam diferentes tipos de tráfego.

Uma potencial solução para a diversificação é a partilha de datasets entre diferentes clientes de modo a que todos possuam um dataset global. Porém, essa partilha torna-se inviável devido a questões legais e de privacidade relacionadas ao elevado nível de sensibilidade dos dados. Para mitigar essa limitação, surge um novo paradigma, aprendizagem federada (Federated Learning), que permite a combinação de diferentes fontes de dados sem que os mesmos sejam partilhados. Desta forma, Federated Learning permite a criação de modelos mais robustos e generalizados, capazes de responder a uma maior variedade de atividades maliciosas.

Nesta tese, procedeu-se à implementação de um modelo que permite a múltiplos clientes colaborarem no treino de um modelo global. Os nossos resultados demonstram a boa performance deste modelo, na medida em que, para certas métricas, este atinge melhores resultados comparativamente a modelos construídos em ambientes centralizados. Adicionalmente, avaliamos a resiliência de Federated Learning face a ataques de envenenamento de dados, tendo-se verificado a sua não resiliência. Para mitigar tal exposição a ataques, esta tese foi estendida com a incorporação e o desenvolvimento de mecanismos que tornam o modelo mais robusto contra manipulação maliciosa de dados.

Palavras-chave: Sistemas de detecção de intrusão, Aprendizagem federada, Aprendizagem automática, Redes neuronais, Segurança, Malware, C2, Ataques de Envenenamento

Abstract

In recent years, cyber threats have not only become more widespread but have also evolved in sophistication, posing increasingly complex challenges to security systems and defense strategies. In particular, it has been observed that malware leverages *Command and Control* infrastructures to establish persistent communications between the compromised system and the malicious infrastructure. This type of communication has often been carried out through cryptographic protocols such as TLS 1.2 and TLS 1.3, which, due to their inherent encryption, makes detection considerably more complex. Intrusion Detection systems attempt to uncover such activity by identifying patterns that may indicate malicious behavior. These systems often rely on machine learning models trained on data stored in a single location. Nevertheless, such an approach can pose a limitation to detection capabilities, since certain relevant patterns may not be reflected in the training data, as different entities observe different types of traffic.

A potential solution is to share datasets across different parties and create a unified dataset, however this creates legal and privacy concerns, as sensitive information could be exposed. To overcome this challenge, a new paradigm, Federated Learning, allows for different datasets to be combined without the need to share the datasets themselves. By leveraging distributed datasets, federated learning can create more robust and generalized models capable of detecting a wider variety of threats.

In this thesis, we design and implement such a system, allowing multiple parties to collaboratively train a global and transversal machine learning model. Our results demonstrate that the federated learning model achieves strong performance and, in certain metrics, even outperforms traditional centralized approaches. Furthermore, we evaluate the resilience of the decentralized framework and show that standard federated learning is vulnerable to simple data poisoning attacks. To address this, we extend our work by incorporating defense mechanisms that enhance the robustness of the system against adversarial manipulation.

Keywords: IDS, Federated Learning, Machine Learning, Neural Network, Security, Malware, C2, Poisoning attacks

Table of Contents

List of Tables ix
List of Figures xi
List of Abbreviations xiii

1. Introduction 1
   1.1. Proposed Approach 2
   1.2. Document Structure 3
2. State of The Art 5
   2.1. Malware Traffic Detection 5
   2.2. Machine Learning Based Malware detection 6
      2.2.1. Command and Control Detection 7
      2.2.2. Evaluating Network Traffic Classifiers 8
      2.2.3. Limitations of Centralized Machine Learning Models 9
   2.3. Neural Network Models 9
      2.3.1. Structure 11
      2.3.2. Learning 11
   2.4. Federated Learning and Security Considerations in C2 Detection Paradigm 12
      2.4.1. Selection of Machine Learning Models for Federated Learning 15
      2.4.2. Federated Learning in Malware Detection 15
      2.4.3. Federated Learning Threats 16
      2.4.4. Security and Robustness 19
      2.4.5. Critical Learning Period 22
3. Neural Network Model Construction 23
   3.1. Datasets 23
   3.2. Neural Network Structure and Hyperparameter Selection 24
4. Federated Learning Model 31
   4.1. System Model 32
   4.2. Partitioning the Dataset 32
   4.3. Standard Implementation 33
   4.4. Custom Implementation 35
      4.4.1. Broadcast 36
      4.4.2. Local Client Training 36
      4.4.3. Model Updates Aggregation 37
   4.5. Adversarial Scenarios 38
      4.5.1. Data Poisoning 39

4.5.2. Model Poisoning 39
4.6. Robust Aggregation 40
5. Results and Discussion 45
5.1. Experimental Setup 45
5.2. Centralized vs Decentralized 46
5.3. Adversarial Scenarios 48
5.4. Robust Aggregation 50
5.5. Critical Learning 52
6. Conclusions 55
6.1. Future Work 55

Bibliography 56

List of Tables

3.1. HyperParameters 25
3.2. Best model reached after hyperparameter search 27
3.3. Metrics comparison relative to the original proposal. 28

5.1. Federated Learning Rounds under Data Poisoning Attack 52
5.2. Federated Learning Rounds under Model Poisoning Attack 52

List of Figures

1.1. Cyber Kill Chain Phases 1

2.1. Visualization over Federated Learning Process 14

3.1. Overview of hyperparameter search results 28
3.2. Learning rate (a) and Optimizer (b) influence over metrics. Colour represents loss values 29
3.3. Training of best model 29

4.1. Division between Centralized Environment and the Federated Environment 32
4.2. Model weights delta aggregation - Example 37
4.3. 2D illustration of client updates, with the global model (center), benign updates (blue), and a malicious update (red) 41
4.4. Impact of Client Updates in the Aggregation 41

5.1. Machine Learning Model Comparisons 47
5.2. Model Performance 49
5.3. Model Performance with Robust Aggregator under Malicious Presence 51
5.4. Round limited Attacks Impact over Model Performance 54

List of Abbreviations

C2 Command and Control. 1

CLP Critical Learning Period. 22

DoHBrw CIRA-CIC-DoHBrw-2020. 23

FL Federated Learning. 2

MTA Malware-Traffic-Analysis.net. 23

NIDS Network Intrusion Detection System. 23

NN Neural Network Model. 2

ReLU Rectified Linear Unit. 24

TFF TensorFlow Federated. 33

TLS Transport Layer Security. 2

TNR True Negative Rate. 8

TPR True Positive Rate. 8

Tranco Tranco Web TLS 1.2 and TLS 1.3 dataset. 23

1. Introduction

Malicious software, or malware, is designed to infiltrate and compromise target systems [1]. The Cyber Kill Chain model [2], developed by Lockheed Martin, describes how such compromises typically unfold through seven key stages. As shown in Figure 1.1 It begins with reconnaissance, where attackers gather information about their target, followed by weaponization, in which they craft a malicious payload. This payload is then delivered through channels such as phishing emails or malicious websites.

Once delivered, the malware is executed during the exploitation phase and establishes persistence in the installation phase. The attack continues with Command and Control (C2), enabling remote access to the compromised system, and concludes with the actions on objectives, where attackers achieve their goals, such as data theft or system disruption.

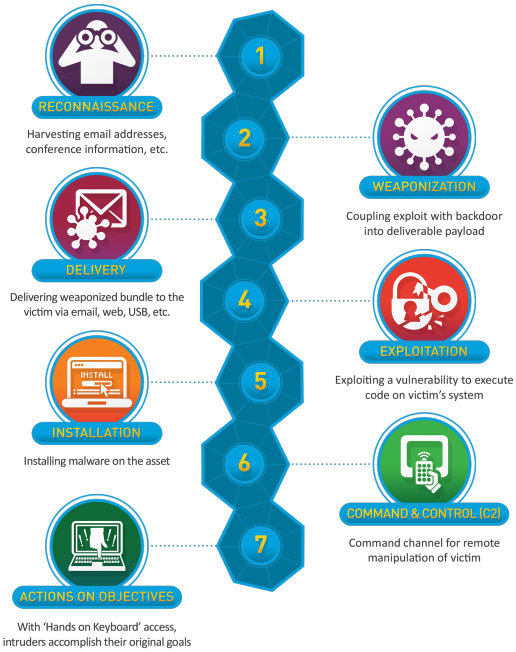


Fig 1.1 - Cyber Kill Chain Phases

The C2 phase is key in this process, due to the link that has been established between the victim's machine and the malicious infrastructure. As malicious activities increase, so does the volume and sophistication of C2 communications, where, according to the latest European Union Agency for Cybersecurity (ENISA) Threat Landscape report, threat actors are continually improving their C2 techniques to evade detection [3].

In fact, C2 communications are leveraging the Transport Layer Security (TLS) protocol, which reduces the effectiveness of detection mechanisms by limiting the exposure of traffic information. As cryptographic protocols continues to evolve, the amount of observable patterns in network traffic decreases even further [4]. This makes traffic inspection more difficult and increases the complexity for network analyzers attempting to detect C2 communications.

Building a complete and robust machine learning model capable of detecting malicious patterns requires large amounts of data to be able to learn and make accurate inferences about network traffic. Not only is the quantity of data important, but diversity is also key, since by training on a wide and varied dataset, the model is better able to capture and understand complex patterns within the traffic [5].

In this context, collaborative learning for traffic detection can offer significant benefits to participating parties by enabling them to share insights and patterns related to malicious activity. By leveraging such insights from others, each party can enhance their own ability to identify security threats within their networks more effectively. However, collaborative learning requires sharing sensitive data, which poses a privacy challenge. Due to the privacy-sensitive nature of network traffic, collecting and sharing such datasets becomes prohibitive and impractical, as organizations may not fully trust their peers or be willing to expose internal network traffic.

Federated Learning (FL) is a new computation paradigm in the machine learning world that seems suitable for implementing a private and secure collaborative learning environment, as it does not require the sharing of sensitive information [6].

Another key factor in building a model that incorporates information from multiple clients is that the model becomes directly influenced by the quality of their inputs. As observed in centralized machine learning solutions, poor or malicious data can significantly degrade the overall performance of the model. These impacts can also be extended to decentralized systems such as Federated Learning, ultimately raising concerns about the security of such models.

1.1. Proposed Approach

Firstly, this thesis reviews existing machine learning-based C2 detection solutions and adapts them to a Neural Network Model (NN). To achieve this, hyperparameter optimization is conducted in order to achieve an optimal structural parameters.

Subsequently, a Federated Learning enviroment will be designed and implemented accordingly with the previously defined NN model hyperparameters. A behavioral study of this system will then be conducted to assess its performance.

Lastly, adversarial scenarios will be considered and implemented into the Federated Learning deployment. This will allow for an impact analysis followed by their security countermeasures implementation.

1.2. Document Structure

The thesis starts by introducing the research problem, outlining the objectives, and describing the overall structure in Chapter 1. It then reviews the state of the art and the relevant technologies in Chapter 2, before presenting the design and creation of the neural network model in Chapter 3. Chapter 4 extends this work with the implementation of the Federated Learning model. The experimental results are reported and analyzed in Chapter 5, and the dissertation concludes in Chapter 6 with final considerations and directions for future work.

2. State of The Art

This chapter introduces the fundamental principles needed to fully comprehend the work presented in this thesis. Given the problem's foundation on the domains of network traffic analysis and machine learning model approaches, it is imperative to provide the theoretical and technological bases of the challenge and review the relevant literature.

2.1. Malware Traffic Detection

Malware refers to any software that intentionally executes malicious actions on a target system. It includes a wide range of types, such as viruses, worms, Trojan horses, rootkits, and ransomware [1].

In alignment with the thesis central research, C2, serve as a means for an attacker to maintain persistent access to a compromised system. C2 represents a communication channel that enables remote command execution, allowing the attacker to control the infected machine and carry out further malicious activities. Typically, the establishment of a C2 channel occurs after the initial malware infection, marking it as a post-intrusion phase of the attack lifecycle [7].

Network traffic refers to the transmission of data packets between two or more systems across a network. These packets contain essential information required for establishing and preserving connections, such as source and destination IP addresses, port numbers [8], and digital certificates used during authentication procedures[9].

Even when encryption protocols such as TLS 1.2 are employed to protect the content of the communication, certain metadata within packets remain visible in plaintext. This is because only application data and the final handshake messages are encrypted. Before the cipher suite negotiation and certificates exchange, the communications happens in cleartext along with lower-layer headers (IP/TCP), which are added by the network stack and remain unencrypted for routing purposes [9], [10].

The existence of unencrypted metadata of network communications makes it possible to identify potentially harmful activities occurring within the network. A common and recommended [11] practice among organizations is to leverage Indicators of Compromise (IOCs), such as known malicious IP addresses or digital certificates used by malicious actors to trigger alerts or block communications [12], [13]. IOCs are based on intelligence

gathered from previous incidents and are typically associated with known threat actors or cyber campaigns reported by security vendors, intelligence feeds, or national cybersecurity agencies.

While this approach is highly effective in detecting known threats, it fails when dealing with novel, polymorphic attacks or previously unreported attacks. Such attacks lack associated IOCs, making traditional detection methods ineffective[14]. Furthermore, the reliability of static indicators such as IP addresses is increasingly limited, as attackers frequently change or adapt their infrastructure using technologies like VPNs or proxy services, giving them access to a large and dynamic range of IP addresses.

2.2. Machine Learning Based Malware detection

To proactively detect previously unknown attacks, it is essential to develop an understanding of what represents a normal network behavior. By establishing a baseline of typical traffic patterns within a network, assuming it is not already compromised, machine learning models can identify deviations or outlier patterns that may indicate malicious activity.

Recent research has increasingly focused on applying machine learning techniques to establish these traffic baselines and detect anomalies indicative of malware, including C2 communications [15]. Unlike traditional methods that rely on Indicators of Compromise, machine learning approaches attempt to detect suspicious patterns, offering a more generalized and adaptive detection framework which don't rely on predefined IOCs.

Since network traffic supports most digital communications, it generates massive volumes of data, making manual analysis both highly difficult and complex. However, machine learning algorithms can be trained to recognize subtle patterns or correlations that are not easily observable at human sight and by operating at greater speed it also increases the analysis efficiency. These models can analyze features extracted from raw packet data, such as timing, size, frequency, and encryption-related parameters, to infer whether a connection is benign or potentially malicious.

The work presented in [16] focused on the development of a machine learning model to identify command and control communications within TLS 1.2 and TLS 1.3 encrypted traffic.

2.2.1 Command and Control Detection

According to the most recent Threat Landscape study from ENISA [3], command and control attacks are increasingly being carried out in a encrypted fashion (using SSl/TLS) to blend in with regular network traffic flow. Detecting C2 communications in encrypted traffic, particularly within TLS 1.3, presents a significant challenge for traditional malware detection techniques. As encryption conceals both payloads and metadata, it is necessary to explore alternative methods to extract some knowledge from the encrypted traffic. In this context, [16] introduces two machine learning models (Decision Tress and Random Forest Trees) designed to infer C2 activity from observable network traffic. These models rely on features such as certificate size and specific behaviors in application-layer message lengths, offering effective detection methods even without access to decrypted content.

Certificate Based Approach

The authors in [16] denote that in the previous security protocol, TLS 1.2, C2 communications often made use of short and self-signed certificates, a behavior frequently observed especially in attacks conducted by low-skilled actors ("*script kiddies*") using pre-built malicious payloads. This behavior was expected to persist in malicious communications over TLS 1.3.

Since certificate contents are encrypted in TLS 1.3, unlike in TLS 1.2, it is no longer possible to inspect them directly. However, a model can still infer certificate characteristics, such as their size, by analyzing the length of the messages exchanged during the handshake. This approach demonstrated promising initial results. When tested against traffic from a public Metasploit (MS) TLS 1.2 dataset, the model achieved 100% accuracy in classifying C2 communications as malicious.

However, the proposed model has known limitations. More sophisticated adversaries can bypass detection by artificially inflating certificate sizes, thereby evading models that rely solely on size-based heuristics. In such scenarios, the model's effectiveness dropped to 0% accuracy.

Application Behavior-Based Approach

In order to respond to more sophisticated attacks, the authors of [16] propose another model. Their study demonstrated that C2 traffic has a specific pattern where specific payloads lengths are typically found in the intially exchanges messages, meaning that

relying on certificates size inference is no longer needed. Their detection method requires isolating application data. Therefore, TLS 1.3 messages related to the handshake or alerts must be filtered out beforehand to ensure the model receives only the relevant content.

The results demonstrated improved generalization. Although this model achieved slightly lower accuracy on the Metasploit TLS 1.2 dataset, it proved significantly more effective in scenarios where attackers artificially increased certificate sizes. By adopting a behavior-based approach, the model shows greater resilience and is capable of detecting a wider variety of C2 communication patterns.

2.2.2 Evaluating Network Traffic Classifiers

Network Traffic Classifiers, such as the ones analyzed, must be extremely reliable in correctly identifying what constitutes malicious behavior (=1) and what does not (=0), since they are the last line of defense. However in real-world scenarios most of the traffic constitutes benign activity. As such, this creates a concern for the consequences of models with low false positive rate, which can lead to an overwhelming and unmanageable number of erroneous alerts. To properly assess a model's effectiveness and mitigate this risk, a range of performance metrics is employed.

Accuracy measures the portion of correct predictions made by the model out of the total number of predictions, more concretely:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Although accuracy might seem like a suitable metric, it can sometimes be misleading. In situations where the data is imbalanced, a model may achieve high accuracy simply by correctly predicting the majority class, while consistently failing to identify the minority class. This is problematic, especially in our context, since organizations typically generate significantly fewer malicious communications compared to normal activity. As a result, accuracy alone may hide poor performance in detecting malicious behavior.

To better evaluate how a model performs in distinguishing between benign and malicious traffic, two key metrics are commonly used: the True Negative Rate (TNR) and the True Positive Rate (TPR).

The TNR quantifies the model's ability to correctly classify benign traffic, and is defined as:

$$\text{TNR} = \frac{TN}{TN + FP}$$

On the other hand, TPR reflects the proportion of malicious traffic that is correctly identified, and is expressed as:

$$\text{TPR} = \frac{TP}{TP + FN}$$

Lastly, **Loss** provides a broader measure of the model's prediction error, defined by the chosen loss function during training. A lower loss indicates that the model's predictions are better aligned with the true labels.

2.2.3 Limitations of Centralized Machine Learning Models

While machine learning models have demonstrated potential in detecting malware, including C2 communications as proved by [16], several challenges hinder their effectiveness in real-world applications. A significant concern is the shortage of publicly available datasets for training and evaluating these models [17],[18]. Machine learning models performance increase as the amount of data available to train increases [5] as well as their quality. Collecting the data from various sources could be the solution to increase the number of data avialable, however as noted in [18], entities are often hesitant to share data due to privacy concerns and legal constraints, leading to a lack of diverse and representative datasets. In the absence of such knowledge, organizations train their models on their own data, which leads to the development of models that are oblivious to different communication, attack, and environmental patterns.This ultimately leads to the existence of models across the different organizations that are unreliable and incomplete, increasing the risk of not being correctly alerted of possible ongoing attacks.

2.3. Neural Network Models

Neural networks, as described in [19], are a paradigm within deep learning and unlike traditional machine learning methods, which depend heavily on feature engineering and task-specific assumptions, neural networks can automatically learn features from the data, thus reducing the reliance on handcrafted inputs and allowing the model to represent intricate and non-linear dependencies. A neural network model has two different kinds of parameters: internal parameters, such as weights or bias, which are adjusted during training and affect the model's predictions, and hyperparameters, which are predefined and

affect the learning process. These parameters control the machine learning model's structure and behavior. Learning rate, optimizer functions, batch size, dropout rate, number of layers, number of neurons per layer are examples of hyperparameters.

- **Batch size** determines the number of samples processed before the model's weights are updated. Smaller batch sizes tend to provide more frequent updates, which can help the model converge faster but may result in noisier gradients. Larger batch sizes provide more stable gradient estimates but require more memory and can lead to slower convergence.
- **Number of layers** directly influences the model's capacity to learn and represent data. By adjusting this, it is possible to control whether the model is shallow (better for simple problems) or deep (better for learning hierarchical and complex patterns). Selecting it as a hyperparameter allows for fine-tuning based on the complexity of the dataset and the task requirements.
- **Number of neurons per layer** affects the model's ability to represent features at each level of the network. By optimizing this, the model has enough capacity to learn patterns in the data without overfitting or wasting resources. This balance is critical for efficient and effective model performance.
- **Dropout** is a regularization technique that randomly ”drops” a fraction of neurons during training to prevent the model from over-relying on specific neurons. Higher dropout rates enforce stronger regularization, which is useful in preventing overfitting but may hinder the model's ability to capture the full complexity of the data. Lower dropout rates provide minimal regularization, which can be effective for simpler models or when overfitting is less of a concern.
- **Learning rate** controls the step size at which the model updates its parameters during optimization. A smaller learning rate allows for finer adjustments, which is critical for reaching the optimal solution in complex loss landscapes but may require more training epochs. A larger learning rate accelerates convergence but risks overshooting the optimal solution or oscillating around it.

- **Optimizer Function** controls how the model's weights are updated during training, directly affecting learning efficiency and performance. Examples of such include the Adam function, which uses a combined approach for weight decay and optimization, which can sometimes lead to overfitting, whereas AdamW decouples weight decay from the optimization process, improving regularization and generalization [20].

2.3.1 Structure

The work in [21] details the structure of a typical neural network model which consists of a series of interconnected layers of neurons. Each layer progressively learns to extract and represent patterns from the input data.

The first layer, known as the input layer, contains a number of neurons equal to the number of features in the training dataset. This ensures that all available information is fed into the model. The data is then propagated through one or more inner layers, where increasingly complex patterns are captured through successive transformations.

The number of neurons within the hidden layers plays a critical role in determining the learning capacity of a neural network. A larger number of neurons can enable the model to capture more intricate and abstract patterns, potentially leading to a deeper understanding of the data. However, an excessively large number can lead to overfitting, a situation where the model memorizes the training data too perfectly and performs poorly on new data due to lack of generalization [22].

2.3.2 Learning

Learning in a neural network occurs through the adjustment of internal model parameters to improve predictive performance over time. This process is guided by a loss function, which quantifies the discrepancy between the model's predictions and the true target values [21]. An optimizer then uses this feedback to update the model's weights iteratively, with the goal of minimizing the loss and progressively enhancing the model's accuracy. Despite the adjustment of the internal parameters throughout the training, the process is guided and set by the hyperparameters set beforehand.

In order to mitigate the lack of generalization, also known as overfitting, it is crucial to consider how hyperparameters influence neural network performance. As mentioned earlier, increasing the number of layers or the number of neurons can improve overall metrics,

since the model becomes capable of capturing more patterns. However, if these hyperparameters are increased excessively, the model may simply memorize the patterns extracted from the training and validation datasets. As a result, when presented with new data, the model fails to produce accurate predictions, as it has been tailored too closely to the original dataset [22]. A technique to address this lack of generalization is called dropout, which is often used during the training process [23]. Dropout is a regularization technique that prevents overfitting by randomly setting a fraction of input units to zero during training. This means that during each training step, some neurons are randomly dropped out of the network, which forces the network to learn redundant representations. The proportion of neurons that deactivate during training is defined by the dropout rate. For example, if a layer has a dropout rate of 10% applied to it, approximately 10% of its neurons are randomly deactivated during each training iteration. This means that each neuron's output is retained with a probability of 0.9 and set to zero with a probability of 0.1.

2.4. Federated Learning and Security Considerations in C2 Detection Paradigm

A paradigm developed to tackle the problem of data scarcity in the training of machine learning models is Federated Learning [6]. Federated learning allows for different participants to share among them information, in a private manner, in order to create a global machine learning model that leverages multiple data sources without disclosing any sensitive information throughout the learning process. In this approach, a machine learning model is trained without the existence of a single location containing all datasets from the participants.

To achieve this level of a distributed machine learning paradigm, it is important to define the environment clearly. Each of the N participants $\{p_1, p_2, \ldots, p_N\}$ in a federated learning environment has its own dataset $\{d_1, d_2, \ldots, d_N\}$, and a central server S is in charge of coordinating the clients' activities (i.e., local learning) and building a global model. The entire procedure is coordinated by the server using r communication rounds.

The federated learning process typically consists of three main steps in each r rounds [24]:

1. **Initialization:** At the beginning of each round, the server distributes the latest version of the global machine learning model to all clients.

2. **Local Training:** Each client p_i, where $i = 1, \ldots, N$ copies the global model to a local model and trains it using the local dataset d_i. The local model is updated as a result of this training. Once the training is complete, the client provides these updates to the server. Two methodolgies exist to transmit the local updates: either the client provides the new model with the corresponding weights, or it communicates the difference (model delta) between the updated local model and the original global model.

3. **Model Aggregation:** Upon receiving the updates from the clients, the server aggregates them using a predefined aggregation function to update the global model. The authors of [6], while introducing FL, also present a standard and innovative aggregation function that is the *FederatedAveraging* Algorithm. It consists of the computation of a weighted mean from the client updates, where clients with more data samples have an higher influence over the global model parameters when compared to clients which don't have as many data samples. This can be interperted as clients with more data samples know more information and can extract more knowledge thus can provide better updates which will result in a better global model.

This sequence of phases is done in an itterative manner throughout r rounds.

As result, all participant clients benefit by the correctness of the federated system since they now possess an enhanced machine learning model that has evolved not only using their private data but also the other participants data without disclosing sensitive data. In 2.1, a diagram illustrating the Federated Learning process can be found. In the diagram example, three distinct clients holding their own datasets train a local model which result in the transmission of information to an aggregation function hold at the server side which will be used as input to a global model.

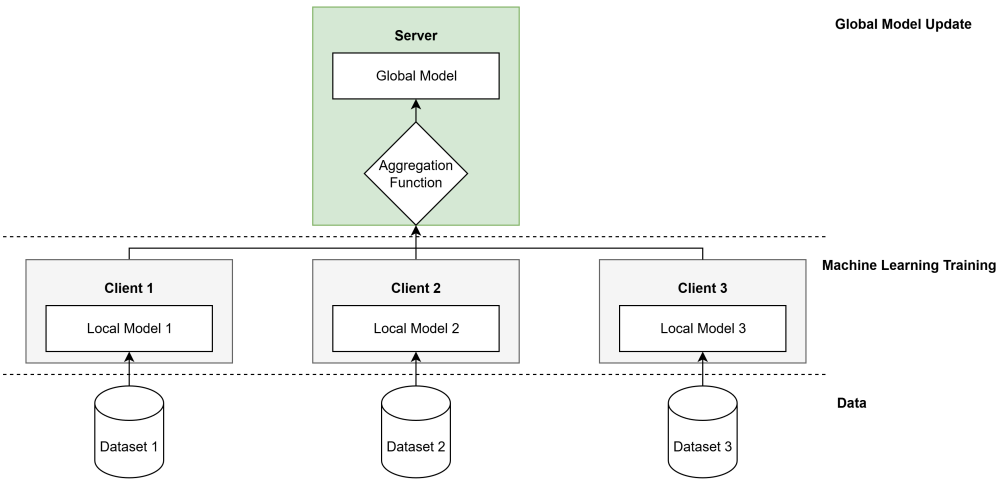


Fig 2.1 - Visualization over Federated Learning Process

An alternative perspective on the process is presented in the form of pseudo-code, highlighting the functions executed on both the server side and the client side.

Algorithm 1 Federated Learning: Server and Client Procedures

Server Side:
Initialize W {Initial model weights}
for each round **do**
    Send W to clients
    Receive local updates Δ_i from clients
    Aggregate updates Update global model W
end for

Client Side:
for each round **do**
    Receive global model W
    Train local model on dataset D_i
    Compute update $Delta_i$ and send to server
end for
=0

To better explain the pseudo-code presented, the process begins with the server initializing a global neural network model and distributing it to the clients. Upon receiving it, each client trains a local copy of the model using its private dataset. After training, the

client computes the update (delta) by comparing the trained local model with the one received from the server and sends this update back. Once all client updates have been collected, the server aggregates them—commonly through an averaging function such as FederatedAveraging—and uses the result to update the global model, which will then be redistributed in the next round.

As denoted by [24], three different architectures exist for handling data from each client. In a **Horizontal FL architecture**, all N clients share the same feature space, meaning they have identical data structures but may differ in sample sizes. In contrast, a **Vertical FL architecture** assumes clients share the same sample set but possess different feature spaces, i.e., their data structures vary. Finally, **Federated Transfer Learning** applies to scenarios where clients differ in both data structures and sample sizes. In this master thesis, the focus is on the horizontal architecture, as all datasets share the same feature space.

2.4.1 Selection of Machine Learning Models for Federated Learning

A foundational step in constructing a federated learning model is to define which machine learning approach the server and clients will adopt for training with their respective datasets. Neural networks are the preferable choice for federated learning systems in several studies [25], [26], [27], due to how their structure representation allow for direct and simple averaging to a single model using methods such as Federated Averaging [25]. In contrast, other machine learning architectures such as Decision Trees or Random Forests introduce greater complexity, as each client may generate data-dependent tree structures with varying depths, node splits, and feature selections which cannot be directly averaged or aggregated into a single model. This results in the most trivial choice which were neural networks.

2.4.2 Federated Learning in Malware Detection

In FEDetect [28], the work focused on testing different configurations of models such as traditional machine learning, deep learning, and federated learning. Since this thesis aims to detect Command and Control presence in network traffic, a binary problem, the analysis of FEDetect was concentrated on the binary problem that the work addressed using federated averaging. The neural network models trained under [28] were carried out using the Adam optimizer function instead of the typical SGD (stochastic gradient descent) presented in [6] as different studies [29], [30], [31] have shown better results. In contrast to

SGD, the Adam optimizer assigns adaptive learning rates to individual parameters, which results in faster convergence and reduces the sensitivity of training to the initial learning rate selection, allowing for overall better results [32].

Python was used as the programming language, alongside with TensorFlow for building the deep learning models, Keras as a high-level API, and support libraries like NumPy, pandas, and scikit-learn for data handling and preprocessing.

The research results demonstrated the effectiveness of avoiding the centralization of data in a single repository and allowing for a more scalable solution by demonstrating that **models operating in a federated environment attain the exact same level of accuracy as non-federated models**. Notwithstanding the favorable outcomes, the simulation was conducted with homogeneous data distribution, which could have a beneficial impact on the outcomes.

2.4.3 Federated Learning Threats

Even though Federated Learning is a suitable solution to facilitate distributed machine learning models, there are security considerations in addition to privacy ones.

In [33], attacks are divided in two main areas: **model performance** and **privacy attacks**. And as the focus of this master's thesis is on securing the global model against malicious client contributions, the analysis primarily concentrates on **model performance attacks**

Model performance attacks are described as attacks where malicious clients send incorrect or corrupted data in a direct attempt to influence the global model performance decline.

By manipulating data labels, malicious clients can affect or add biases to the global model to serve their own objectives. In order to prepare for a future C2 attack on a target environment, for example, an adversary may submit well-crafted, subtle modifications that mislead the global model. This might lead to the development of a backdoor, which would enable their particular attack methods to avoid detection while preserving the model's functionality for the remaining participants. [34]

Data Poisoning Attacks

Participants in a Federated Learning environment are neither constrained nor directly supervised regarding the data they manage and use for local training. As long as the information they transmit to the server is aligned to the expected format, they are free to perform any form of data preparation. This lack of oversight creates an opportunity for adversaries to exploit the system by injecting fabricated data or intentionally manipulating it, through mislabeling or altering class distributions, so that it no longer reflects the true underlying patterns [33].

In [35], three different data poisoning attacks are listed as possible to happen under a supervised environment, which are:

- **Benign label flipping**. In this scenario, the attacker's goal is to flip all target values from 0 (benign) to 1, indicating the attack presence. Ultimately, this aims to achieve a TNR of 0%, meaning that all network traffic would be classified as malicious, resulting in an overload of alarms to be analyzed by the security team.
- **Attack Label flipping**. This attack, in opposition to the previous attack described, aims to achieve a TPR of 0%, meaning that all network traffic would be classified as non-malicious, thus manipulating the model so that no attack can be recognized. This is achieved by flipping all labels to 0 (benign).
- **All labels flipping**. By flipping all labels to their opposite, the last attack type aims to reduce the model's accuracy as much as possible. This means that all malicious labels will be converted to benign labels and all benign labels to malicious ones. The model would be affected by this attack in such a way that only benign activity would trigger alarms, while maliicous activity would not be alerted.

All the previous attacks assume a supervised learning environment, as the presence of labels is necessary for the attacker to manipulate them as intended. Unsupervised attacks were not explored, as they fall outside the scope of the problem addressed in this study.

As demonstrated in [35] and corroborated by [36], the presence of a single malicious client could disrupt with the server's ability to handle requests when performing data posinoning attacks. However, by having a higher presence, the Sybil attack, which uses numerous false identities that are legitimate participants within the federated environment, amplifies the attack's impact [33]. In a benign environment, the clients carry out their local training and transmit the model updates to the central server. However, the client is free to carry out any action we desire as long as the updates' structure stays the same, which means

he could alter the outcomes to suit our needs and launch an attack on his target and objectives.

Model Poisoning Attacks

[35] also studies a broader class of adversarial strategies aimed at compromising the integrity of machine learning models, known as Model Poisoning Attacks. In contrast to data poisoning, which is sending poisoned or incorrectly labeled data for local training, model poisoning affects the global model by modifying the local model updates prior to their transmission to the server for aggregation. Here, a malicious client deliberately modifies its post-training model updates (e.g., scaling their weights to increase their control over the global model in comparison to honest participants). Additionally, when both are used together to optimize their effects, model poisoning attacks can also include data poisoning approaches. Examples of such attacks include the following. To facilitate understanding, we first define the variables used throughout this section:

- K: the total number of clients participating in the federated learning process.
- f: the number of malicious clients.
- α: a scaling factor applied to the updates of malicious clients, which is to be determined for each attack scenario.

- **Gradient factor attack**. In this attack the attacker aims to cancel the benign clients updates to the server by sending an update with a scaled value associated that is in the opposite direction, meaning that the malicious attacker would take full control over the global model in the server.

$$\frac{1}{K}\Big(K - f + \alpha_{\text{grad}} \cdot f\Big) = -1 \tag{2.1}$$

- **Model cancelling attack**. In this attack, the goal is to set the current global model to 0, meaning that the attacker would now send an model update scaled by a factor that would point to 0 all the weights

$$K - f + \alpha_{\text{param}} \cdot f = 0 \tag{2.2}$$

The findings of the experiment conducted in [35] were obtained in a federead environment with eight distinct clients under different aggregation methods, however as guide to

understand the attack disruption an focused analys was done the Federated Averaging algorithm. The existence of a single attacker in data poisoning employing all labels flipping lowers the global machine learning model's F1-score from 99 to 84. As predicted, the impact of malicious clients increases as their number rises. For example, the F1 score dropped from 99 to 51, or nearly 50%, with three malicious clients. This demonstrates how the global model is significantly impacted, even by simple and low-effort attacks, which leads to a decline in the detection performance of all clients.

The impact is far more detrimental when this attack is extended by carrying out a Gradient Factor attack because having just one malicious client lowers the F1 score from 99 to 10. Whether there are three malicious clients or only one, the attack's impact remains constant regardless of the number of malicious clients. These results are significant because they demonstrate how the typical federated environment setup is completely vulnerable to data poisoning and model poisoning attacks.

In regard to the Model canceling attack, it resulted in a F1-score drop from 99 to 0 in the presence of a single malicious actor, showing that the theoretical reasoning had partical validation as it was able to cancel the global model.

Free-Riding Attacks In this scenario, the attacker's objective is not to deliberately poison the global model but rather to benefit from it without expending resources on local training. Instead, the participant sends random data or trains on a minimal amount of data before sending updates to the server. This update contributes to a degradation of the global model's performance due to the poor quality of updates provided by the malicious client.

Since the dissertation's problem involves the creation of a federated learning environment in which clients collaborate to develop a machine learning model capable of detecting C2 communications, it is critical to address the security of such an environment because disruptions or even attacks can impair the model's performance, making participation in the decentralized solution more difficult. Studying security countermeasures for the different kinds of attacks that can compromise such federated systems is therefore essential.

2.4.4 Security and Robustness

When working in a decentralized environment, it is impossible to control how and what each participant contributes, as each participant operates independently. The point where security measures can be applied is at the server side, since this is where all updates

from the clients are collected. Therefore, the server must be a secure and trusted location where appropriate security controls can be implemented. The aggregation function typically relies on simple methods such as the mean; however, previous studies have shown that these approaches lack resilience against malicious contributions [35][36].

To address this, different approaches exist as alternative aggregation functions designed to enhance robustness.

- **Coordinate-wise median** [35]. This method completely excludes outliers by selecting only the median value for each coordinate. However, it may sacrifice potentially valuable contributions from clients whose updates are considered outliers.
- **Coordinate-wise trimmed mean** [35].In this approach, aggregation discards a predefined number of the lowest and highest values based on a constant c. For example, if c = 2, the two smallest and two largest values for each coordinate are excluded from the aggregation.
- **S-resampling** [35]. Here, the goal is to combine different client updates into a single update. Instead of directly using local models trained by individual clients, the server generates updates that are an average of randomly selected client updates. As a result, the update from one client would, on average, incorporate contributions from other clients. However, this approach carries a potential risk: if an attacker's influence is present in multiple updates, it could affect the aggregated result more broadly rather than being confined to a single compromised update.
- **Zeno** [24]. A scoring mechanism based on each client's contribution to the global model's loss function reduction is used by this aggregation function. High-scoring clients are chosen for the global model updates in this scenario, while low-scoring clients are punished and excluded from the global model aggregation for the newly updated model. The server calculates the impact of each client's updates on the global model loss function. Ultimately, only trustworthy clients are chosen, strengthening the learned model's resistance to adversarial circumstances. This approach may suffer from high computational complexity and reduced performance, as the server needs to perform validation on its dataset for all n clients in each round, and this overhead grows with the number of clients.
- **L2 Norm Clipping** [37]. The authors proposed and explored a robust aggregation function based on the inference that scaled attacks as the ones described under

the model poisoning attack have a magnitude of updates higher than non malicious clients which influence directly the norm produced by each client. With this, a threshold can be defined to reduce the updates universe to an upper bound meaning that updates that exceed the threshold are bounded by the maximum limit (clipping concept).

- **Geometric Median** [36]. The aggregation can be formulated as a geometric optimization problem, where the goal is to find a global model v^* that minimizes the sum of Euclidean distances to all client updates:

$$v^* = \arg\min_{v} \sum_{i=1}^{m} \|v - w_i\|_2$$

where $w_i \in \mathbb{R}^d$ denotes the update from client i, and m is the total number of participating clients. This is a problem that cannot be solved with an analytical method, meaning it only has a numerical method which is an approximation using the Weiszfeld algorithm[38] which results in a iterative update.

Only one of the robust aggregation functions described fails in situations where the server is unaware of the number of attackers in an environment. For example, if the number of clients exceeds $C * 2$, the Coordinate-wise median will fail because it will not invalidate or discard more malicious client updates. The remaining ones, are agnostic to the number of attackers in the environment, thus providing a more resilient aggregation.

As for the computational complexity, the aggregation functions discussed except Zeno and Geometric Median are non-iterative. This benefits and make such functions more suitable to larger scale federated learning deployments as the overhead scales easly with the number of clients. Although iterative processes are used in both Zeno and the geometric median (through the Weiszfeld algorithm), their natures are very different. In order to rank and filter clients based on their unique contributions to lowering the global loss on a server-side validation set, Zeno iterates over client updates. In [36], by minimizing $\sum_{i=1}^{m} \|v - w_i\|_2$, the Weiszfeld algorithm, on the other hand, iteratively refines the global model estimate $v^{(t)}$ to approximate the geometric median. Zeno's iteration concentrates on client selection, while Weiszfeld's iteration concentrates on aggregate convergence.

To evaluate the effectiveness of robust aggregation functions, several studies have compared their performance under adversarial scenarios. The **Coordinate-wise Median**, as presented in [35], demonstrates notable robustness against model poisoning attacks.

However, its effectiveness decreases as the proportion of malicious clients increases, particularly in data poisoning scenarios such as label flipping. For instance, with no malicious clients (0% corruption), the global model achieves an F1-score of 96%, but this drops significantly to 57% when 25% of the clients are adversarial. Although it is not a perfect aggregator, it is an improvement over the standard Federated Averaging aggregator because, in the case of model canceling attacks, the presence of 12.5% would cause the f1 score to drop to 0%.
A broader comparison in [36] reinforces these findings, highlighting the **Geometric Median**, followed by the **Coordinate-wise Median** and L_2 **Norm Clipping**, as the most effective methods under severe adversarial conditions. In scenarios with high corruption levels (40% malicious clients), the accuracy of these methods ranges approximately from 60% to 56%. These results demonstrate that while all three methods enhance resilience, the Geometric Median provides the strongest defense against adversarial behavior, however require more computational power affecting it's overall performance.

As reviewed, there is still no perfect aggregator and further research is needed to better respond to real life threats in the federated learning environment.

2.4.5 Critical Learning Period

Critical Learning Period (CLP) were discovered in [39] and explored under [40]. CLP refers to a period of time during the Federated Learning Process where the global model that is being trained and created is more subject to major changes. In practice, during CLP, deficits such as low quality or quantity of data cause irreversible model degradation despite the additional training that may occur afterwards. Through FedFIM introduced in [39] and FGN proposed in [40], results show that the CLP can be found in the first few rounds of the Federated Learning Process.

On the adversarial side, an attacker aiming to achieve a pronounced disruption level over the global model performance while spending low computational cost (performing the attack in all round) may limit his attack by focusing during the CLP. In [40], it is shown that launching an attack during this period, as opposed to attacks initiated midway through training (i.e., after half the total number of rounds), results in an impact more than five times greater.

3. Neural Network Model Construction

Having as foundation the concepts and related work explored under the previous chapter, we build upon the work of [16]. Our goal is first to ascertain if one can effectively adapt their predictive models to a more FL-friendly setup, namely neural networks, as seen in the previous chapter.

Firstly, we describe the datasets that were used for training and evaluation in the work of [16], which are the same datasets adopted for training and evaluating our target neural network model.

3.1. Datasets

The data used for model training and evaluation comprises a combination of multiple publicly available datasets. While their usage follows the work where they were initially proposed [16], here they are also split in a way that captures the FL scenario being explored. The datasets focus on TLS traffic only, and its usage by both benign applications and malware C2. In particular, they address the transition to TLS 1.3, due to the changes brought by this version and which are relevant in this Network Intrusion Detection System (NIDS) context.

For benign traffic, the Tranco Web TLS 1.2 and TLS 1.3 dataset (Tranco) is used. It contains TLS traffic captured while performing automated web-browsing. A dataset obtained from Malware-Traffic-Analysis.net (MTA) is used as a mixed source of both C2 and non-C2 traffic. TLS flows are positively labelled as C2 using server certificate information and, as such, this dataset contains TLS 1.2 traffic only. Part of the CIRA-CIC-DoHBrw-2020 (DoHBrw) was used as an external source of benign TLS 1.3 traffic for validation purposes.

The MS dataset contains traffic for a malicious sample executable contacting a Metasploit Framework server over TLS. It is used as another source of C2 traffic. Metasploit is a penetration testing framework that has also been previously associated with attacks. These tools are open-source and easily obtainable, making them an attractive resource for researchers, but also for ”*script-kiddies*”. For that reason, they are a good demonstration of the type of traffic one might wish to prevent; at the same time, we argue that the procedures followed to capture this traffic can be easily replicated by malicious actors as a step towards a poisoning attack. The network traffic captures were made while

running the sample in a victim system, and with a Metasploit Framework instance (C2 server) issuing commands and receiving responses. By definition, an attacker already has access to all of these elements. The MS dataset contains captures of this C2 traffic while using different TLS versions–1.2 and 1.3–and different server certificates, 'short' self-signed certificates, or 'long' certificate chains.

3.2. Neural Network Structure and Hyperparameter Selection

The neural network model, as seen in Chapter 2, is structurally defined by two different parameters. Internal parameters are with respect to weights and bias and are continuously changing throughout the learning process, while the hyperparameters define what structure the model must follow.

The chosen model architecture consists of multiple dense layers with Rectified Linear Unit (ReLU) activation functions. ReLU introduces non-linearity into the network by allowing positive values to pass through unchanged while mapping negative values to zero, thereby enabling the model to learn complex patterns efficiently. To prevent overfitting, dropout layers are incorporated for regularization. The final layer employs a sigmoid activation function, which maps outputs to a probability value between 0 and 1, making it particularly well-suited for binary classification tasks. The model is trained using the binary cross-entropy loss function, a standard choice for evaluating the performance of models in two-class classification problems.

Each NN model must be adapted, and its hyperparameters selected, to fit the specific task at hand. This is achieved through a hyperparameter search, also known as tuning. This was conducted using `Keras Tuner`[1]. A custom model builder function was defined to dynamically configure the architecture considering the tunable parameters. These parameters, the reasoning behind their tuning, and their possible values are listed in Table 3.1.

To address the heavy class imbalance of our the dataset, class weights were calculated using a function from `sklearn`[2], but were further adjusted to ensure that the minority class received appropriate emphasis during training, as during initial tests the model was not behaving as expected, resulting in low model performance.An early stopping mechanism was also implemented to monitor validation loss with a patience of 10 epochs, allowing training to halt early if performance did not improve.

[1] `https://www.tensorflow.org/tutorials/keras/keras_tuner`

[2] `https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html`

Parameter	Possible Values	Reasoning
Batch Size	32,64,128	Defines how many samples are processed per update, balancing noisy fast learning (small) versus stable slower learning (large).
Layer Size	1,2,3,4	Controls model depth, with shallow networks handling simple tasks and deeper ones capturing complex hierarchical patterns.
Neurons number	32,64,96,...,256	Determines representational capacity at each layer, where too few limit learning and too many risk overfitting.
Dropout Rate	0.1,0.2,0.3,0.4,0.5	Regularization method that randomly deactivates neurons to prevent overfitting, with higher rates enforcing stronger regularization.
Learning Rate	0.01, 0.001, 0.005, 0.0001	ets the step size of parameter updates, where small values ensure precision but slow training, and large ones speed up but risk instability.
Optimizer	Adam, AdamW	Governs weight updates, where Adam combines weight decay with optimization, while AdamW decouples the two processes, influencing regularization and generalization differently.

Tab 3.1 - HyperParameters

Using Keras Tuner's RandomSearch algorithm, 300 trials were executed, optimizing the validation loss while also employing early stopping for the best hyperparameter search.

Below, the code responsible for defining the hyperparameters and their possible configuration (*build_model*) is shown alongside with the function that will search the best configuration model (*tuner_search*).

We begin by defining the variables corresponding to the hyperparameters listed in Table 3.1. Next, we build the model object and add the first layer of neurons, whose size matches the number of features in the training data. After the input layer, we add the hidden layers according to the specified number of layers and neurons, followed by a dropout layer for regularization. Finally, we include the output layer, which performs binary classification using the sigmoid activation function, and compile the model with the chosen optimizer.

```
1 def build_model(hp):
2     num_layers = hp.Int('num_layers', min_value=2, max_value=4)
3     layer_dims = [hp.Int(f'units_{i}', min_value=32, max_value=256, step=32) for
      i in range(4)]
4     dropout_rate = hp.Float(f'dropout_rate', min_value=0.1, max_value=0.5, step
      =0.1)
5     lr = hp.Choice('learning_rate', values=[1e-4, 5e-4, 1e-3, 1e-2])
6     optimizer_name = hp.Choice('optimizer', ['adam', 'adamw'])
```

```
    batch_size = hp.Choice('batch_size', values=[32, 64, 128])
    model = tf.keras.Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape=(x_train_scaled.shape[1],)))
    for i in range(num_layers):
        model.add(tf.keras.layers.Dense(units=layer_dims[i], activation='relu'))
        model.add(tf.keras.layers.Dropout(dropout_rate))
    model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
    if optimizer_name == 'adam':
        optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
    elif optimizer_name == 'adamw':
        optimizer = tf.keras.optimizers.AdamW(learning_rate=lr, weight_decay=1e
    -5)
    model.compile(
        optimizer=optimizer,
        loss='binary_crossentropy',
        metrics=['accuracy', 'Precision', 'Recall']
    )
    return model
```

With the model compiled, we instantiate the tuner to perform hyperparameter optimization using the `RandomSearch` algorithm. The search is configured with a maximum of 300 trials, where in each trial a different combination of hyperparameters is evaluated. Model evaluation follows an 80/20 split, meaning that 80% of the dataset is used for training and the remaining 20% for validation. To monitor the process, training metrics and results are logged using TensorBoard, which facilitates visualization and analysis of the tuning process.

```
def tuner_search(tuner, x_train_scaled, y_train, groups_train, n_splits=5):
    tuner.search(x_train_scaled, y_train, epochs=20, validation_split=0.2,
    class_weight=class_weights_dict, callbacks=[early_stopping,keras.callbacks.
    TensorBoard("TLS13CBehav/logs")], verbose=1)
tuner = kt.RandomSearch(
    build_model,
    objective='val_loss',
    max_trials=300,
    overwrite=True,
    directory='TLS13CBehav/tb',
    logger=None
```

```
)
tuner_search(tuner, x_train_scaled, y_train_err_noerr, groups_err_noerr, n_splits
    =5)
```

Once the search was complete, the best-performing hyperparameters were retrieved and the corresponding final model was built. These parameters are listed in Table 3.2. Training logs were recorded using TensorBoard[1] to facilitate experiment tracking and performance visualization.

Parameter	**Best value**
Batch Size	64
Number of Inner Layers	2
Neurons per Inner Layer	{192, 256}
Dropout Rate	0.1
Learning Rate	0.0005
Optimizer	Adam

Tab 3.2 - Best model reached after hyperparameter search

Table 3.3 presents a comparison between our final model and the models proposed in [16], according to the metrics defined in their paper. By utilizing the metrics defined under the article that proposed the model we stay consistent when evaluating the newly created model.

Although the performance on Tranco data is slightly lower, the model achieves near-perfect results on predominantly malicious datasets such as Metasploit, thus providing to be a sucessful model.

[1] https://www.tensorflow.org/tensorboard

Model	Tranco TLS 1.3	MS TLS 1.3		DoHBrw
		short	long	(TNR)
Decision Tree [16]	98.64%	93.93%	93.54%	97.27%
Random Forest [16]	99.78%	93.87%	93.21%	99.42%
Neural Network	97.10%	99.91%	99.90%	96.60%

Tab 3.3 - Metrics comparison relative to the original proposal.

Figure 3.1 presents visually the hyperparameter search conducted and which were the configuration values used along side with their respective performance.

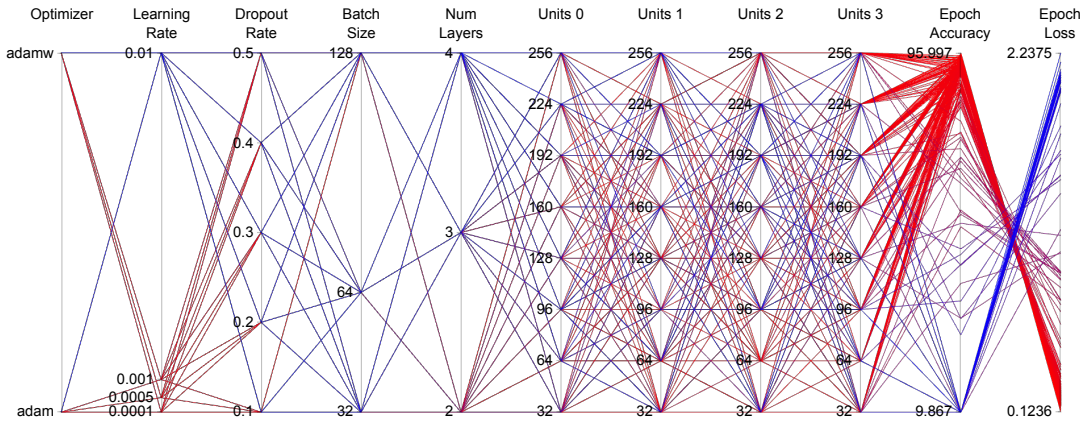
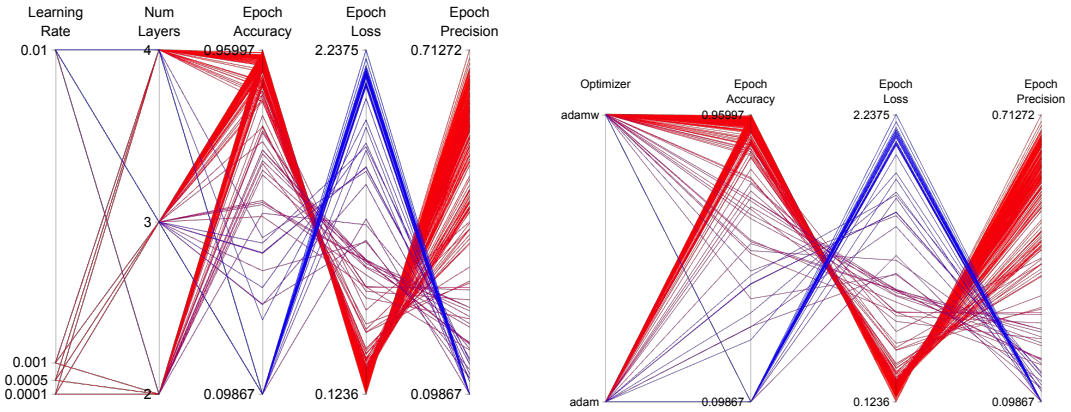


Fig 3.1 - Overview of hyperparameter search results

In Figure 3.2, we observe that using a lower `learning_rate` leads to improved precision and a reduction in loss. This behavior is expected, as a smaller learning rate allows the model to converge more gradually, enabling a more refined and thorough exploration of the solution space, which results in better performance. Regarding the optimizer algorithm, as shown in Figure 3.2, no definitive conclusions can be drawn between the two selected options, as both optimizers yield similarly strong performance.



(a) (b)

Fig 3.2 - Learning rate (a) and Optimizer (b) influence over metrics. Colour represents loss values

Lastly, and coming back to the results shown in Table **??**, the graphs highlighted in Figure 3.3 provide insights on how the resulting and discovered model was trained and how the performance metrics evolved during the epochs. As expected, as the number of epochs increases, the model's accuracy and precision improve, while the loss values decrease over successive iterations.

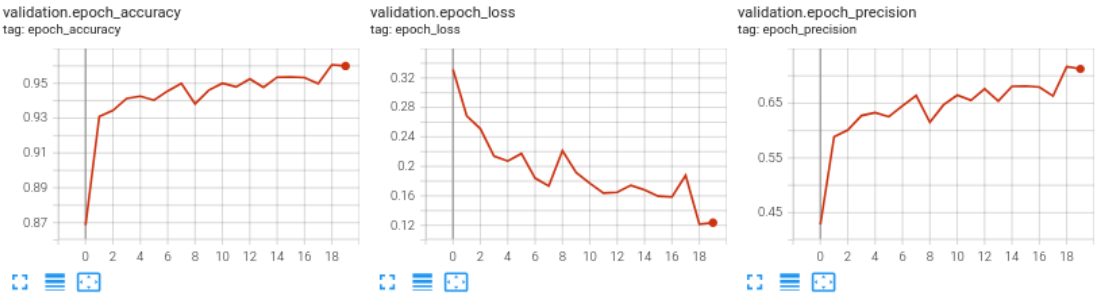


Fig 3.3 - Training of best model

4. Federated Learning Model

We now assess the feasibility of using the adapted malware detection in a federated learning setting. We introduce our system model and discuss how we adapt our datasets to a federated learning environment.

Figure 4.1 illustrates the difference between the environment that was built previously in a centralized manner, where the dataset was held at a centralized location, versus an environment where the dataset is distributed across different clients.

The initial process begins by dividing the dataset into two subsets: training data and test data. In the centralized environment, the training data is used to train the model through a K-Fold cross-validation procedure. In this approach, the dataset is partitioned into K equally sized folds, and the model is trained K times, each time using $K - 1$ folds for training and the remaining fold for validation. This process is repeated until all folds have been used as validation once, ensuring a robust evaluation. At the end of these iterations, a neural network model is built and trained.

An alternative approach is the decentralized (federated) environment, where the training data remains distributed across different clients and is never shared directly. Each client independently trains its own neural network model on its local data and periodically shares only the learned parameters (model weights) with a central global model. Through multiple rounds of federated training, the global model is iteratively updated and refined. Once the process converges, the result is a trained global neural network model built collaboratively without the need to centralize the data.

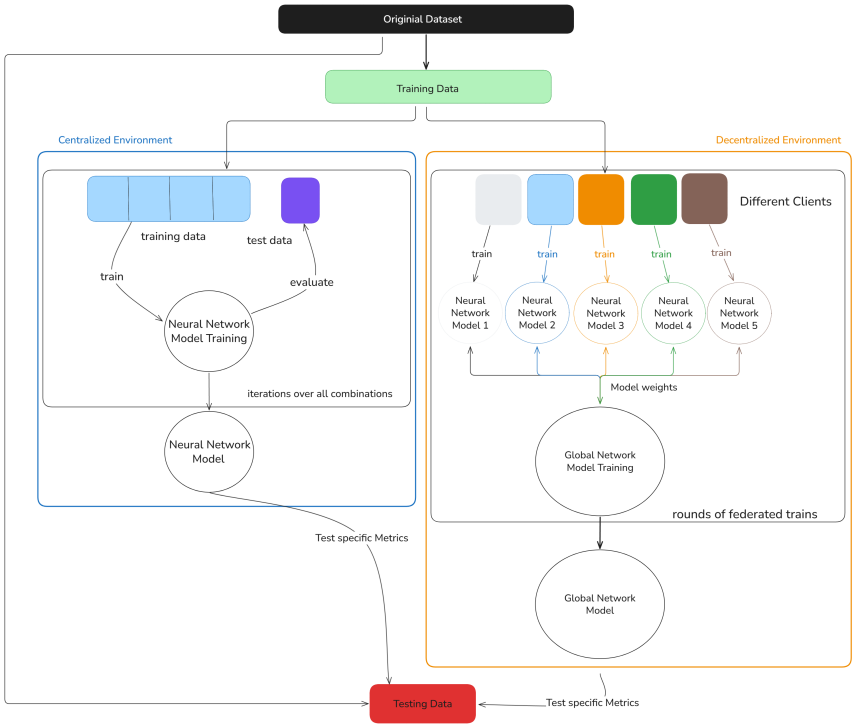


Fig 4.1 - Division between Centralized Environment and the Federated Environment

4.1. System Model

We consider a set of $n = 10$ participants $p_i \cdot i \in [1..10]$ and a central server S which instantiates a neural network model M_0. Each participant p_i possesses a private local dataset d_i. The learning process proceeds for a fixed number of r communication rounds. In each round, the following steps are executed:

1. The central server S distributes M_0 to all participant clients.
2. Each participant p_i creates a copy of M_0 to M_1 and trains M_1 using d_i. Once the training is complete, each p_i computes the model update $\delta_i = M_1 - M_0$ and return the vector to the server S
3. After receiving all δ_j, the global model is updated as:

$$M_0 \leftarrow M_0 + \frac{1}{n}\sum_{i=1}^{n}\delta_i$$

4.2. Partitioning the Dataset

In order to simulate a federated environment where each participating client possesses a private dataset, we emulate this scenario using a dataset composed of multiple network traffic capture files. While several approaches could be applied to partition this dataset,

our goal is to assess whether a federated learning setup can perform as effectively as a centralized one. Therefore, we adopted a specific partitioning strategy.

To partition the dataset, we used the *StratifiedGroupKFold* method, which divides the dataset into K folds while preserving both class distribution and group integrity. This ensures that each partition maintains a class distribution similar to that of the overall dataset and that groups which are defined as complete network traffic capture files are not split across clients. This is essential, as it would be unrealistic for different clients to contain fragmented parts of the same traffic capture.

Although in a real-world scenario, dataset partitioning would not necessarily take into account for uniform class distribution, since different organizations may experience varying levels of malware presence, we chose to preserve class balance across clients for evaluation purposes in this decentralized setting. Additionally, different clients may be exposed to different types of attacks, which can significantly influence how the model learns and generalizes. Although not addressed, investigating how to isolate and represent these variations in the dataset would consittue an improvement since we would be approximating the conditions in which the neural network model would be trained to the real-world.

The dataset is divided into 11 parts, even though we only simulate 10 clients. The additional partition is reserved for the server, which uses it to evaluate the performance of the global model after aggregating the clients' updates.

4.3. Standard Implementation

To evaluate the best-performing neural network model, we utilize the output from the hyperparameter search (see Section 2). The federated learning system was implemented using TensorFlow Federated [1], chosen for its compatibility with our previous work, which also relied on TensorFlow-based neural network models.

TensorFlow Federated (TFF) provides a high-level interface, `tff.learning`, that facilitates the implementation of common federated learning tasks, such as federated training, without requiring configuration of the underlying mechanics.

We begin by defining and constructing the neural network model:

```
1 def create_keras_model(input_size=80, layer_dims=[192,256], dropout=0.1):
```

[1] `https://www.tensorflow.org/federated`

```
    model_layers = [tf.keras.layers.InputLayer(input_shape=(input_size,), name='x')]
    for layer_dim in layer_dims:
        model_layers.append(tf.keras.layers.Dense(layer_dim, activation='relu', kernel_constraint=tf.keras.constraints.max_norm(1.0)))
        model_layers.append(tf.keras.layers.Dropout(dropout))
        model_layers.append(tf.keras.layers.Dense(1, activation='sigmoid'))
    return tf.keras.models.Sequential(model_layers)
```

Next, we wrap the Keras model into a TFF-compatible model, specifying the loss function and evaluation metrics::

```
def model_fn():
    keras_model = create_keras_model(data_X.shape[1],layer_dims=neurons,dropout=0.1)
    return tff.learning.models.from_keras_model(
        keras_model,
        input_spec=input_spec,
        loss=tf.keras.losses.BinaryCrossentropy(),
        metrics=[tf.keras.metrics.BinaryAccuracy(),
        tf.keras.metrics.Precision(name="precision"),
        tf.keras.metrics.Recall(name="recall"),
        tf.keras.metrics.AUC(name="auc")]
    )
```

Using the wrapped TFF model, we construct the federated training process with `tff.learning.algorithms.build_unweighted_fed_avg`, which applies the unweighted Federated Averaging algorithm, aligned with the aggregation formulation previously described. The optimizer used for the local updates by the clients and for the server to apply the aggregated model updates is Adam, configured with the learning rate determined during the hyperparameter search:

```
trainer = tff.learning.algorithms.build_unweighted_fed_avg(
    model_fn=model_fn,
    client_optimizer_fn=tff.learning.optimizers.build_adam(learning_rate=0.0005),
    server_optimizer_fn=tff.learning.optimizers.build_adam(learning_rate=0.0005),
)
```

Finally, we initialize the server state using the previously configured federated learning process (*trainer*). Then, for each defined round, we perform a full training cycle, which includes broadcasting the global model, local client training, update transmission, and aggregation. At the end of each round, the server state is updated to reflect the newly aggregated global model.

```
state = trainer.initialize()
with summary_writer.as_default():
    for round_num in range(1, 200):
        result = trainer.next(state, data)
        state = result.state
```

While this approach allows for quick deployment and integration, it does not support customization of internal mechanisms. In particular, it limits the ability to simulate model poisoning scenarios or to introduce fine-grained control over the training and aggregation steps.

4.4. Custom Implementation

To deep dive into the full implementation of federated learning functions, we must deviate from the previous implementation using the *tff.learning* API and explicitly define the Federated Learning functions:

1. Server-to-Client broadcast
2. Local client Training
3. Server update (aggregation)

The federated learning process happens in an iterative manner, meaning we must define a function to encompass the various building blocks.

```
def next_fn(server_weights, federated_dataset):
  server_weights_at_client = broadcast(server_weights)
  client_weights = client_update(federated_dataset, server_weights_at_client)
  mean_client_weights = mean(client_weights)
  server_weights = server_update(mean_client_weights)
  return server_weights
```

4.4.1 Broadcast

This function distributes the federated value across all the clients in a equal manner, in this case the $server_weights$. TFF allows for this seamless implementation by using the $tff.federated_broadcast$ function.

4.4.2 Local Client Training

In this function, the client begins by receiving the model weights $server_weights$ sent by the server and by selecting its local optimizer (Adam).

At the beginning of training, the client initializes the optimizer state and the model weights as the ones received from the server and proceeds (lines [2-5]) to iterate over its local dataset in data-batches. For each batch, a forward pass is executed to generate predictions and compute the corresponding loss. Using `tf.GradientTape`, the gradient of the loss with respect to the model weights is calculated. These gradients are then used by the optimizer to update the model weights incrementally throughout the training loop (lines [6-18]). This is the core neural network training as we seen in the centralized enviroment.

Finally, after all batches have been processed, the client computes the *model delta* which represents the difference between the updated local model weights and the original weights received from the server and returns it ([19-23]).

The data transmitted to the server consists of an array of values that represent the adjustments made by the client to the original global model.

```
1  def client_update(model, dataset, initial_weights, optimizer):
2      client_weights = initial_weights.trainable
3      optimizer_state = optimizer.initialize(
4          tf.nest.map_structure(tf.TensorSpec.from_tensor, client_weights)
5      )
6      for x, y in dataset:
7          with tf.GradientTape() as tape:
8              tape.watch(client_weights)
9              outputs = model.predict_on_batch(
10                 model_weights=(client_weights, ()),
11                 x=x,
12                 training=True
13             )
14             loss = model.loss(output=outputs, label=y)
```

```
        grads = tape.gradient(loss, client_weights)
        optimizer_state, client_weights = optimizer.next(
            optimizer_state, weights=client_weights, gradients=grads
        )
    delta = [ w_c - w_s
              for w_c, w_s in zip(client_weights, initial_weights.trainable) ]
    return tff.learning.models.ModelWeights(
        trainable=delta, non_trainable=())
```

4.4.3 Model Updates Aggregation

The client model updates are then processed by computing it's mean using the TFF function $tff.federated_mean$ resulting in

$$M_0 \leftarrow M_0 + \frac{1}{n}\sum_{i=1}^{n}\delta_i$$

As illustrated in Figure 4.2, in a scenario where the federated environment consists of two clients, the updated global model weights are computed as the average of the weights from both clients combined with the previous global model weights.

Global Model Weights	1	1	1
Client 1 Model Updates	5.0	1.0	2.0
Client 2 Model Updates	1.0	3.0	2.0
Updated Global Model Weights	1 + (5+1) / 2 = 4	1 + (1+3) / 2 = 3	1 + (2+2) / 2 = 3

Fig 4.2 - Model weights delta aggregation - Example

Once the server calculates the mean of the local deltas, it updates the global model by directly adding these aggregated deltas to the current trainable weights which results in a new global model.

```
mean_client_weights = tff.federated_mean(updated_client_weights)
```

```
def add_delta_fn(w, d):
    return tff.learning.models.ModelWeights(
        trainable=[wt + dt for wt, dt in zip(w.trainable, d.trainable)],
        non_trainable=w.non_trainable)
```

To support scenarios in which certain clients should have more influence over the global model, we introduced a configurable scale factor. This value modifies the magnitude of a client's model update before it is aggregated by the server. Conceptually, a scale factor $k = 3$ means that the client's update is multiplied by 3, effectively giving it the same impact as three normal clients during aggregation.

When partitioning the dataset across clients, a scale factor is assigned to each client and stored alongside its data. This value is then used during the training process, just before the client transmits its update. Notably, the aggregation step on the server remains unchanged as it simply receives and averages the incoming updates, unaware of any scaling that may have occurred locally.

4.5. Adversarial Scenarios

We now present an alternative setting with *malicious* participants. There are multiple ways to characterize the influence of an adversary in such a system, but for the sake of clarity in the results presented, we focus on the introduction of t additional participants whose objective is to disrupt the federated learning process. For simplicity, we restrict our analysis to the case $t = 1$, while noting that the generalization to larger values of t is straightforward Instead of replacing an existing benign client with a malicious one, we opted to introduce an additional client. This approach better supports the study of robust mechanisms for mitigating such attacks. In this way, if the proposed mechanism successfully excludes the malicious inputs, the resulting aggregation would be equivalent to the scenario with ten benign clients.

These participants will have access to the special dataset d_T. This allows for the worst-case adversary: one that is knowledgeable of what data will be used to validate the training, and can select its inputs to the aggregation function accordingly. The reasoning for this decision is not purely theoretical: if one considers the goal of an adversary to exploit a vulnerability and attack an IDS; then he is perfectly aware of the trace that his vulnerability will leave in said system.

Throughout the federated learning rounds, the malicious client follows the standard federated learning communication flow, with the distinction of submitting manipulated model updates. Two types of adversarial strategies were implemented, namely data poisoning attacks and model poisoning attacks.

4.5.1 Data Poisoning

In Data Poisoning attacks, the most disruptive attack would be when a client which in this case is a malicious one, modifies it's dataset he holds in a way to flip all its labels, thus inducing the All labels flipping attack aiming completely induce the IDS in error by classifing bening traffics as malign and the opposite way around.

To perform this:

- p_{11} will simply train over d_T with flipped labels to M_t, and produce $M_t - M_l$ as its output. This entails training a model that tries to characterize all benign traffic as malign and vice-versa.

4.5.2 Model Poisoning

To achieve a more disruptive attack, model poisoning occurs when a client deliberately modifies its training outputs and applies an operation, often multiplication, to amplify their effect. Combined with label flipping, this can drive the model's learning in the opposite direction. If nine clients guide the model toward an optimal solution, a single malicious client can inject conflicting updates, preventing convergence to a high-performing or near-perfect state.

In this case:

- $\mathcal{A}_2$ will proceed similarly to $\mathcal{A}_1$, but will multiply the weights produced in d_t by a constant value c.

The former attack would allow for a more silent attack, but since the adversary is only one in ten, its impact on the federated model is expected to be low. The model poisoning attack, in contrast, by artificially increasing the delta values return have a more impact over the federated model performance. In this setting, we let the function Train denote the standard training of the model, and Agg be the averaging of model weights.

The reasoning for considering such a behavior is that it intuitivelly captures the most straightforward poisoning against an IDS: take a set of traces that correspond to attacks that $\mathcal{A}$ is willing to execute, and tag them as benign, to trick the federated model to learn the inverse. This also represent the idea of *script kidies* attacks, which are considered to be the ones with the most presence as it's based on already existing payloads of malware that can be easily deployed and targeted against any given network.

4.6. Robust Aggregation

To attempt and mitigate attacks against the standard federated learning enviroment such as the one implemented, we must focus and harden the aggregation function as it is the layer that receiving and handles the potentialy maliciously crafted updates from the clients.

Concretely, our goal is to instantiate Agg as a function that behaves as similarly as possible to the averaging of values, while excluding blatant attempts to manipulate the system. The ideal result for this is to create a fucntiona that no matter the attack the following model would always be the same$M_f = M'_f$. This would entail that a setting where adversarial manipulation occurs is equivalent to that where no such adversary exists.

Before exploring potential methods for this robust function, we must first examine the behavior of the current aggregation function when different clients submit their updates. For example, 4.3 illustrates a scenario in which four clients send their updates, while the global model remains at the center due to the non existence of a delta. This serves as a simplified visualization of the federated learning process we have defined: the central point represents the global model distributed to the clients, the blue and red points represent the delta updates that attempt to shift the global model toward a new position.

In practice, neural networks operate in a very high-dimensional parameter space. However, for the sake of interpretability, we illustrate the concept using a two-dimensional example to facilitate understanding.

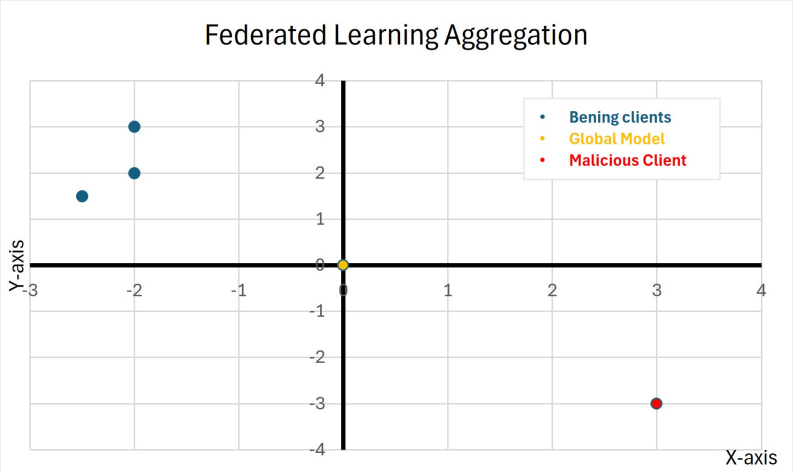


Fig 4.3 - 2D illustration of client updates, with the global model (center), benign updates (blue), and a malicious update (red)

Our current approach of aggregation is based on unweighted Federated Averaging algorithm that is basically a standard averaging of the clients updates where all contributions have an equal impact. Using this method, the delta that would be added to the global model would be the one marked as green in 4.4 that shows that the malicious client denoted by the color red by contributing with his updates prevents the model to reach better states, such as deltas more close to the contribution of the bening clients.

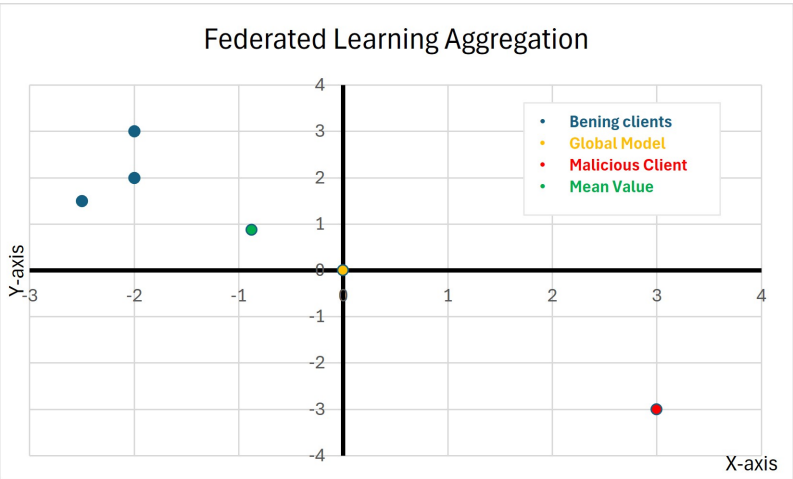


Fig 4.4 - Impact of Client Updates in the Aggregation

From the diagram above, clearly client red is an outlier and this reasoning can be the trigger for creating a robust aggregator.

From the analysis over state of the art aggregation functions that were considered more

secure against the presence of outliers, we can see that the TFF framework already provides a built in function to deal with it: $tff.learning.robust_aggregator$. This function ”zeroes out extremely large values for robustness to data corruption on clients, and clips in the L2 norm to moderately high norm for robustness to outliers.”. The function receives three parameters that are:

- Zeroing - Boolean - Adaptive zeroing for data corruption mitigation.
- Clipping - Boolean - Adaptive clipping in the L2 norm for robustness.
- Weighted - Boolean - Mean weighted.

This function is conceptually similar to the one described in [37], where a threshold determines whether a client's updates are acceptable. In [37], if the updates exceed this threshold, they are clipped—that is, scaled down so that their magnitude is bounded by the maximum limit.

In our TFF implementation, we can choose whether to clip the updates or to zero them out. Zeroing means completely disregarding the offending updates, effectively removing their influence from the aggregation process.

Since our goal is to fully mitigate the impact of malicious clients and to achieve a model whose performance is indistinguishable with or without such clients, we opted for the zeroing strategy.

```
aggregator = robust_aggregator(
    zeroing=False,
    clipping=True,
    weighted=False
)
```

Regarding the weighted option, we did not explore it in detail as its underlying logic was not explicity described in the Tensorflow Federated documentation we decided to not pursue such direction. Nevertheless, we interpret it as the standard federated learning aggregation function in which the data amount of each client influenced the impact of their updates on the global model. In our simulation, this approach could be reasonable since we have accurate knowledge of each client's dataset size. However, in realistic scenarios, this information is typically reported by the clients themselves. This opens the possibility

for a malicious client to misreport its dataset size in order to disproportionately influence the global model's performance.

One important limitation is that this function only works at the high-level TFF API. For our low-level implementation, we must explicitly re-define this logic to replicate the same behavior.

In the `next_fn` function defined earlier, we introduce additional steps to implement **robust aggregation** using a secure thresholding mechanism.

When clients send their model updates (deltas), instead of aggregating them directly via the mean, we first compute the **L2 norm** of each client's update. This results in a single scalar per client, representing the magnitude of their model update. Using these values, we compute the **mean** and **standard deviation**, which respectively quantify the central tendency and dispersion of the update magnitudes.

Next, we define a threshold that serves as an upper bound for allowed client updates. This threshold is calculated as:

$$\text{Threshold} = \mu + (k \cdot \sigma)$$

where:

- μ is the mean of the L2 norms,
- σ is the standard deviation of the L2 norms,
- k is a tunable sensitivity constant.

Since k is a constant, it is important to know which value can balance security and usability. A lower value would be regarded as more secure than a high value because more updates would be blocked, which would increase the likelihood of zeroing a malicious actor that is distant in terms of its delta contributions when compared to its peers of the federated learning process. In order to achieve this, we conducted a number of experiments using various k values and came up with a value that could manage a robust value for preventing fraudulent contribution while also somewhat balancing the overall performance of the global model. In this instance, the constant 3 was the value discovered. In a perfect setting, the constant would be as low as possible so that, in a scenario with 11

clients (10 benign and 1 malicious), it would behave exactly as it would if there were just 10 benign clients, indicating that the attack had been mitigated. [1]

Each client's update is then processed as follows:

- If the client's L2 norm is less than or equal to the threshold, the update is **kept** unchanged.
- If the client's L2 norm exceeds the threshold, the update is **zeroed out**, meaning it contributes nothing to the global aggregation.

Once this filtering is complete, we compute the mean of the remaining (non-zeroed) updates and apply it to update the global model.

We also made a declaration that specified the mean and threshold for each round, as well as each client's l2 norm and a value that indicated whether they were zeroed or kept and taken into consideration for the global model contribution, so that we could see what the server was zeroing or maintaining.

[1] This approach represents a simplified strategy. The selection of such a constant is inherently heuristic, potentially susceptible to adversarial manipulation, and, in an ideal design, would not remain fixed throughout the entire federated learning process.

5. Results and Discussion

This chapter showcases the experimental results obtained by following the methodology described in the Implementation chapter. Our results aim to demonstrate the effectiveness of the federated learning systems including a performance assessment of federated learning under different adversarial scenarios.

5.1. Experimental Setup

All experiments were conducted on a Unix-based operating system, specifically Ubuntu 22.04.5 LTS, using Python version 3.10.12. The implementation relied on the following libraries and corresponding versions: TensorFlow 2.14.1, TensorFlow Federated 0.87, NumPy 1.25.2, Pandas 2.2.3, and scikit-learn 1.6.1. Additionally, all experiments were done considering a fixed seed, so the executions were done in a deterministic manner so a consistent analysis could be possible. An integer value (2025) was set as the seed for all operations that normally use random values, including data preprocessing (shuffling and splitting into folds) and building the neural network model, to keep the process deterministic.

The hardware configuration consisted of a multi-core CPU with 16 processing cores based on the x86_64 architecture, 64 GB of RAM, and a dedicated NVIDIA GeForce RTX 3090 GPU. This setup ensured sufficient computational capacity for both training and evaluation phases, including the execution of the hyperparameters search and the execution of multiple federated learning processes. The simulation of federated learning process with ten clients, each managing its own dataset and independently training a local neural network model, effectively results in ten separate models being trained concurrently. This parallel execution substantially increases memory usage, as multiple model instances and their associated data must be held in memory simultaneously. Consequently, a high-capacity RAM configuration was essential to support the computational demands of this environment.

The choice of hardware and software components is reported here to provide context regarding the computational constraints under which the experiments were executed.

5.2. Centralized vs Decentralized

The neural network model built using the result of the hyperparemters search was trained by following a stratified group Kfold approach, where the dataset is divided in 5 similar parts where 4 are for training purposes while the remaining is preserved for its performance testing. This happens in a iterative manner until there are no further combinations of folds. This results in a neural network model which performs relativetly well achieving an accuracy of 94% and a Loss of 0.21, however to compare and have a better judgement over the model capabilities we focus on key metrics that were also explored in [16]. As shown in the intermediate results presented in the implementation chapter (see Table 3.3), the described NN model achieves an accuracy of 97.10% in Tranco TLS 1.3 dataset. For the MS TLS 1.3 dataset with short-length certificates, the accuracy is 99.91%, decreasing to 99.90% when long-length certificates are also included. Lastly, for the True Negative Rate metric on the DoHBrw dataset, the model achieves a 96.60% accuracy.

When applying this model to a decentralized environment like federated learning, there was a development limitation in the neural network model integration with *tensorflow federated* library that blocked us from adding the dropout layer between the various inner layers of neurons. Nevertheless, even without the dropout layer, which prevents the model from overfitting in a federated learning scenario made up of ten different clients, the model's performance is even higher in some of the metrics examined. As demonstrated by the work done in [41] Federated Learning also achieve better results comparing to the centralized approach for several metrics.

Figure 5.1 shows the machine learning model proposed in [16], the Neural Network Centralized model, the FL model built with the default Federated Learning functions of TFF, and finally our custom FL model implemented with our own functions.

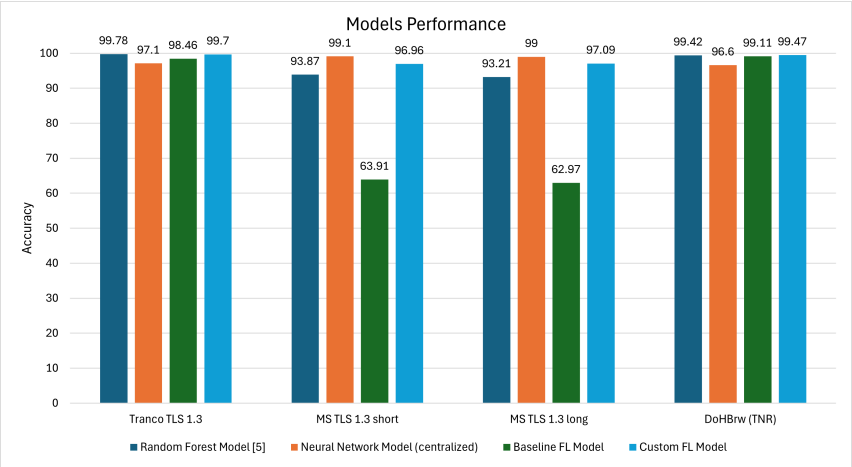


Fig 5.1 - Machine Learning Model Comparisons

The centralized neural network model achieves performance comparable to the one reported in [16], but it substantially outperforms it in the correct classification of targets within the Metasploit datasets, thus proving to be a stronger model in this scenario.

When extending this approach to a federated learning environment, two different strategies were tested: the standard implementation provided by TFF, which relies on predefined high-level functions, and our custom implementation.

For the Tranco TLS 1.3 and DoHBrw datasets, which are dataset composed of only benign traffic, the Baseline FL Model achieves higher accuracy than the centralized neural network model, which is a promising result since it can better classify non malicious traffic. However, for the MS TLS 1.3 short and MS TLS 1.3 long datasets, which are composed of only malicious traffic, its performance drops drastically, with accuracies of only 63.91% and 62.97%, respectively. Due to its inability to accurately categorize malicious traffic, the Baseline FL Model performs significantly worse than the centralized model, according to the data. C2 communications could go unnoticed because of this limitation. This weakness is especially significant because identifying malicious traffic is significantly more crucial than getting high accuracy on benign traffic. Detecting ongoing attacks is directly impacted by a lack of visibility in these situations.

These results indicate that the standard TFF implementation is not a reliable solution in this context. No further conclusions or insights could be drawn to explain the model's low performance under this setting, as TFF operates at a high level and offers limited configuration options.

In contrast, by implementing custom functions, the model shows improved performance

being much more consistent in the displayed metrics. The model avoids the failures observed in the standard version and achieves accuracy values very close to the centralized model. On the MS datasets, it reaches 96.96% (short) and 97.09% (long), which is only about 2% below the centralized version. More importantly, on Tranco TLS 1.3 and DoHBrw, it actually surpasses the centralized model, achieving improvements of 2.6% and 2.87%, respectively.

These findings highlight that, when properly implemented, federated learning provides a viable and effective solution for collaborative learning across multiple parties, as it can achieve performance comparable to centralized approaches, where all data is stored in a single location and can be fully leveraged without restrictions.

Despite the high effectiveness and performance of Federated Learning models, it remains essential to evaluate them in environments beyond the ideal case where all clients are benign. In a decentralized setting, it cannot be assumed that the environment is secure or that all participants act in a benign manner trying to improve a good model. In fact, an attacker may be interested in such a decentralized learning system because, by joining the federated learning process and introducing specific data, it may introduce flaws in the machine learning model to avoid detection of specific attack patterns of its own. This could be a preemptive attack that manipulates the learning process so that the attacker's patterns are later treated as benign, effectively blinding the other participants' systems to the actual intrusion.

Therefore, evaluating how these models perform in the presence of malicious clients is crucial.

5.3. Adversarial Scenarios

To evaluate the robustness of the federated learning model against adversarial clients, we defined three different scenarios to capture the exact impact of malicious activity on the system:

1. Baseline: 10 benign clients.
2. Data Poisoning Attack: 10 benign clients + 1 malicious client performing data poisoning attack
3. Model Poisoning Attack: 10 benign clients + 1 malicious client performing model poisoning attack

In the baseline scenario, all 10 clients are benign, corresponding to the performance previously reported for the custom federated learning model.

In our initial experiments, the federated learning model built with the TFF high-level API performed poorly, even under benign conditions. As a result, we excluded it from adversarial evaluation.

The data poisoning attack scenario introduces an additional client which, by leveraging the same dataset stored in the central server of the federated learning environment (used for testing purposes), flips all labels. As a result, the malicious client trains its local model on corrupted data, driving the learning process in the opposite direction.

Lastly, in the model poisoning attack, the malicious client not only corrupts its local dataset in the same way as the previous attack but also amplifies its model updates before transmitting them to the central server. Specifically, its update vectors are multiplied by a constant factor of 10. This manipulation increases its relative contribution, making it equivalent to approximately 10 clients, thereby exerting a disproportionately large influence over the aggregation process. To simulate this behavior, each client is assigned a contribution parameter, which is normally set to 1. In the case of the malicious client, this parameter is increased to 10.

The results of these experiments are summarized in Figure 5.2

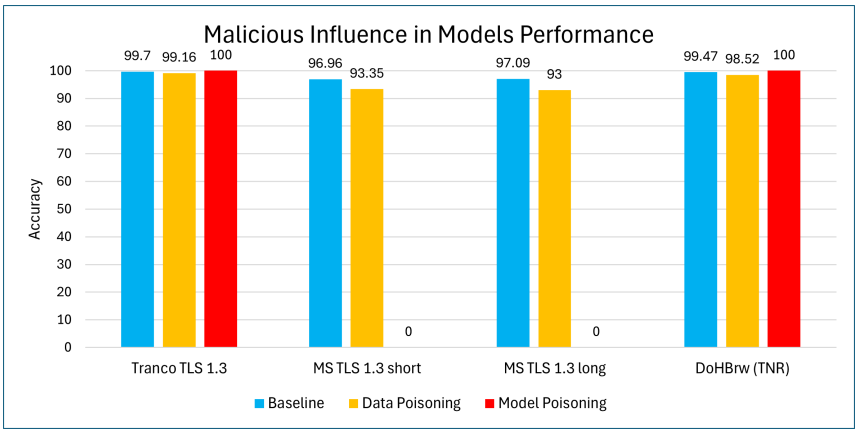


Fig 5.2 - Model Performance

As shown, even under a data poisoning attack with only 1/11% malicious participation, the impact is significant. All metrics exhibit reduced accuracy, with the effect being particularly evident in the MS datasets, where performance drops from approximately 97% to around 93%. In the case of a model poisoning attack with the same proportion of malicious clients,

the outcome is far more severe where the federated learning model completely collapses failing to make correct predictions in both the MS TLS 1.3 short and long datasets thus achieving a 0% accuracy. In a scenario where the dataset is balanced, flipping the class labels would simply result in a model that learns the inverse mapping of the ground truth. However in this case, the data is highly imbalanced: before the label flipping the benign class contained 40,172 rows, while the malicious class contained 39,1355, meaning the malicious class had almost 10 times the amount of the benign class. After flipping, the malicious class becomes the minority, and the majority of the samples are now assigned to the benign class. Consequently, the model will train mostly with benign data and by further amplifying his updates, his contribution results in the observed behaviour where no malicious traffic is correctly classified.

These results clearly demonstrate that, although the model performs effectively under benign conditions, it lacks robustness against adversarial scenarios, showing vulnerability even to relatively simple data poisoning attacks.

5.4. Robust Aggregation

We created a robust aggregator, as described in Section 4, to address the typical federated learning model's lack of resistance against adversarial behavior with the federated averaging algorithm. The variance seen in each client update can be used to determine an acceptance threshold by creating and setting up a reliable aggregator that uses the mean and standard deviation to identify outlier client updates. In this manner, outlier updates would be eliminated, preventing any effect on the model's performance.

To evaluate the impact of implementing a robust aggregator, we structured the analysis according to the previously defined scenarios:

1. Robust aggregator in Baseline: 10 benign clients.
2. Robust Aggregator in Data Poisoning Attack: 10 benign clients + 1 malicious client performing data poisoning.
3. Robust Aggregator in Model Poisoning Attack: 10 benign clients + 1 malicious client performing model poisoning.

Below, in Figure 5.3, it is presented the metrics we have been focusing for evaluation purposes under the three scenarios described above, already incorporating the robust aggregator.

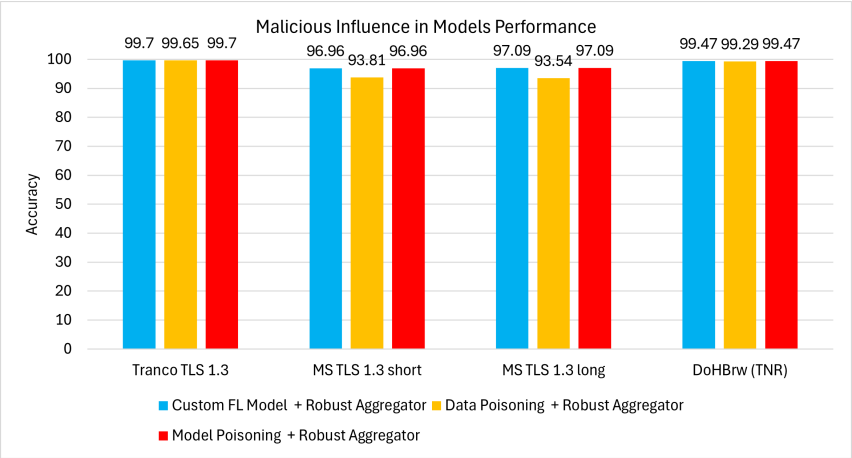


Fig 5.3 - Model Performance with Robust Aggregator under Malicious Presence

In comparison to the results displayed in 5.2, both the model with and without the robust aggregator in the scenario where all clients are benign perform exactly the same, meaning that all traffic was classified exactly in the same manner for the metrics encompassed. This proves that the robust aggregator built on top of the original model doesn't impact optimal scenarios where no malicious presence exists, thus maintaining the high level of accuracy for all the metrics.

When the malicious client attempts data poisoning attacks, the robust aggregator provides a noticeable improvement in comparison to the lack of the robust aggregator. The accuracy increases overall because the aggregator can sometimes detect outlier clients, though not in every round. For the benign datasets Tranco TLS 1.3 and DoHBrw, the accuracy increases by 0.49% and 0.77% respectively. As for the MS datasets, there is an average increase of 0.50%. From Table 5.1, it is evident that in the first two rounds client updates diverge more strongly, as local models are still far from the global model, leading to higher variability. In these rounds the updates remain within the defined threshold. After the third round, however, the benign clients' updates begin to converge, making the malicious client stand out more clearly as an outlier. Its updates exceed the threshold and are discarded, as shown in red.

By the final round, benign client updates have decreased by roughly a factor of five compared to the first round, indicating that local training produces models that closely align with the global model. In contrast, although the malicious client's updates also decrease in the final round compared to the first, an analysis of the intermediate rounds shows that they remain relatively stable, averaging around 30. This consistency suggests that the

model trained on corrupted data does not converge with the global model but instead maintains a persistent distance from it.

Round	Std Dev	Threshold	Benign (Client 0–9)	Malicious (Client 10)
1	3.0993	71.9493	Norm ≈ [61–67]	Norm = 54.1793
2	1.5846	58.9554	Norm ≈ [51–55]	Norm = 58.5430
3	5.3550	62.8807	Norm ≈ [42–46]	Norm = 63.4761
200	4.7978	29.1342	Norm ≈ [12–14]	Norm = 29.7836

Tab 5.1 - Federated Learning Rounds under Data Poisoning Attack

Regarding model poisoning attacks, the behavior is rather different. In this scenario, the malicious client scales its updates after training with flipped labels. From the server's perspective, such updates appear as clear outliers compared to clients as the updates are disproportionality from the rest of the clients. Consequently, the robust aggregator discards them effectively, mitigating the attack and maintaining the same performance as in the benign-only scenario (Table 5.2). Under the model poisoning scenario, benign clients exhibit the expected convergent behavior, progressively aligning both with one another and with the global model. The malicious client, however, follows a different result. Since model poisoning deliberately manipulates the scale of the updates, the norm representing the delta between its local model and the global model increases steadily over time. This forced divergence contrasts with the natural convergence of benign clients, meaning that while the benign models iteratively refine towards the shared global objective, the malicious client's model drifts progressively farther away as training advances.

Round	Std Dev	Threshold	Benign (Client 0–9)	Malicious (Client 10)
1	145.0604	544.5493	Norm ≈ [61–67]	Norm = 568.0628
2	195.4090	701.5712	Norm ≈ [51–55]	Norm = 701.5712
3	244.7010	855.5562	Norm ≈ [42–46]	Norm = 855.5562
200	3314.3459	11003.9395	Norm ≈ [12–13]	Norm = 11541.7832

Tab 5.2 - Federated Learning Rounds under Model Poisoning Attack

5.5. Critical Learning

Another vector of study in this thesis was the concept of critical learning periods, which refers to periods of time, rounds, where the model is more subject to changes, meaning that during this period of time its more prone to learn and make inferences. We have

attempted to discover the critical learning period of the model by introducing poisoning attacks in specific rounds.

In our experiments, model poisoning attacks were restricted to predefined ranges of rounds in order to observe where a CLP might occur. More concretely, attacks were applied within the following intervals: [0–-19], [20–-39], [40–-59], [60–-79], [80–-99], and [100–-119]. These attacks, limited to specific intervals of rounds, simulate a temporary corruption of the client model. This scenario reflects the case where an adversary exploits a newly discovered vulnerability to infiltrate the client system without being detected. However, once the vulnerability is patched, the client model recovers and is again able to correctly identify such malicious activity.

To assess the impact of the attack, the objective was not to analyze individual evaluation metrics in detail but rather to obtain a general indication of the resulting accuracy degradation. To this end, an aggregated metric was defined as the mean of Tranco TLS 1.3, MS TLS 1.3 short, MS TLS 1.3 long, and DoHBrw (TNR).

By analyzing the graph in Figure 5.4, we observe a linear trend indicating a reduction in the attack's impact as the number of training rounds increases[1]. In the experiments conducted, the attack applied during rounds [20–39] proved to be the most harmful, reducing accuracy from 98.3% in the benign scenario to 57.5%. In contrast, in the final experiment, where attacks were introduced in rounds [100–119], the model still managed to achieve an accuracy of 94.8%. This behaviour aligns with the work done in [40] where the CLP occurs in the first rounds of the federated learning process. As observed, attacks occurring between rounds 20 and 39,where the CLP is located, cannot be mitigated, even in the absence of malicious activity in later rounds of federated learning. This is because the model has been fundamentally altered, making long-term mitigation impossible. This emphasizes the need to better safeguard the federated learning process during this crucial time by using a robust aggregator with strict controls that prevent the model from performing many significant adjustments. Alternatively, if the critical learning period were concealed through offline training and the global model were initialized in a pre-trained state, we could avoid these consequences because the critical learning period would no longer be exposed to potential malicious activity. The security constraints might be loosened after the CLP is complete to enable the aggregation of more benign outlier updates

[1] This observation represents a simplified approximation and may overlook certain nuances of the learning process. Nevertheless, it offers a pragmatic means of superficially evaluating the impact of poisoning at different stages of training

resulintg in a more efficient and scalable model.

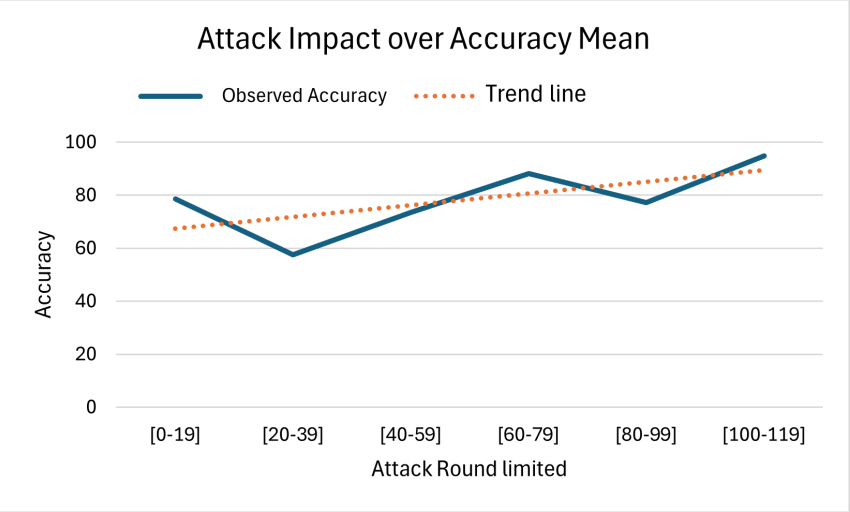


Fig 5.4 - Round limited Attacks Impact over Model Performance

6. Conclusions

In this thesis, we explored the implementation of a decentralized system for building a machine learning model capable of detecting C2 communications within TLS 1.3 traffic. We began by evaluating state-of-the-art traditional machine learning models and then transitioned to deep learning, more concretely, neural networks. The models were then adapted to operate within a FL environment, where each client maintained a local model and iteratively contributed to the construction of a global and transversal model. This approach showed promising results where in some areas even surpassing the same model under a centralized environment.

With this foundation in place, we evaluated the robustness of the system under adversarial settings, focusing on two main attack vectors: data poisoning and model poisoning. Our experiments showed that in a standard FL setup, the global model is vulnerable, as even a small proportion of malicious clients (1 out of 11) significantly degraded performance and resilience.

To counter this effect, we studied state-of-the-art robust aggregation functions suitable for deployment at the FL server, where client updates are collected. Among the top methods, we implemented L2 Norm Clipping, which demonstrated the ability to filter out abnormal client contributions and improve the reliability of the global model.

Finally, we investigated the concept of a critical learning period in Federated Learning and demonstrated that short, well-timed attacks can still introduce great impacts in overall global model performance with reduced computational cost on the adversary side.

6.1. Future Work

As a foundation for this thesis, we divided the dataset in a way that does not perfectly reflect real-world conditions, where the ratio of malicious to benign clients and the types of attacks can vary significantly. By investigating how this model would behave under different data partitioning strategies for the C2 detection, we can gain deeper insights into their robustness and practical applicability.

In addition, the validation data used for the FL model remained constant throughout the experiments. So, in order to better evaluate the FL model performance, it would be interesting to experiment how different validation data would influence the results.

The work presented in this thesis could also be refined by adopting more robust aggregation functions, such as the Geometric Mean. This would enable additional experiments and provide deeper insights into which approaches offer the best defense against poisoning attacks.

Further improvements could be incorporated into the implemented aggregation function, as certain attack vectors were only considered theoretically but not yet explored experimentally. For example, in a scenario with two malicious clients, the first attacker could inject extreme values and noise to influence the threshold calculation, while the second attacker could then provide moderately abnormal updates that evade detection, thereby degrading the performance of the global model.

A possible mitigation for this attack vector would be to calculate the mean and standard deviation iteratively, each time a client contribution is discarded. In this way, outliers could be filtered out one by one until no abnormal clients remained. While this approach increases computational cost and overhead of the server due to more iterations over the values provided by each client, it would theoretically enhance and mitigate such attack vectors.

References

[1] “Malware definition by crowdstrike.” [Online]. Available: https://www.crowdstrike.com/en-us/cybersecurity-101/malware/malware-vs-virus/ [Cited on pages 1 and 5.]

[2] E. Hutchins, M. Cloppert, and R. Amin, “Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains,” *Leading Issues in Information Warfare Security Research*, vol. 1, no. 1, pp. 80–106, 2011. [Cited on page 1.]

[3] “Enisathreatlandscape.” [Online]. Available: https://www.enisa.europa.eu/topics/cyber-threats/threat-landscape [Cited on pages 1 and 7.]

[4] E. Rescorla, “The transport layer security (tls) protocol version 1.3,” RFC 8446, IETF, August 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446 [Cited on page 2.]

[5] A. Kariluoto, A. Pärnänen, J. Kultanen, J. Soininen, and P. Abrahamsson, “Quality of data in machine learning,” *Proceedings of the IEEE 21st International Conference on Software Quality, Reliability Security (QRS) Companion*, pp. 216–221, 2021. [Cited on pages 2 and 9.]

[6] H. McMahan, E. Moore, D. Ramage, S. Hampson, and B. Arcas, “Communication-efficient learning of deep networks from decentralized data,” *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017)*, pp. 1273–1282, 2017. [Cited on pages 2, 12, 13, and 15.]

[7] J. Gardiner, M. Cova, and S. Nagaraja, “Command control: Understanding, denying and detecting - a review of malware c2 techniques, detection and defences,” *Centre for the Protection of National Infrastructure (CPNI)*, 2015. [Cited on page 5.]

[8] J. Postel, “Internet protocol,” RFC 791, IETF, September 1981. [Online]. Available: https://www.rfc-editor.org/rfc/rfc791 [Cited on page 5.]

[9] T. Dierks and E. Rescorla, “The transport layer security (tls) protocol version 1.2,” RFC 5246, IETF, August 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5246 [Cited on page 5.]

[10] “Malware definition by crowdstrike.” [Online]. Available: https://www.crowdstrike.com/en-us/cybersecurity-101/malware/malware-vs-virus/ [Cited on page 5.]

[11] K. Watson, “Operational value of iocs white paper,” *White Paper at U.S. Cybersecurity Infrastructure Security Agency*, 2021. [Online]. Available: https://www.cisa.gov/resources-tools/resources/operational-value-indicators-compromise-white-paper [Cited on page 5.]

[12] “Indicator of compromise usage and examples - crowdstrike.” [Online]. Available: https://www.crowdstrike.com/en-us/cybersecurity-101/threat-intelligence/indicators-of-compromise-ioc/ [Cited on page 5.]

[13] “Indicator of compromise usage and examples - microsoft.” [Online]. Available: https://www.microsoft.com/en-us/security/business/security-101/what-are-indicators-of-compromise-ioc [Cited on page 5.]

[14] S. Mohanad, S. Layeghy, M. Gallagher, and M. Portmann, “From zero-shot machine learning to zero-day attack detection,” *International Journal of Information Security*, vol. 22, no. 4, pp. 495–516, 2023. [Cited on page 6.]

[15] O. Joseph and R. Ajax, “Comparison of traditional vs. ai-based intrusion detection and prevention systems: Efficiency and accuracy,” *ResearchGate*, 2025. [Cited on page 6.]

[16] D. Barradas, C. Novo, B. Portela, S. Romeiro, and N. Santos, “Extending c2 traffic detection methodologies: From tls 1.2 to tls 1.3-enabled malware,” *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 181–196, 2024. [Cited on pages 6, 7, 9, 23, 27, 28, 46, and 47.]

[17] L. Gehri, R. Meier, D. Hulliger, and V. Lenders, “Towards generalizing machine learning models to detect command and control attack traffic,” *Proceedings of the 15th International Conference on Cyber Conflict (CyCon) — “Meeting Reality”*, 2023. [Cited on page 9.]

[18] J. Gardiner and S. Nagaraja, “On the security of machine learning in malware cc detection: A survey,” *ACM Computing Surveys*, vol. 49, no. 3, p. 59, 2016. [Cited on page 9.]

[19] P. Wang, E. Fan, and P. Wang, “Comparative analysis of image classification algorithms based on traditional machine learning and deep learning,” *Pattern Recognition Letters*, vol. 141, pp. 61–67, 2020. [Cited on page 9.]

[20] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” *Proceedings of the 7th International Conference on Learning Representations (ICLR 2019)*, 2019. [Cited on page 11.]

[21] M. Islam, G. Chen, and S. Jin, “An overview of neural network,” *American Journal of Neural Networks and Applications*, vol. 5, no. 1, p. 7–11, 2019. [Cited on page 11.]

[22] M. Bejani and M. Ghatee, “A systematic review on overfitting control in shallow and deep neural networks,” *Artificial Intelligence Review*, vol. 54, no. 3, pp. 6391–6438, 2021. [Cited on pages 11 and 12.]

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014. [Cited on page 12.]

[24] P. Qi, D. Chiaro, A. Guzzo, M. Ianni, G. Fortino, and F. Piccialli, “Model aggregation techniques in federated learning: A comprehensive survey,” *Future Generation Computer Systems*, vol. 150, pp. 272–293, 2024. [Cited on pages 12, 15, and 20.]

[25] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data.” PMLR, 2017, pp. 1273–1282. [Cited on page 15.]

[26] S. Reddi, Z. Charles, M. Zaheer, Z. Garrett, K. Rush, J. Konečnỳ, S. Kumar, and H. B. McMahan, “Adaptive federated optimization,” *Proceedings of the International Conference on Learning Representations*, 2021. [Cited on page 15.]

[27] Z. Cheng, X. Huang, P. Wu, and K. Yuan, “Momentum benefits non-iid federated learning simply and provably,” *ICLR 2021*, 2023. [Cited on page 15.]

[28] Z. Çıplak, K. Yıldız, and S. Altınkaya, “Fedetect: A federated learning-based malware detection and classification using deep neural network algorithms,” *Arabian Journal for Science and Engineering*, vol. 50, pp. 16 107–16 134, 2025. [Cited on page 15.]

[29] J. Mills, “Federated machine learning in edge computing,” *University of Exeter*. [Cited on page 15.]

[30] C. Tham, L. Yang, A. Khanna, and B. Gera, “Federated learning for anomaly detection in vehicular networks,” *Proceedings of the 2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*, pp. 1–6, 2023. [Cited on page 15.]

[31] X. Yao, K.and Cao, “Uav land classification method based on federated learning,” *Academic Journal of Computing & Information Science*, vol. 5, no. 7, pp. 73–78, 2022. [Cited on page 15.]

[32] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *Proceedings of the International Conference on Learning Representations*, 2015. [Cited on page 16.]

[33] H. Neto, J. Hribar, I. Dusparic, D. Mattos, and N. Fernandes, “A survey on securing federated learning: Analysis of applications, attacks, challenges, and trends,” *IEEE Access*, vol. 11, pp. 41 928 – 41 953, 2023. [Cited on pages 16 and 17.]

[34] A. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo, “Analyzing federated learning through an adversarial lens,” *Proceedings of the 36th International Conference on Machine Learning*, pp. 634–643, 2018. [Cited on page 16.]

[35] V. Rey, P. Sánchez, A. Celdrán, and G. Bovet, “Federated learning for malware detection in iot devices,” *Computer Networks*, vol. 204, p. 108693, 2022. [Cited on pages 17, 18, 20, and 21.]

[36] K. Pillutla, S. Kakade, and Z. Harchaoui, “Robust aggregation for federated learning,” *IEEE Transactions on Signal Processing*, vol. 70, pp. 1142–1154, 2022. [Cited on pages 17, 20, 21, and 22.]

[37] Z. Sun, P. Kairouz, A. Suresh, and H. McMahan, “Can you really backdoor federated learning?” *2nd International Workshop on Federated Learning for Data Privacy and Confidentiality at NeurIPS 2019*, 2019. [Cited on pages 20 and 42.]

[38] E. Weiszfeld, “Sur le point par lequel la somme des distances de n points donnves est minimum. tohoku mathematical journal, 43, 355-386.” *Tōhoku Mathematical Journal*, vol. 43, pp. 355–386, 1937. [Cited on page 21.]

[39] G. Yan, H. Wang, and J. Li, “Critical learning periods in federated learning,” *arXiv preprint arXiv:2109.05613*. [Cited on page 22.]

[40] G. Yan, H. Wang, X. Yuan, and J. Li, “Enhancing model poisoning attacks to byzantine-robust federated learning via critical learning periods,” *Proceedings of*

the 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), pp. 496–512, 2024. [Cited on pages 22 and 53.]

[41] R. Hsu, Y. Wang, C. Fan, B. Sun, T. Ban, T. Takahashi, T. Wu, and S. Kao, "A privacy-preserving federated learning system for android malware detection based on edge computing," *15th Asia Joint Conference on Information Security (AsiaJCIS)*, 2020. [Cited on page 46.]