Arquitetura de detecção de intrusão por anomalias com Federated Learning em redes IoT

Samuel Carlos Menêses Soares

Brasília, Setembro de 2022

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

Trabalho Final de Graduação

Arquitetura de detecção de intrusão por anomalias com Federated Learning em redes IoT

Samuel Carlos Menêses Soares

Relatório submetido ao Departamento de Engenharia
Elétrica como requisito parcial para obtenção
do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Prof. PhD. Rafael Timóteo de Sousa Júnior, ENE/UnB
Orientador

MSc. Francisco Lopes de Caldas Filho, ENE/UnB
Co-orientador

PhD. Fabio Lucio L. de Mendonça, ENE/UnB
Examinador

Agradecimentos

Com mais uma etapa se encerrando, nada mais justo que agradecer a todos aqueles que estiveram presentes durante toda minha jornada dando o máximo de apoio possível. Agradeço a toda minha família por sempre estarem ao meu lado com todo carinho e amor, sempre me ajudando nos momentos mais difíceis. Agradeço em especial ao meu pai Antônio Carlos Soares, minha mãe Sandra Maria Menêses Soares e ao meu irmão Carlos Eduardo M. Soares pelo suporte familiar em todos esses anos de universidade.

Durante esta jornada, muitos amigos também estiveram presentes na caminhada compartilhando dos momentos divertidos, tristes e difíceis que a universidade fornece. Agradeço em especial o William Ícaro, Rickson Ribeiro, Pedro Henrique Souza, Guilherme Luís, Pedro Aguiar, Hugo Calado e Mateus Leite por tudo que passamos juntos nesta trajetória.

Agradeço aos amigos de longa data por estarem comigo nos momentos importantes durante todos esses anos. Em especial o Gustavo Hossoe, Victor Dirceu, Carlos Eduardo Mancini, Thiago Villa, Dayany Yamamoto, Leonardo Inacio, Pedro Lima e Vinicius Carvalho.

Gostaria de agradecer a todos os professores do curso que me ensinaram tudo que precisei para realizar este trabalho de conclusão. Agradeço em especial ao meu co-orientador Francisco Lopes e ao meu professor orientador Rafael Timóteo por todo suporte, orientação e instrução que deram de bom grado durante a confecção deste projeto. Agradeço também ao pessoal do laboratório UIoT por toda ajuda que foi dada.

Agradeço o apoio técnico e computacional do Laboratório de Tecnologias para Tomada de Decisão - LATITUDE, da Universidade de Brasília, que conta com apoio do CNPq - Conselho Nacional de Pesquisa (Outorgas 312180/2019-5 PQ-2 e 465741/2014-2 INCT em Cibersegurança), do Ministério da Economia (Outorgas 005/2016 DIPLA e 083/2016 ENAP), do Conselho Administrativo de Defesa Econômica (Outorga CADE 08700.000047/2019-14), da Advocacia Geral da União (Outorga AGU 697.935/2019), do Departamento Nacional de Auditoria do SUS (Outorga DENASUS 23106.118410/2020-85), da Procuradoria Geral da Fazenda Nacional (Outorga PGFN 23106.148934/2019-67) e dos Decanatos de Pesquisa e Inovação e de Pós-Graduação da Universidade de Brasília (Outorga 7129 FUB/EMENDA/DPI/COPEI/AMORIS).

Agradeço especialmente a Deus por todo o cuidado e iluminação que foram dados durante todas as etapas da faculdade. Em muitos momentos de desespero, momentos que pareciam que nada iria dar certo, buscar a Deus trouxe as respostas, o sossego e a paz nas horas que eu mais precisava. A jornada não é fácil, ela é árdua e penosa. E é por isso que ter pessoas ao seu lado como os familiares, amigos, professores e principalmente Deus, é necessário para que tudo dê certo no fim.

Agradeço a todos!

Samuel Carlos Menêses Soares

Resumo

O mercado tecnológico de Internet das Coisas (IoT) teve um grande avanço nos últimos anos, possuindo cada vez mais dispositivos capazes de realizar tarefas de forma inteligente tanto em ambientes industriais como em ambientes domésticos. A composição desses dispositivos apresenta uma grande simplicidade em termos de arquitetura por questões de custo-benefício e praticidade, porém, essa questão traz problemas de segurança e privacidade para as redes IoT devido à quantidade de vulnerabilidades que vão sendo identificadas e exploradas por conseguinte. Com isso, métodos foram sendo aplicados para solucionar esses problemas de exploração de vulnerabilidades, como a utilização de sistemas de detecção e prevenção de intrusão para identificar possíveis ataques na rede e realizar ações preventivas em tempo real. Um dos métodos utilizado para aplicar esse tipo de sistema é a detecção por anomalias, consistindo na observação de padrões de comunicação para distinguir os tráfegos por sua natureza, sendo altamente utilizado com técnicas de Aprendizado de Máquina e Inteligência Artificial (AI) por questões de inovação e automatização. Então, será apresentada neste trabalho uma solução de sistema de detecção de intrusão por anomalias para redes IoT com o intuito de identificar tráfegos maliciosos e distingui-los de tráfegos benignos, contendo a utilização de métodos de aprendizado de máquina, como o *Federated Learning*, para realizar o aprendizado da detecção de forma distribuída a partir da contribuição de cada dispositivo pertencente à rede IoT.

Palavras-chaves: Federated Learning, Inteligência Artificial, Internet das Coisas, Detecção de Intrusão, Prevenção de Intrusão, Detecção de Anomalias, Aprendizado de Máquina.

Abstract

The Internet of Things (IoT) technological market has had a great advance in recent years, with more and more devices capable of performing tasks intelligently in both industrial and domestic environments. The composition of these devices presents a great simplicity in terms of architecture for cost-benefit and practicality reasons, however, this issue brings security and privacy problems to IoT networks due to the amount of vulnerabilities that are being identified and exploited as a result. Therefore, methods have been applied to solve these vulnerability exploitation problems, such as the use of intrusion detection and prevention systems to identify possible attacks on the network and perform preventive actions in real time. One of the methods that is used to apply this kind of system is anomaly detection, consisting in the observation of communication patterns to distinguish traffic by its nature, being highly used with Machine Learning and Artificial Intelligence (AI) techniques for innovation and automation reasons. So, it will be presented in this paper a solution for anomaly intrusion detection system for IoT networks in order to identify malicious traffic and distinguish it from benign traffic, containing the use of machine learning methods, such as Federated Learning, to perform the detection learning in a distributed way from the contribution of each device belonging to the IoT network.

Keywords: Federated Learning, Artificial Intelligence, Internet of Things, Intrusion Detection, Intrusion Prevention, Anomaly Detection, Machine Learning.

SUMÁRIO

SUMÁRIO 6

LISTA DE FIGURAS 8

1 INTRODUÇÃO 1
1.1 Objetivo Geral 2
1.1.1 Objetivos específicos 2
1.2 Justificativa 3

2 REVISÃO BIBLIOGRAFICA E MARCO TEÓRICO 4
2.1 Marco Teórico 4
2.1.1 Redes IoT 4
2.1.2 Sistema de detecção e prevenção de intrusão 5
2.1.3 Redes Neurais e Deep Learning 7
2.1.3.1 Convolutional Neural Networks 9
2.1.3.2 Métricas de desempenho 10
2.1.4 Federated Learning 12
2.2 Revisão Bibliográfica 15

3 ARQUITETURA PROPOSTA 19
3.1 Modelo Adversário 22
3.1.1 Mirai Botnet 22
3.1.2 Implementação do ambiente de testes 25
3.2 Módulo de Detecção 32
3.2.1 Estrutura básica 32
3.2.2 Suricata 34
3.2.3 Apache Kafka 38
3.2.4 Apache Spark com Structured Streaming 41
3.2.5 Modelo de Treinamento 44
3.3 Implementação do Federated Learning 47
3.3.1 Servidor 49
3.3.2 Clientes 52
3.3.2.1 Divisão do conjunto de treinamento 53
3.3.2.2 Definição dos Modelos de Redes Neurais 56
3.3.2.3 Definição de Parâmetros e Execução do Treinamento 58

4 RESULTADOS E ANÁLISE 62
4.1 Análise do tráfego do Modelo Adversário 62

4.2 Análise de desempenho do Federated Learning 65
4.2.1 Modelo Agregado e Inicialização dos Parâmetros 66
4.2.2 Desempenho de Classificação das Categorias de Comportamento 71
4.2.3 Gráficos de Desempenho 79

5 CONCLUSÕES E TRABALHOS FUTUROS 82

Bibliografia 83

ANEXO A – SCALA CODE 88

ANEXO B – POM FILE 90

ANEXO C – CLIENT CODE 94

ANEXO D – SERVER CODE 99

LISTA DE FIGURAS

Figura 2.1 – Diagrama das classificações de um sistema de detecção de intrusão. Fonte: [57] 6
Figura 2.2 – Estrutura de Redes Neurais conectadas por camadas. Fonte: [1] 8
Figura 2.3 – Estrutura de Redes Neurais conectadas por camadas. Fonte: [53] 10
Figura 2.4 – Exemplo de Matriz de Confusão. Fonte: [61] 11
Figura 2.5 – Arquitetura básica com *Federated Learning*. Fonte: [28] 14
Figura 3.1 – Arquitetura proposta 20
Figura 3.2 – Fluxograma da arquitetura 22
Figura 3.3 – Estrutura lógica do Mirai. Fonte: [52] 24
Figura 3.4 – Estrutura implementada no laboratório. 25
Figura 3.5 – DNS Server na controladora do IDS. 26
Figura 3.6 – *Web Server* levantado no alvo. 27
Figura 3.7 – Página *Web* com os binários. 28
Figura 3.8 – Mirai sendo executado. 29
Figura 3.9 – Servidor CNC na controladora do Mirai. 29
Figura 3.10–Execução do Mirai e ScanListen. 30
Figura 3.11–Binário sendo executado em um Raspberry Pi. 30
Figura 3.12–Conexões do dispositivo infectado. 31
Figura 3.13–Fluxograma do Mirai 31
Figura 3.14–Fluxograma da comunicação do Mirai com a *botnet* 32
Figura 3.15–Fluxograma do Módulo de Detecção 34
Figura 3.16–Lista de sources habilitadas no Suricata. 35
Figura 3.17–Lista de regras do Suricata 36
Figura 3.18–Exemplo de uma captura em JSON. 37
Figura 3.19–Captura do Suricata em execução. 38
Figura 3.20–Arquitetura do Apache Kafka. Fonte: [5] 39
Figura 3.21–Criação do tópico de entrada. 39
Figura 3.22–Execução do Kafka *producer*. 40
Figura 3.23–Execução do Kafka *consumer*. 40
Figura 3.24–Estrutura de *producer* e *consumer* do projeto. 40
Figura 3.25–Execução do Apache Spark. 42
Figura 3.26–Esqueleto do *dataset* no console. 43
Figura 3.27–Saída do *dataset* no console. 43
Figura 3.28–Saída final do *dataset* principal. 44
Figura 3.29–Diagrama do processamento e transformação dos *logs*. 44
Figura 3.30–Modelo de treinamento após o tratamento dos dados. 46
Figura 3.31–Estrutura de captura de novos eventos a serem classificados. 47
Figura 3.32–Estrutura básica do *Federated Learning*. 48
Figura 3.33–Fluxograma do treinamento de FL. 49

Figura 3.34–Bibliotecas utilizadas no servidor. 50
Figura 3.35–Código referente à execução do servidor. 50
Figura 3.36–Estratégias utilizadas. 51
Figura 3.37–Geração dos certificados do SSL. 51
Figura 3.38–Inicialização do servidor. 52
Figura 3.39–Bibliotecas do código dos clientes. 52
Figura 3.40–Distribuição das classes. 53
Figura 3.41–Estrutura do *dataset* após o mapeamento. 54
Figura 3.42–Leitura do *dataset* nos clientes. 55
Figura 3.43–Estrutura do modelo de treinamento por MLP. 55
Figura 3.44–Estrutura do modelo de treinamento por 1D-CNN. 55
Figura 3.45–Dados das camadas referente ao modelo comum. 57
Figura 3.46–Dados das camadas referente ao 1D-CNN. 57
Figura 3.47–Dados de configuração do aprendizado. 58
Figura 3.48–Resultados de saída do treinamento. 58
Figura 3.49–Identificação do cliente 1. 59
Figura 3.50–Identificação do cliente 2. 59
Figura 3.51–Configuração de inicialização dos clientes. 59
Figura 3.52–Cliente pronto para o treinamento. 59
Figura 3.53–Treinamento iniciado pelo cliente 1. 60
Figura 3.54–Treinamento iniciado pelo cliente 2. 60
Figura 3.55–Agregação e avaliação pelo servidor. 60
Figura 3.56–Avaliação dos treinamentos de cada cliente 1. 60
Figura 3.57–Avaliação dos treinamentos de cada cliente 2. 60
Figura 3.58–Recebimento das avaliações no lado do servidor. 60
Figura 4.1 – Volume do tráfego por UDP *Flood*. 64
Figura 4.2 – Volume do tráfego por SYN *Flood*. 64
Figura 4.3 – Inicialização dos parâmetros no modelo comum. 67
Figura 4.4 – Inicialização dos parâmetros no 1D-CNN. 67
Figura 4.5 – Gráfico de métricas dos parâmetros iniciais dos dois modelos. 68
Figura 4.6 – Treinamento completo no modelo comum. 69
Figura 4.7 – Treinamento completo do 1D-CNN. 69
Figura 4.8 – Resultados da agregação no modelo comum. 69
Figura 4.9 – Resultados da agregação do 1D-CNN. 70
Figura 4.10–Gráfico de desempenho da agregação em termos de acurácia. 71
Figura 4.11–Gráfico de desempenho da agregação em termos de perda. 71
Figura 4.12–Primeiro treinamento do cliente 1 no modelo comum. 73
Figura 4.13–Último treinamento do cliente 1 no modelo comum. 73
Figura 4.14–Primeiro treinamento do cliente 1 no 1D-CNN. 75
Figura 4.15–Último treinamento do cliente 1 no 1D-CNN. 75
Figura 4.16–Primeiro treinamento do cliente 2 no modelo comum. 76
Figura 4.17–Último treinamento do cliente 2 no modelo comum. 76

Figura 4.18–Primeiro treinamento do cliente 2 no 1D-CNN. 77
Figura 4.19–Último treinamento do cliente 2 no 1D-CNN. 78
Figura 4.20–Gráfico de desempenho de cada classe no modelo comum do cliente 1. 80
Figura 4.21–Gráfico de desempenho de cada classe no 1D-CNN do cliente 1. 80
Figura 4.22–Gráfico de desempenho de cada classe no modelo comum do cliente 2. 81
Figura 4.23–Gráfico de desempenho de cada classe no 1D-CNN do cliente 2. 81

LISTA DE ABREVIATURAS

Acrônimos

NIDS	Network Intrusion Detection System
HIDS	Host Intrusion Detection System
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
IDPS	Intrusion Detection and Prevention System
UnB	Universidade de Brasília
IoT	Internet of Things
DDoS	Distributed Denial of Service
DoS	Denial of Service
JSON	JavaScript Object Notation
YAML	Yet Another Markup Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
SSH	Secure Shell
SSL	Secure Sockets Layer
HTTP	Hypertext Transfer Protocol
POM	Project Object Model
FL	Federated Learning
ML	Machine Learning
CNN	Convolutional Neural Network
AI	Inteligência Artificial
API	Application Programming Interface
CSV	Comma-separated Values
RPC	Remote Procedure Call
HTTP	HyperText Markup Language
SQL	Structured Query Language
CNC	Command and Control
DNS	Domain Name Server
JAR	Java Archive
PCAP	Packet Capture
RDD	Resilient Distributed Dataset
IDE	Integrated Development Environment
XML	Extensible Markup Language
TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative

1 Introdução

A Internet das Coisas (IoT) [34], é um fenômeno digital que consiste na conexão de dispositivos em uma rede, conseguindo realizar tarefas com alto grau de automação e interatividade com a finalidade de aplicar soluções inteligentes para diversos tipos de infraestrutura, como *smart homes*, geradores de energia, áreas de saúde, área automotiva, dentre outros. O seu avanço no mercado vem ocorrendo de forma intensa e acelerada nos últimos anos, considerado cada vez mais uma opção viável e vantajosa para ser utilizado em sistemas inteligentes. Com essa grande difusão e necessidade de rápido aprimoramento, diversos tipos de dispositivos de micro-arquitetura que apresentam uma estrutura de modelo simplificado surgiram no mercado como sensores, Raspberry Pi, câmeras IP. Pelo fato desses dispositivos possuírem arquiteturas simples, há a presença de diversas vulnerabilidades e maior custo computacional para processos complexos, principalmente na sua utilização para pesquisas envolvendo Inteligência Artificial e *Machine Learning*.

A presença de diversas vulnerabilidades nos dispositivos IoT permitiu a utilização de *malwares* e *botnets* como o Mirai [42] de forma aprofundada quanto a ataques de DDoS, para a aplicação de ataques como envenenamento de dados, ataques distribuídos, ataques de inferências, reconstrução de dados, dentre outras. Então, com a finalidade de identificar esses ataques sendo realizados na rede e aplicar ações preventivas de segurança, evitando possíveis danos aos dispositivos, é possível estruturar um sistema de detecção de intrusão (IDS) [38] e sistema de prevenção de intrusão (IPS) [63]. Estes sistemas são importantes para introduzirem uma camada de segurança nas redes e reduzir as chances das vulnerabilidades serem exploradas por softwares maliciosos. Os tipos de detecção de uma IDS podem ser baseadas em assinaturas, que consiste em avaliar o tráfego com base em um conjunto de regras pré-estabelecidas e armazenadas em um banco de dados, e podem ser baseadas em anomalias [12], consistindo em avaliar o tráfego com base em padrões anteriores de dados previamente coletados referentes a comunicações benignas e também maliciosas para então realizar a classificação desejada acerca da natureza do fluxo de pacotes identificado na rede.

A utilização de Inteligência Artificial (AI) para soluções inteligentes é amplamente difundida nas áreas de pesquisa de redes IoT. Uma aplicação de Aprendizado de Máquina que traz aprimoramentos de segurança e privacidade para os dispositivos durante o treinamento é o *Federated Learning* [37]. Essa implementação consiste em realizar o aprendizado de máquina de *datasets* ou modelos de forma descentralizada em uma rede, tal que cada dispositivo será selecionado a cada rodada do procedimento para fazer o treinamento, apresentando então resultados de forma coletiva a partir de vários clientes distintos. Para com que cada cliente participe das etapas, um servidor previamente preparado coordena e recebe esses treinamentos em formatos de pesos (resultados com base em parâmetros pré-definidos) por parte de cada cliente, para realizar uma agregação e obter o modelo final com as classificações desejadas. O *Federated Learning* é uma opção de AI para descentralizar e garantir maior privacidade durante os treinamentos, envolvendo a participação de múltiplos dispositivos e clientes que utilizarão os seus próprios dados locais para realizar a aprendizagem.

Neste projeto, o objetivo é desenvolver uma arquitetura de detecção de anomalias de tráfegos identificados e obtidos em uma rede IoT a partir de um componente de captura. De tal forma que esses dados serão estruturados em modelos de aprendizado de máquina a partir de ferramentas de processamento de fluxo de dados, sendo então utilizados para realizar vários treinamentos através da técnica de *Federated Learning* com Redes Neurais por múltiplas camadas em cada dispositivo IoT selecionado para executar a tarefa, feito então o treinamento do modelo obtido com base em padrões gerados a partir de amostras feitas na rede utilizando um modelo adversário contendo uma *botnet* ativa realizando ataques de negação de serviço distribuído em uma rede local para fins de estudo.

Para cumprir este objetivo, será proposta uma arquitetura a ser implementada, dividida em um modelo adversário para gerar ataques de *Flood* dentro da rede, modelo de captura e detecção para coletar o tráfego com base em regras de assinatura para então transformá-lo em um *dataset* contendo dados de fluxo. Por último, é realizada a construção de um ambiente de aprendizado federado para realizar um treinamento distribuído entre mais clientes, tendo a presença de um servidor que irá coordenar o processo. Este irá aplicar a atualização dos modelos por um algoritmo denominado de *Federated Averaging* (FedAvg) [43] visando utilizá-lo para realizar avaliações e testes de classificação de dados não identificados para determinar os seus comportamentos na rede.

1.1 Objetivo Geral

O objetivo do projeto é de construir uma arquitetura de NIDS (do inglês, *Network Intrusion Detection System*) [10] por detecção de anomalias com utilização de *Federated Learning*. A presença de aprendizado de máquina distribuído tem o intuito de aprimorar a privacidade e a segurança de redes IoT utilizando técnicas de aprendizado de máquina com Redes Neurais e *Deep Learning* em um modelo gerado por ataques reais através de um Modelo Adversário implementado em uma rede de Internet das Coisas, ou seja, com a presença de dispositivos de microarquitetura.

1.1.1 Objetivos específicos

- Construir uma arquitetura para redes IoT com detecção de intrusão por anomalias;
- Implementar um modelo adversário para gerar tráfegos maliciosos na rede através de uma *botnet*;
- Utilizar um ambiente real com o intuito de estruturar uma rede local para realizar os ataques de forma isolada e para realizar a captura do tráfego em tempo real;
- Realizar a captura de tráfegos na rede para processá-los e transformá-los em um modelo de aprendizado de máquina;
- Aplicar o treinamento do modelo gerado por *Federated Learning* em dispositivos de Internet das Coisas;
- Realizar a análise do volume de tráfego gerado pelo modelo adversário no alvo;

- Realizar a análise de desempenho dos testes e classificações provenientes do treinamento do modelo de tráfegos da rede por aprendizado federado;

1.2 Justificativa

Devido ao fato dos dispositivos IoT apresentarem uma microarquitetura simples, tal que incentiva o aumento na quantidade de ameaças a essas redes, é necessário aplicar soluções de segurança e privacidade com o intuito de não apenas detectar possíveis ataques como também aplicar formas de prevenção. Ao passar dos dias, cada vez mais tipos de *malwares*, *trojans*, novos métodos de DDoS surgem ao encontrarem novas vulnerabilidades nos sistemas, e muitas dessas vulnerabilidades são *Zero-Day Vulnerability.* Ou seja, os fabricantes não possuem conhecimento da vulnerabilidade, possibilitando então que os atacantes se aproveitem da falha para prejudicar o sistema, de tal forma que estes não são identificados.

Dessa forma, o projeto em questão surge para introduzir um sistema nessas redes IoT com o intuito de evitar com que esses ataques passem despercebidos. A arquitetura proposta aplica uma estrutura de NIDS [10] que se baseia na detecção de intrusões e ataques através do tráfego da rede, fazendo a distinção entre fluxos benignos e maliciosos a partir da classificação dos pacotes enviados e recebidos utilizando técnicas de Inteligência Artificial.

Com o objetivo de tornar o sistema escalável, eficaz e inteligente, é aplicado o método de *Federated Learning* para descentralizar a classificação dos tráfegos, diminuindo então as chances de que ocorra envenenamento de dados e roubo de informações sensíveis pelo fato dos dispositivos apresentarem um baixo nível de segurança e privacidade. Apesar dos dispositivos serem simples e apresentarem dificuldades de processamento e comunicação, o *Federated Learning* é apresentado a partir de um plano simples de treinamento, de tal forma que os dispositivos enviarão de volta ao servidor não o modelo de forma completa, e sim parâmetros e pesos de aprendizado que serão utilizados para estruturar a classificação final.

2 Revisão Bibliografica e Marco Teórico

2.1 Marco Teórico

2.1.1 Redes IoT

A Internet das Coisas surgiu como uma nova forma de tecnologia no mercado com o objetivo de transformar as formas de industrialização, automação e gerenciamento de forma inteligente nas mais diversas áreas. Pelo fato do mercado exigir cada vez mais soluções inteligentes, eficientes e com alta interoperabilidade, a integração e utilização de métodos inovadores tanto para negócios inteligentes públicos e privadas se tornaram necessários, com isso surgiu a aplicação de dispositivos e *frameworks* capazes de automatizar os processos requeridos no dia a dia. Então, a introdução de uma arquitetura IoT para as infraestruturas de redes está sendo realizada com mais frequência pelas diversas vantagens operacionais apresentadas [9], estando altamente presentes de forma funcional em redes de alguns segmentos, como por exemplo:

- Sistemas de *smart home* para automatização e gerenciamento de energia.
- Sistemas de saúde para monitoramento de condição física através de sensores.
- Agropecuária para avaliação de parâmetros temporais através de sensores.
- Fábricas para gerenciamento de produtividade e monitoramento de equipamentos.

As redes IoT possuem uma estrutura de arquitetura geral que define todas as suas funcionalidades e formas de comunicação [34], consistindo na composição de equipamentos com funções específicas como os dispositivos e *gateways*, além das 5 camadas que compõem os sistemas IoT em termos de operação e funcionamento, sendo então definidas a seguir:

1. Camada de percepção: constituída dos dispositivos de microarquitetura que formam a estrutura básica de uma rede IoT para a automatização de tarefas, podendo ser sensores, antenas, rastreadores, câmeras, dentre outros.
2. Camada de rede: transmite as informações capturada pelos dispositivos para então enviar ao processamento da informação, tal que essa transmissão pode ocorrer por *Wi-Fi*, *Bluetooth*, *ZigBee*, redes 4G/5G, etc.
3. Camada intermediária: consiste na recepção das informações transmitidas pela camada de rede para ocorrer a avaliação acerca da decisão computacional dos dados obtidos.
4. Camada de aplicação: responsável por utilizar as informações processadas e as decisões realizadas pela camada intermediária para gerenciar os dispositivos pertencentes a estrutura da rede.

5. Camada de negócios: realiza o controle de todo o sistema IoT através da constante análise e visualização das informações e estatísticas presentes em toda a rede, para então definir estratégias e planos para a administração do ambiente.

Dentro da arquitetura dos sistemas IoT, em termos de operabilidade e conectividade para realizar o funcionamento da rede a partir das camadas citadas anteriormente, a estrutura depende da comunicação ativa entre os dispositivos, serviços e aplicações. Geralmente, a estrutura básica contém aplicações IoT como administração, análise de *Big Data* [2], gerenciamento e visualização de dados e estatísticas, servidores IoT que coordenam a atividade de cada dispositivo instalado, e também de *gateways* IoT responsáveis pela recepção e encaminhamento de dados na rede como a comunicação dos dispositivos com a internet, além da agregação, controle e encapsulamento de informações em protocolos distintos daqueles presentes em uma rede IoT para possibilitar a comunicação entre os diferentes pontos do sistema.

Para abordar e administrar corretamente uma grande quantidade de dados em uma comunicação de dispositivos IoT, é utilizado normalmente nas arquiteturas de Internet das Coisas tanto o *Cloud* quanto o *Fog* e *Edge computing* [19]. São formas de transmissão e comunicação de fluxo de dados entre diferentes sistemas IoT, tal que o Edge computing, especificamente falando, torna possível o tratamento de dados para fins de visualização, integração, dentre outras funcionalidades, podendo o processamento computacional de redes externas. O *Cloud Computing* pode ser amplamente utilizado para a utilização de servidores que exigem menor custo computacional e que garantem maior eficiência e segurança quanto ao armazenamento de dados da rede, sendo também responsável pela parte analítica, de integração e gerenciamento de armazenamento.

Os sistemas IoT, por ser uma tecnologia recente e que apresenta dispositivos de arquiteturas simples, apresenta vários desafios a serem superados e problemas vigentes. Esses problemas envolvem questões de segurança e privacidade, tal que redes IoT nos últimos anos vem sofrendo bastante ataques devido às vulnerabilidades dos dispositivos e afins, gerando a necessidade de montar sistemas de detecção de intrusão e implementação de protocolos/políticas eficientes. Além da segurança e privacidade, existem desafios e problemas relacionados à interoperabilidade para ocorrer a integração correta dos sistemas com os dispositivos, assim como questões de escalabilidade de serviços e disponibilidade de equipamentos, pelo fato dos dispositivos exigirem muita memória, processamento e uso de banda, resultando então nas implementações de soluções como o *Cloud*, *Fog* e *Edge computing* [34].

2.1.2 Sistema de detecção e prevenção de intrusão

Com o surgimento e consequente desenvolvimento de sistemas inteligentes envolvendo arquiteturas de redes de comunicação, houve um aumento significativo no número de intrusões e atividades maliciosas por parte de terceiros, seja de forma interna ou externa, de tal forma que são criados *malwares* com alta capacidade destrutiva que tentam explorar as vulnerabilidades das arquiteturas. Então, por motivos de segurança, ocorreu o surgimento de sistemas de detecção e prevenção de intrusões, referenciados como *Intrusion Detection System* (IDS) e *Intrusion Prevention System* (IPS) respectivamente, abordados em [38] e [63], que consistem em um constante monitoramento

de uma rede para identificar possíveis ataques, incidentes ou eventos que possam prejudicar o sistema a ser analisado.

O IDS tem como papel verificar e identificar, por meio de coleta de tráfegos de rede, arquivos de dispositivos, dentre outros, atividades não autorizadas e maliciosas que ferem os princípios de confidencialidade, integridade e autenticidade, configurando então como uma possível intrusão. Essa análise ocorre usando conjunto de algoritmos e regras para comprovar o nível de ameaça do evento e diferenciar os fluxos positivos dos negativos, para então ocorrer um alerta para a administração da rede ficar ciente da invasão. O IPS, além da detecção, também possui a tarefa de conter e mitigar o ataque usando métodos como bloqueio de tráfego, utilização de *firewall* e encerramento da comunicação, agindo então como um IDS ativo.

O IDS pode ser classificado de diversas formas, variando conforme a sua utilização, métodos de detecção, localização, decisão, arquitetura e reação. A Figura 2.1 revela as diferentes categorias de IDS e demonstra que um IDS pode operar de várias maneiras tal que cada sistema ou arquitetura pode apresentar melhor funcionamento a depender das características de operação do sistema de detecção.

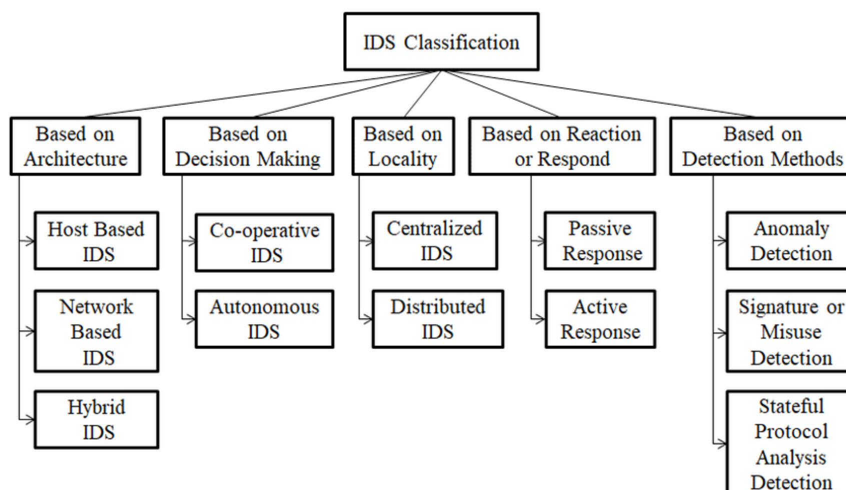

Figura 2.1 – Diagrama das classificações de um sistema de detecção de intrusão. Fonte: [57]

Uma das classificações mais importantes para a definição de um IDS ou IPS é a baseada na arquitetura, sendo um *Host Intrusion Detection System* (HIDS). Inicialmente, o HIDS aborda a detecção através de ações de *hosts* (como dispositivos) através dos seus arquivos, atividades realizadas como aplicativos e serviços, logs gerados por eventos que foram acionados, registros, auditorias do sistema operacional, dentre outras informações que são provenientes de um hospedeiro de uma rede. Ou seja, o HIDS foca em realizar o monitoramento de ações internas que não têm relação com o tráfego da rede e sim com seus hospedeiros. Então, esse IDS opera com base no comportamento do sistema para cada *host*, fazendo com que ele seja mais centralizado [59].

O NIDS [59], que será abordado neste projeto, realiza a detecção através do fluxo de dados a circular pela rede da arquitetura através de uma captura constante de pacotes, fazendo então a separação dos tráfegos benignos e maliciosos através dos padrões de comportamento apresentados. As atividades maliciosas que são mais encontradas por um NIDS são externas, de tal forma que é

possível identificar o ataque nas suas fases iniciais e evitar com que se espalhe por todo o sistema. O NIDS é implantado juntamente de um *firewall*, podendo estar na própria rede como um componente ativo ou pode ser implementado em uma rede DMZ (*Demilitarized Zone*). Então, o NIDS foca em realizar o monitoramento de fluxo de informações como pacotes, requisições, respostas, serviços, portas, comunicações de rede, dentre outros critérios. Com isso, uma das principais desvantagens de um NIDS é a presença de falsos positivos e negativos, ou seja, confundir um tráfego benigno com um malicioso, e vice e versa, sendo necessário outras técnicas para evitar esse tipo de problema, como por exemplo o *Machine Learning*.

Outro tipo de classificação importante é o de métodos de detecção sendo: detecção por assinatura ou detecção por anomalias [51]. A detecção por assinatura se baseia na utilização de um banco de dados constituído de regras e marcações de ataques já existentes para ocorrer a detecção da invasão ao reconhecer um desses ataques. Esse reconhecimento ocorre por um algoritmo, que avalia a compatibilidade da natureza do tráfego com os tipos de ataques já demarcados e padronizados, isso ocorre avaliando os serviços que são utilizados, a forma da comunicação, os cabeçalhos dos pacotes, estado da conexão, etc. Ou seja, a detecção por assinatura avalia a comunicação para checar se ocorrem semelhanças com algum tipo de ataque conhecido, tal que a principal desvantagem é a detecção de novos tipos de *malwares* como ataques do dia zero, onde não há nenhum registro do tipo de invasão e com isso a detecção é menos eficiente.

A detecção por anomalias [51], sendo um dos focos deste trabalho, se baseia no aprendizado do comportamento da rede e dos tráfegos para compreender os padrões da comunicação e identificar atividades anômalas e maliciosas. Ou seja, usando técnicas como *Machine Learning* com Redes Neurais, é possível criar um modelo de comportamento do tráfego e realizar seu treinamento para classificar a natureza dos dados que transitam na rede. Determinando os tipos de tráfegos normais e benignos, é possível avaliar quando um fluxo de dados foge do padrão e apresenta um possível comportamento malicioso, isso sendo efetuado através do treinamento e do teste do modelo de detecção utilizado pelo NIDS. Uma vantagem desse tipo de detecção é de identificar ataques do dia zero pelo fato de não precisar conhecer o ataque de antemão, e sim determinar o comportamento anômalo do tráfego. Com isso, uma desvantagem é o número de falsos positivos pelo fato de muitas vezes tráfegos benignos ocorrerem de forma diferente, fazendo com que o NIDS identifique como anômalo, porém pode-se contornar esse problema com melhorias nas técnicas de *Machine Learning*, de Redes Neurais, de avaliação estatística e de detecção.

2.1.3 Redes Neurais e Deep Learning

As Redes Neurais [27] são uma técnica avançada de *Machine Learning* (ML) que consiste na simulação das redes de neurônios do cérebro como arquitetura para realizar o aprendizado de máquina. O aprendizado utiliza dados de treinamento passados e, por meios de retroalimentação (*feedback*), aprende novos dados enquanto os utiliza novamente para dar continuidade ao processo. Este método serve também para realizar novas previsões e testes de dados, importante para utilizar o treinamento já finalizado com o objetivo de aplicar classificações em amostras desconhecidas.

A sua estrutura segue a conexão de múltiplas camadas totalmente conectadas com a presença

de nós que recebem entradas de dados e geram parâmetros de resultados para enviá-los à próxima camada, visando continuar o processamento da informação e realizar seu aprendizado. Essa estrutura é formada por camadas e nós definidas como Redes Neurais, de tal forma que cada nó é identificado como um neurônio.

Na estrutura, existem três tipos de camadas [27]: a camada de entrada, as camadas intermediárias ou escondidas e a camada de saída. A camada de entrada consiste em neurônios que receberão os dados iniciais, sejam novos dados ou dados de realimentação, para prosseguir com o treinamento nas camadas intermediárias. Essas camadas escondidas são responsáveis pelo processamento das informações de entradas, fazendo o aprendizado ao gerar resultados no formato de pesos de saída além de parâmetros chamados de viés ou *bias*. Esses pesos são passados de neurônio para neurônio até chegar na última camada que realiza o processo de ativação. A função de ativação é responsável por utilizar um neurônio para cada tipo de classificação e realizar a predição dos dados de entrada com base nos pesos que passaram por todo o processo de aprendizagem. A Figura 2.2 evidencia a estrutura básica de Redes Neurais explicada anteriormente.

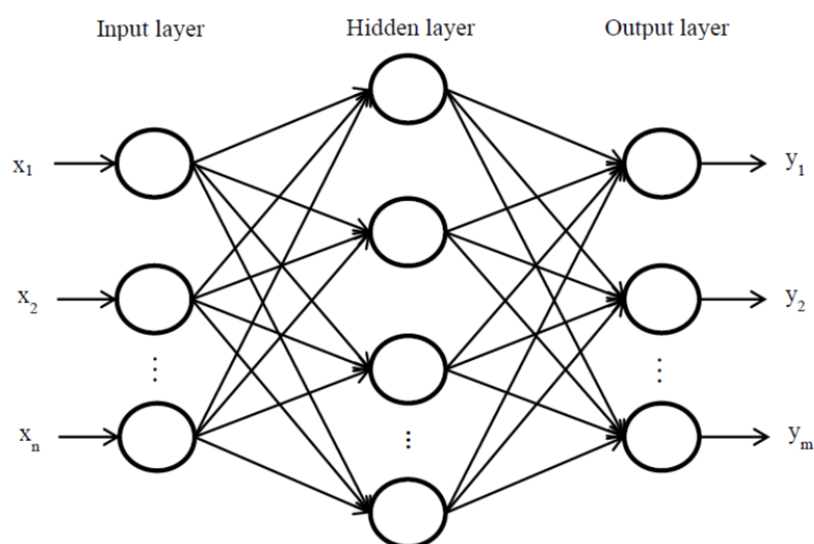

Figura 2.2 – Estrutura de Redes Neurais conectadas por camadas. Fonte: [1]

Os treinamentos por Redes Neurais utilizam diversos tipos de definições de execução para tornar eficiente e customizável o aprendizado por múltiplas camadas [32]. Um dos quesitos utilizados é o número de épocas (*Epochs*), consistindo tempo de execução para realizar o treinamento a partir de uma única direção (*feedforward*), de tal forma que quantos mais épocas melhor para a ocorrência do aprendizado. Outro quesito importante é o *Batch Size* ou tamanho do *Batch*, que consiste na quantidade de exemplos de treinamento em um percurso, seja na direção da camada de entrada para a camada de saída, ou seja, na retroalimentação de uma informação.

A partir da definição de Redes Neurais, surge o chamado *Deep Learning* ou também aprendizado profundo [44]. Esta definição consiste na utilização de Redes Neurais com várias camadas intermediárias, tendo a presença de camadas de extração de características, de tal forma que os neurônios apresentam mais conexões e conseguem realizar aprendizados mais complexos com previsões mais eficientes. O *Deep Learning* na aplicação de aprendizados não supervisionados ou

semi-supervisionados, de tal forma que consegue treinar com alta precisão modelos complexos e volumosos que exigem maior custo computacional, apresentando mais desempenho do que técnicas simples e antigas de *Machine Learning*. Ou seja, o *Deep Learning* apresenta diversas vantagens de performance e gasto computacional quanto a técnicas comuns de aprendizado [44], utilizando do conceito de Redes Neurais de forma aprofundada e mais complexa.

Algumas desvantagens do aprendizado profundo incluem a necessidade de se utilizar tecnologias de recursos mais avançados com alto poder de processamento, também tem a necessidade de modelos bem construídos para que o treinamento ocorra com boa performance e também existem situações que, com o *Deep Learning*, não é possível compreender as justificativas de uma certa rodada de treinamento. Porém, ainda sim é um método eficaz que trouxe diversos avanços na área de Inteligência Artificial e de Redes Neurais.

O *Deep Learning* apresenta diversas ramificações e arquiteturas derivadas que utilizam do método de aprendizado profundo por Redes Neurais para realizar a etapa de aprendizagem [35]. Exemplos de arquiteturas podem ser: as Redes Neurais Recorrentes (RNN) utilizado em processamento de textos e vídeos, *Generative Adversarial Networks* (GAN) que consiste na geração de dados falsos e adversários para distinguir dos verdadeiros, *Long Short-Term Memory* (LSTM) que possui conexões de *feedback* para realizar o aprendizado com o conceito de memória, e também tem o método de *Convolutional Neural Networks* ou Redes Neurais Convolucionais (CNN) que serão abordadas neste projeto.

As Redes Neurais com o *Deep Learning* são técnicas fundamentais para a Inteligência Artificial, sendo altamente usados nas áreas de reconhecimento de texto, reconhecimento de padrões como nas áreas medicinais, de segurança da informação, processamento de linguagem natural, reconhecimento e processamento de imagens, dentre outros.

2.1.3.1 Convolutional Neural Networks

As Redes Neurais Convolucionais [49] são um algoritmo de *Deep Learning* utilizado para fazer o aprendizado a partir da extração de características de uma entrada por meio de filtros, atribuir níveis de importância para diferenciar os diferentes objetos e então realizar a classificação final por uma função de ativação. É um algoritmo de classificação com pouco processamento e pode funcionar em arquiteturas com treinamento recorrente ou por *feedforward*.

Inicialmente, o CNN utiliza de camadas convolucionais com filtros (ou também chamado de kernel) para fazer as extrações das características de um dado ou imagem, de tal forma que são analisadas por funções matriciais para determinar os dados mais marcantes e realizar um mapeamento [49]. O mapeamento de características é importante para dar robustez e facilitar a administração dos dados necessários para prosseguir com o treinamento. Quanto mais filtros são utilizados, mais profunda é a etapa de extração e mais detalhes são detectados a partir das entradas.

Uma etapa importante do CNN é de realizar a subamostragem ou *Pooling* dos dados [53]. A partir das funções matriciais geradas com o mapeamento das características, ocorre uma simplificação e redução da área do mapa para realizar a sumarização das informações. Ou seja, a etapa do *Pooling* serve para escolher as principais informações das filtragens feitas pela camada

convolucional anterior [53]. Essa etapa é importante para reduzir as chances de *overfitting*, que consiste no treinamento de dados repetidos fazendo com que o modelo não seja preciso em termos de classificação. Além disso, é importante definir que o CNN utiliza vários parâmetros de configuração e filtragem para aplicar as camadas, como o *kernel* já citado anteriormente, tem o *stride* correspondente a distâncias das características no mapa matricial e também o *padding* referente a expansão da área de uma informação a ser processada pelo CNN.

Após todas as camadas de filtragem e de subamostragem, o CNN utiliza também de camadas totalmente conectadas por Redes Neurais para fazer o processamento das informações por neurônios em todas as camadas intermediárias. No fim, é utilizada uma função de ativação como o *Softmax* ou *Sigmoid* para gerar um vetor de classificação e atribuir uma categoria para aquela entrada que estava sendo treinada a partir dos seus dados filtrados pelas camadas convolucionais [49].

O CNN é comumente utilizado para treinar e classificar imagens, assim como dados em 2 dimensões. Porém, existe uma versão modificada do CNN chamada de 1D-CNN [33] que consiste em realizar o treinamento de dados sequenciais e seriais de apenas 1 dimensão, também utilizando as mesmas técnicas do CNN padrão com as camadas de filtragem e de subamostragem. O 1D-CNN será utilizado neste projeto, pelo fato do modelo a ser treinado corresponder a informações de 1 dimensão apenas. A Figura 2.2 mostra um exemplo da utilização das camadas do CNN em Redes Neurais.

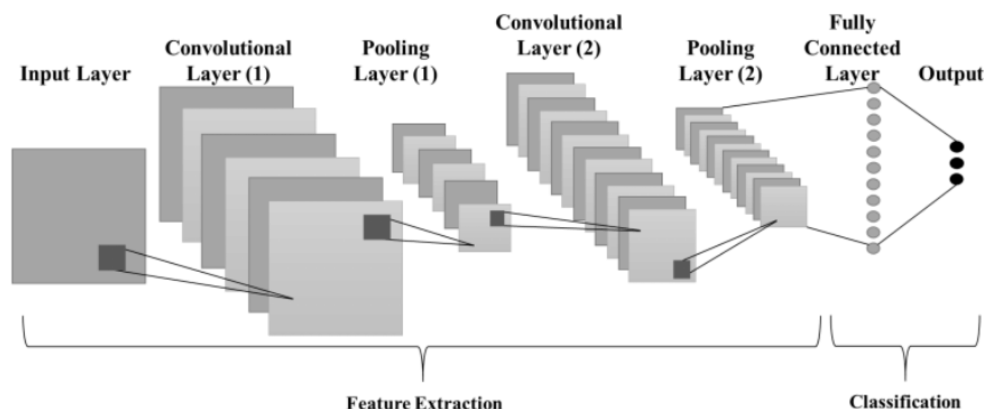

Figura 2.3 – Estrutura de Redes Neurais conectadas por camadas. Fonte: [53]

2.1.3.2 Métricas de desempenho

Dentro do aprendizado por *Machine Learning*, também com *Deep Learning* por Redes Neurais, existem diversos tipos de métricas de desempenho associadas com o processo de classificação. A classificação, podendo ser binária ou multi-categórica, tem o objetivo de decidir a classe do dado processado pelas camadas de treinamento. E para avaliar o desempenho da classificação, ou seja, se esta foi feita de forma correta, existem diversos parâmetros e métricas que servem para analisar a capacidade do modelo treinado, como: a Matriz de Confusão, Acurácia, Perda, Precisão, *Recall* e *F1-Score* [32].

A Matriz de Confusão [32] estabelece o desempenho do treinamento através de uma matriz contendo dados referentes a: Verdadeiros Positivos (TP), Falsos Positivos (FP), Verdadeiros Nega-

tivos (TN) e Falsos Negativos (FN). Esses parâmetros servem para verificar se o modelo apresentou mais resultados corretos ou errôneos, de tal forma que os valores positivos correspondem aquelas classificações que tiveram acerto, e os negativos correspondem às classificações erradas. A Figura 2.4 evidencia a estrutura básica de uma matriz de confusão 2x2.

		Actual Condition		
	Total Samples	Actual Positive	Actual Negative	
Output of Classifier	Classify Positive	TP	FP	PPV (Precision)
	Classify Negative	FN	TN	
		TPR (Recall)	TNR (Specificity)	ACC F-measure MCC

Figura 2.4 – Exemplo de Matriz de Confusão. Fonte: [61]

Um critério de alta importância para a análise de desempenho do aprendizado é a Acurácia [32]. A acurácia é a taxa correspondente ao número de acertos de valores verdadeiros feitos na classificação ao utilizar a relação dos valores verdadeiros com todos os valores presentes na matriz para realizar seu cálculo. A sua importância se deve ao fato de que toda classificação possui uma taxa de acerto e erro, então a sua presença é necessária para determinar o desempenho do modelo. A sua fórmula é:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{2.1}$$

A Precisão [32] é utilizada para avaliar a taxa de acerto de valores verdadeiros e positivos em relação a todos os valores considerados positivos. É uma métrica importante para determinar o desempenho de cada categoria presente no modelo, seja binário ou multi-categórico. A sua fórmula é:

$$Preciso = \frac{TP}{TP + FP} \tag{2.2}$$

A Revocação ou *Recall* [32], é uma métrica para avaliar a taxa de verdadeiros positivos considerando a importância dos valores negativos. Em situações que os valores negativos são de grande influência, de tal forma que a sua presença é prejudicial para a avaliação do sistema na totalidade, é utilizado o *Recall* para determinar o desempenho do modelo seguindo a quantidade de FN na matriz de confusão. Sua fórmula é:

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

Então, tem o *F1-Score* [32] responsável por determinar o balanceamento do modelo em termos de Precisão e *Recall*, de tal forma é definido como um valor de média harmônica entre essas duas métricas. Ou seja, é um parâmetro que define a relação da taxa de acertos das predições e de realizar a recuperação de exemplos da classe. Então, quanto maior o *F1-Score*, melhor é a relação entre a Precisão e o *Recall*, indicando que o modelo tem um bom balanceamento de dados e de previsões. Sua fórmula é:

$$F1Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.4}$$

2.1.4 Federated Learning

A utilização de Inteligência Artificial (IA) para sistemas inteligentes como redes que envolvem Internet das Coisas traz bastantes benefícios para a automatização e aplicação das mais diversas funcionalidades, tendo um considerável avanço nas áreas de pesquisa e na sua implementação no mercado tecnológico atual. Uma das ramificações de IA mais utilizadas no mercado é o *Machine Learning* (ML) ou também Aprendizado de Máquina [32], que aborda o treinamento inteligente de algoritmos e sistemas com o objetivo de torná-los mais eficientes a cada rodada de treino. Através de um conjunto de dados pré-determinados, podendo ser novos dados ou até mesmo resultados de treinamentos antigos, os algoritmos aplicam o treinamento para aumentar o aprendizado a certas respostas e então aumentar a precisão de predições feitas pela máquina.

Este método, podendo ser aplicado com diversas técnicas diferentes como Redes Neurais, Aprendizado por Reforço, dentre outros, realiza o treinamento de um modelo pré-estabelecido para aperfeiçoar as predições e classificações de possíveis resultados de um certo algoritmo. O ML é comumente utilizado em sistemas centralizados onde todos os modelos são treinados e armazenados em um componente central (como um servidor ou hospedeiro), tal que normalmente todos os treinamentos são feitos por um mesmo dispositivo utilizando um mesmo conjunto de dados.

Partindo do *Machine Learning*, surgiu uma técnica derivada para redes e sistemas distribuídos, chamado de *Federated Learning* (FL). O FL, assim como o ML, utiliza também técnicas de aprendizado para realizar o treinamento de um modelo ou algoritmo, porém sendo de forma distribuída entre os dispositivos da rede tal que cada um utiliza dados que estão armazenados localmente. Ou seja, cada dispositivo da rede (dispositivos finais, *gateways* ou até mesmo servidores), através das suas próprias amostras, realiza um treinamento individual do modelo sem que ocorra compartilhamento de informações entre eles [37]. Desta forma, o modelo de aprendizado é treinado individualmente por todos os componentes de maneira isolada, partindo de dados distintos, cada um contribuindo com a versão final, gerando então o denominado modelo global. Com isso, esta técnica proporciona rodadas de treinamento por aprendizado de máquina com mais privacidade e segurança de dados pelo fato de não ocorrer compartilhamento de informações, além de proporcionar acesso a dados heterogêneos.

O *Federated Learning*, podendo utilizar de técnicas como Redes Neurais e *Deep Learning* para realizar a classificação de dados, será aplicado em diversos nós que irão realizar o treinamento e gerar parâmetros ou pesos resultantes para serem utilizados no processo de formação do modelo global. Ou seja, todos os nós irão realizar seu treinamento individual e enviar os resultados para

serem juntados e agregados visando atualizar o modelo principal. Esse processo de agregação pode definir o tipo de arquitetura de *Federated Learning* aplicado na rede, diferenciados conforme a utilização do servidor e a aplicação das rodadas de treinamento. Os diferentes tipos de estrutura de FL são:

- FL Centralizado: utilização de um servidor central responsável por coordenar o processo de *Federated Learning*, fazendo a seleção dos dispositivos participantes e realizando a agregação de todos os treinamentos para atualizar o modelo global.
- FL Descentralizado: não utiliza um servidor para realizar a coordenação, tal que os clientes irão compartilhar as atualizações entre eles mesmos e definir os treinamentos.
- FL Heterogêneo: tem a presença de dispositivos com processamento e capacidade de comunicação distintos, de tal forma que os treinamentos ocorrerão em sistemas que se diferem computacionalmente.

A realização do aprendizado de forma heterogênea partindo da utilização de dados distintos por cada dispositivo seguindo às três estruturas acima pode ter certas complicações e problemas. Muitas vezes, dispositivos IoT que possuem arquitetura simples e problemas de comunicação podem ser um ponto de falha e não realizar o treinamento corretamente. O fato desses dispositivos possuir pouca capacidade de processamento, de memória e por utilizarem meios de comunicação menos potentes que trazem complicações para a transmissão de parâmetros importantes durante o treinamento ocasionam dificuldades no processo de obtenção de um modelo treinado e classificado corretamente. O processo de *Federated Learning* exige com que a rede possua uma coordenação desejável de tal forma que minimize a existência de nós falhos que possam comprometer com o treinamento, sendo importante a aplicação de um aprendizado iterativo por rodadas, técnicas avançadas para evitar contaminações no treinamento partindo de nós infectados (como a utilização de *Blockchain*), rede bem estruturada para evitar problemas de delays, dentre outros. A estrutura básica de uma rede com *Federated Learning* é demonstrada na Figura 2.5 encontrada na referência [28].

Assim como foi determinado anteriormente, a aplicação de rodadas iterativas para o processo de *Federated Learning* é fundamental para que todos os treinamentos ocorram da melhor forma possível. O round de treinamento, podendo ser coordenado por um servidor central ou pelos próprios nós participantes, consiste em algumas etapas que podem variar conforme a estrutura e a montagem da arquitetura, mas que seguem as etapas definidas, de uma maneira generalizada, a seguir:

1. Inicialização: consiste na estruturação e definição do modelo a ser treinado, tendo a escolha do método de treinamento para a formação do plano de *Federated Learning.*
2. Seleção: esta etapa consiste na seleção dos dispositivos que irão participar do treinamento, determinado pela disponibilidade dos nós e se estes estão no estado desejável para a realização do treinamento. Após serem selecionados, ocorre o envio do modelo a ser treinado por meio de um plano de FL.

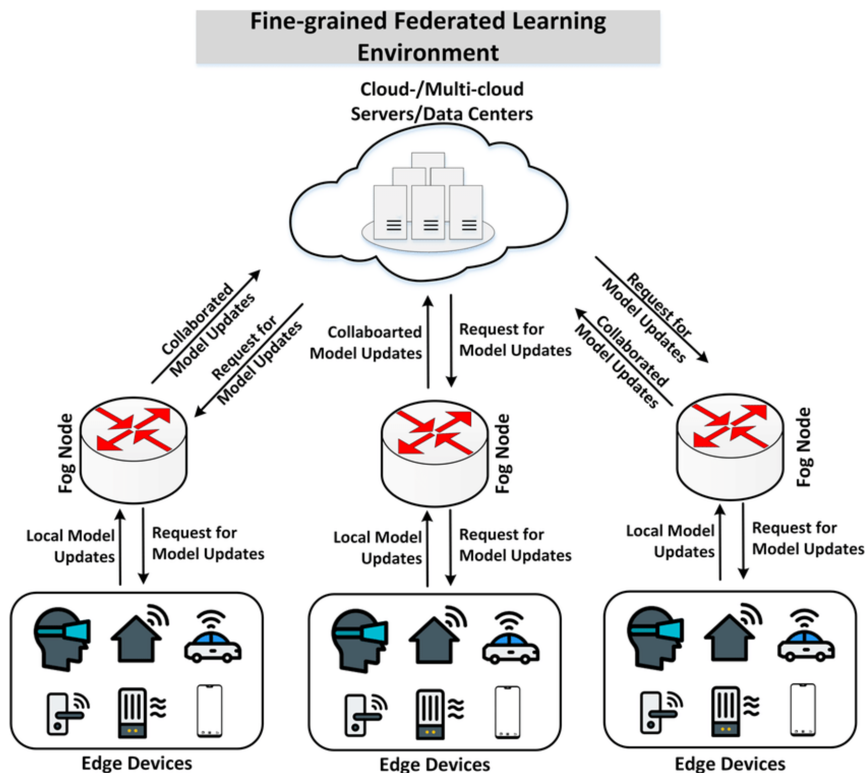

Figura 2.5 – Arquitetura básica com *Federated Learning*. Fonte: [28]

3. Treinamento: após a recepção dos modelos, cada dispositivo, seguindo seus dados localmente armazenados, aplica o treinamento conforme instruído pelo plano de FL (determinado pelo agente central) que pode ser feito por diferentes técnicas, como Redes Neurais, por exemplo.

4. Agregação ou Atualização: quando os dispositivos finalizam o treinamento, estes enviam os parâmetros resultantes para o agente responsável por agregá-los seguindo um algoritmo específico para então atualizar o modelo global. A atualização pode ser feita de forma síncrona, quando todos os dispositivos enviam os parâmetros de forma coordenada, ou de forma assíncrona, onde cada dispositivo realiza o envio apenas quando finaliza o seu treinamento individual.

Um problema do processo de treinamento do *Federated Learning* que muita das vezes ocorre se deve ao fato dos dados locais de cada dispositivo não serem independentes e identicamente distribuídos. Isso ocorre quando o domínio dos dados, em termos de marcações, são distintos e com distribuições estatísticas muito discrepantes. Isso ocasiona perda de acurácia e precisão no treinamento, pelo fato da atualização do modelo global depender de treinos realizados sobre condições não relacionadas, podendo gerar falsos positivos e negativos. A forma mais eficaz de reduzir esse problema é realizando a normalização de dados, de tal forma que ainda diferirão, porém, estarão em um mesmo domínio de marcação e estruturação, fazendo com que a perda de acurácia seja bem menor.

O treinamento de *Federated Learning*, explicado anteriormente, utiliza métodos estatísticos e de aprendizado para realizar o treino do modelo alvo. O método a ser utilizado neste trabalho é o *Deep*

Learning ou Aprendizagem Profunda, sendo uma sub-área do ML consistindo no aprendizado por redes neurais totalmente conectadas por múltiplas camadas que irão se dividir para realizar todo o processamento das informações, fazendo convoluções, amostragens e classificações conforme a rede vai se aprimorando e melhorando o treinamento. Ou seja, pela utilização dados prévios, é uma rede que irá aplicar o processo de aprendizado e classificação de modelos de forma profunda com várias camadas. Este método é aplicado em diversas áreas computacionais como reconhecimento de voz, classificação de imagens, processamento de linguagem neural, e pode ser amplamente usado na área de segurança para realizar o treinamento de IDS e IPS visando aprender e classificar os diferentes tráfegos e determinar os tipos de comunicações [54], assim como vai ser utilizado no projeto em questão

Assim como já foi falado, além do processo de treinamento distribuído utilizando técnicas como redes neurais por *Deep Learning*, e de toda a etapa de preparação do modelo, envio e recepção, uma etapa fundamental é a parte da atualização ou agregação. Sendo por meio de um servidor central ou usando os próprios nós participantes, para que o modelo inicial tenha suas classificações realizadas e seu aprendizado completo, é necessário coletar todos os resultados dos treinamentos realizados por cada dispositivo. Um dos caminhos mais comuns para tal feito é utilizando técnicas de gradiente descendente, ou seja, realizando a transformação dos resultados em gradientes que serão recebidos por um agente a qual aplicará o algoritmo de agregação fazendo uma junção ou média com base em todos os treinamentos realizados (número de clientes, número de amostras, etc). Com isso, o modelo tem seu aprendizado concretizado através de todos os participantes. O principal algoritmo por trás desse processo é o *Federated Averaging* [43], o qual utiliza os gradientes como pesos de atualização para realizar a média e atualizar o modelo global. O modelo atualizado pode então ser utilizado para os fins planejados ou até mesmo ser aplicado em uma nova rodada de treinamento.

Portanto, a técnica de *Federated Learning* é uma opção viável para manter a privacidade de toda a rede, principalmente para os dispositivos, não só pela falta de compartilhamento de informações de dados locais, como também a manutenção dos treinamentos por rodadas e também pelo fato de serem transmitidos apenas parâmetros ou pesos de atualização ao invés de informações concretas e comprometedoras. Mas, mesmo tendo um aprimoramento de segurança em relação ao *Machine Learning*, ainda há problemas no caso de ataques de reconstrução e vazamento de informações, pois mesmo que sejam trafegados parâmetros de atualização, ainda é possível obter informações secretas e é por isso que é viável que se utilize outras técnicas de segurança como, por exemplo, *Blockchain* no trabalho [48] com método de consenso e técnicas de *Differential Privacy* como no trabalho [60]. O FL é bastante utilizado em sistemas de automação, Indústrias 4.0 e inclusive sistemas de detecção e prevenção de intrusão como este trabalho, devido a sua fácil integração com técnicas de redes neurais e pelas vantagens em questões de segurança.

2.2 Revisão Bibliográfica.

O advento da Internet das Coisas (IoT), abrangendo a presença de dispositivos de micro-arquitetura, trouxe diversas pesquisas na área de *Machine Learning* para o treinamento de modelos

focados em áreas de segurança e privacidade, como detecção a intrusões e a tráfegos maliciosos. Devido ao fato das arquiteturas em redes IoT serem mais simples, apresentam maior custo computacional e de comunicação, além de apresentarem mais vulnerabilidades ao realizar treinamento por ML Centralizado. Então, é utilizado o *Federated Learning* para descentralizar os treinamentos e para aumentar o nível de privacidade ao evitar o compartilhamento direto dos dados entre dispositivos.

Para realizar o treinamento via *Federated Learning* (FL), é normalmente utilizado bibliotecas de Python como TensorFlow e PyTorch, aplicando redes neurais como *Deep Learning*. Os autores no FLEAM [36], aplicando mitigação de ataques DDoS durante o percurso em redes *Fog* e *Edge computing*, utilizou rede neural recorrente com GRU para treinar e classificar os tráfegos em benignos ou maliciosos. O VFL [25], projeto que se baseia na prevenção a ataques de inferência e forja utilizando interpolação de Lagrange e *blinding technology* para redes IIoT (*Industrial Internet of Things*), e o projeto [13], que aplica arquitetura de detecção de intrusão PHEC com tolerância a ruído, utilizam predições por redes neurais via *Multi-layer Perceptron* (MLP). Os autores em [41] utilizam as Redes Neurais Convolucionais (CNN) para realizar a redução de parâmetros e mantendo preditores contribuintes para melhores classificações de intrusão. O projeto [64] se baseia na utilização do LSTM (*Long Short-Term Memory*) para retenção de parâmetros fundamentais do FL para detecção de anomalias de comandos de texto via *shell*.

No *Federated Learning*, os servidores centrais são responsáveis por obter os parâmetros provenientes dos treinamentos realizados pelos dispositivos e/ou *Security Gateways*, realizando a agregação para atualização do modelo global. Para evitar problemas de servidor malicioso, vazamento de dados e possíveis reconstruções por collusion attacks, métodos de agregação segura, encriptação como o Secure Multi-party Computation (SMC), aplicação de ruídos randômicos por *Differential Privacy* (DP) ou até mesmo Secure Socket Layer são utilizados para aprimorar a privacidade dos dispositivos nas redes IoT.

Os autores do projeto [65] focam em aplicar o DP com FL por CNN nos sistemas IoT por *Fog Computing*, além de encriptação homomórfica na agregação contra os ataques de reconstrução no servidor afetado. O projeto IFed [29] aplica o *Local Differential Privacy*, sendo um mecanismo avançado e específico de DP para a ofuscação de dados provenientes de sistemas de energia elétrica, configurando um *Power IoT*. O projeto [11] aplica o FedAvg e o SMC para realizar agregação segura, realizando a encriptação de cada atualização e parâmetros de memória interna em sistema de *Federated Learning* com dispositivos Android. Os autores no projeto [3] chamado de IOTFLA implementam um sistema com FL para *smart home* utilizando os algoritmos SecureDAV e ECIPAP, que são Secure Data Aggregators para tornar a agregação mais segura por meio de encriptação. Outro projeto que aplica encriptação é o [8], utilizando NIZKP-HC, sendo uma variação de encriptação homomórfica, para criptografar os parâmetros, aumentando a acurácia e diminuindo o custo computacional.

O *Federated Learning* será aplicado para treinar modelos de arquiteturas que se baseiam na detecção de intrusões e possivelmente aplicando mitigação de ataques, chamados de IDS e IPS respectivamente, podendo ser executados tanto nos dispositivos ou nos *Security Gateways*. Um

tipo de IDS bastante utilizado em sistemas IoT é a detecção de anomalias, consistindo na distinção entre tráfegos benignos e maliciosos através de *datasets* pré-processados. Os autores do projeto [62] implementam FL para detecção de anomalias em *endpoints* como Raspberry Pi e em CPU/GPU para simulação, aplicando *Deep Autoencoder* e FedDetect no treinamento. O projeto [45] utiliza o sistema de detecção de anomalias com *Deep Learning* por LSTM e GRU a partir de pacotes capturados em formato PCAP para detectar ataques como SYN DDoS, Ping DDoS, *Man in the Middle*, etc. O DIoT [46] foi um dos primeiros projetos de *Federated Learning* para detecção de anomalias a partir da detecção por tipos específicos de dispositivos (device-type-specific detection models), realizando testes reais através do *malware* Mirai.

Outro projeto de detecção de intrusão [47], focando em ataques por envenenamento, aplica o IDS nos *Security Gateway* separando o servidor que realiza a agregação com os dispositivos que fazem os treinos locais. Os autores do projeto [40] utilizaram sistemas com *Edge Computing* em redes IoT para aplicar detecção e privacidade quanto a *Phishing* por imagens com spam embutido, utilizando CNN como rede neural e SMC para agregação segura. Uma implementação inteligente e fundamental hoje em dia para sistemas de *Federated Learning* é a *Blockchain*. Com a *Blockchain*, a privacidade e confidencialidade dos modelos globais e parâmetros compartilhados é aumentada, garantindo principalmente a rastreabilidade (traceability) pelo registro das transações. Os autores do [39] utilizam da *Blockchain* com *Miners* para determinar a autenticidade do servidor central visando evitar atividades maliciosas no modelo global, aplicando armazenamento do modelo por *Merkle Patricia Tree*.

Este projeto em questão, assim como os demais trabalhos relacionados, envolve a aplicação de treinamento por *Federated Learning* para fins de segurança, especificamente para realizar a detecção de intrusão em sistemas IoT com base em anomalias. Em comparação aos trabalhos [36], [25], [13], [41] e [64], que utilizam *Federated Learning* com rede neural por GRU, PHEC, LSTM e CNN, o projeto em questão utilizará 1D-CNN e Redes Neurais por MLP como mecanismo de *Deep Learning* no procedimento de FL por dispositivos IoT como um Raspberry Pi a partir de um novo modelo de treinamento que no caso é gerado através de uma captura de tráfego realizado pelo módulo de IDS. Esse treinamento será coordenado por um servidor responsável por elaborar e coordenar cada etapa de parametrização, treinamento, avaliação e agregação.

Quanto ao sistema de detecção a intrusão, em comparação com os trabalhos [62], [45], [46], [47] e [40] que abordam o IDS com anomalias, porém com a identificação por *Deep Autoencoder*, análise de pacotes em PCAP por LSTM/GRU, avaliação de *Phishing* em spam de imagens e de envenenamento de dados. O foco do projeto é de aplicar um NIDS com detecção por anomalias, avaliando os padrões dos tráfegos capturados no componente após um ataque pela *botnet* Mirai assim como no trabalho [46]. Porém, a detecção será feita em um modelo processado por uma solução com Apache Kafka e Apache Spark, através de captura de rede por Suricata. Esse modelo será treinado por FL assim como apresentado anteriormente para ocorrer as classificações de cada tráfego com base em sua natureza: benigno ou malicioso, podendo ter identificação até mesmo do tipo de ataque realizado.

Além disso, em comparação com os trabalhos [39], [65], [29], [11], [3] e [8] que utilizam *Block-*

chain, *Differential Privacy* com encriptação homomórfica, SMC e até mesmo NIZKP-HC, este projeto utilizará de uma comunicação criptografada por SSL (*Secure Socket Layer*) através da geração de certificados por pares de chaves públicas e privadas com o intuito de aumentar a segurança dos treinamentos. Para no fim utilizar o *Federated Averaging*, assim como o trabalho [11], para agregar todos os treinamentos e então atualizar o modelo final para ser utilizado no IDS com o objetivo de detectar possíveis intrusões.

Os autores do trabalho [31] fizeram a implementação de um HIDS (*Host-based Intrusion Detection System*) de forma distribuída para avaliar a detecção de intrusão no *backbone* de uma rede IoT, focando os esforços nos componentes intermediários de uma rede isolada com a presença de uma controladora. A análise das vulnerabilidades ocorre a partir da aplicação de protocolos de monitoramento como o Syslog e SNMP (*Simple Network Management Protocol*). Em comparação, este projeto em questão tem o objetivo de estruturar uma NIDS (*Network Intrusion Detection System*) para avaliar o tráfego desde o *backbone* até os próprios dispositivos IoT, tendo a análise por meio de aprendizado de máquina após a devida captura dos dados da rede.

O paper [20] aborda a implementação de um IPS com uma *honeynet* em redes SDN (*Software-defined networking*) para identificar e mitigar ataques de DoS (*Denial of Service*) a partir da utilização de *OpenFlow* para encaminhar os pacotes suspeitos e tratá-los como uma possível ameaça. Este projeto, em comparação com o paper, aborda a implementação de uma IDS sem a aplicação de ações preventivas, tendo o foco em capturar o tráfego em uma rede IoT para identificar possíveis ameaças por ML distribuído, de forma que não apenas ataques de DoS serão detectados como também atividades recorrentes de *malwares*, de *botnets* e de comunicações periódicas entre os dispositivos infectados com o servidor de comando e controle da *botnet*.

No trabalho [24], é feita a implementação de uma arquitetura de NIDS utilizando os tráfegos da rede para realizar um aprendizado centralizado por 1D-CNN e LSTM, com a captura por pfSense e Suricata, tendo a geração de um modelo de treinamento posteriormente. O projeto deste artigo também implementa um NIDS, porém com aprendizado federado (distribuído entre vários clientes) a partir de uma aplicação de Redes Neurais, de tal forma que a captura dos dados da rede são feitas apenas por Suricata com a utilização de espelhamento de porta no *switch* da infraestrutura local. Os autores do artigo [26] projetaram uma IPS em uma rede de Internet das Coisas utilizando o software Snort para monitorar a rede e realizar a mitigação dos ataques próximos aos dispositivos a partir de uma rede SDN, isolando-os da rede principal. Em comparação, este projeto aborda um IDS sem a utilização de redes SDN com a captura por Suricata para gerar os alertas de assinatura em formato JSON e utilizá-los em um modelo de treinamento por aprendizado de máquina após o processamento dos dados através de uma aplicação de transformação e filtragem.

3 Arquitetura Proposta.

Este trabalho tem o objetivo de apresentar, estruturar e propor uma arquitetura de NIDS (*Network Intrusion Detection System*), ou seja, um sistema de detecção de intrusão a partir dos dados obtidos pela rede como pacotes e tráfegos, aplicado em uma infraestrutura de Internet das Coisas (IoT) contendo dispositivos de microarquitetura. O projeto consiste em identificar atividades maliciosas e possíveis invasões a partir da detecção de anomalias dos tráfegos que circulam pela rede, utilizando métodos de aprendizado de máquina, como o *Federated Learning*, para realizar as classificações de cada pacote capturado. Também será feito o aprimoramento da segurança e da privacidade de todo o processo com a utilização de um túnel SSL integrado com o processo de treinamento.

A arquitetura é iniciada com um componente de detecção, constituindo o núcleo do IDS, que utilizará de dispositivos e *frameworks* capazes de coletar cada pacote que transita entre os demais participantes da rede local, de tal forma que esses dados serão utilizados para estruturar um modelo de treinamento por aprendizado de máquina para ocorrer a classificação de cada tráfego (sendo classificados em benignos ou anômalos) e realizar a detecção de anomalias na rede. A captura será possibilitada por um *switch* com espelhamento de porta ativado, dando a capacidade de se monitorar toda a rede a partir da visualização do tráfego em todas as interfaces do dispositivo comutador.

Esses tráfegos serão gerados a partir de um modelo adversário que irá atacar um alvo a partir de um ataque de DoS (*Denial of Service*), constituído de um *malware* capaz de infectar dispositivos IoT ao explorar vulnerabilidades relacionadas às suas arquiteturas, sendo este chamado de Mirai *botnet*. Com isso, o componente de detecção tem a função de identificar esse tráfego, coletá-lo, estruturá-lo e enviá-lo para o componente de treinamento que irá utilizá-lo para classificar cada dado a partir de sua natureza de comunicação. O modelo de detecção será implementado em uma única máquina contendo as seguintes ferramentas: Suricata [56], Apache Kafka [4] e Apache Spark [7]. A utilização dessas ferramentas tem como base o trabalho [24] referente a uma implementação de arquitetura de IDS com *Deep Learning*.

Após a formação e o tratamento do modelo de aprendizado com os tráfegos previamente capturados, iniciar-se-á o processo de *Federated Learning*. Primeiramente, o modelo será enviado a um servidor em uma máquina virtual que será responsável por coordenar todo o procedimento. O aprendizado federado será realizado através de um *framework* próprio denominado de *Flower* [23]. Este *framework* suporta o treinamento por *TensorFlow*, *Keras* e *PyTorch*, sendo altamente customizável e possibilita a integração de dispositivos IoT. Então, os clientes do aprendizado possuirão a estrutura do *framework* para realizar o treinamento através da linguagem antes definida, de tal forma que o servidor irá administrar toda a parte da avaliação e da agregação do modelo por meio de várias rodadas diferentes.

Tendo a estrutura de treinamento definida, o servidor iniciará o processo ao selecionar os

dispositivos da rede que irão participar da etapa de treino. A partir do servidor, o plano de treinamento será enviado para cada cliente, tal que esses irão utilizá-lo de forma individual a partir de seus dados locais e irão enviar os resultados de volta ao servidor na forma de pesos e parâmetros. O servidor então irá receber essas atualizações e juntará todas elas utilizando um algoritmo de agregação já definida no *framework* para então formar o modelo final contendo as classificações desejadas a partir da contribuição de todos os dispositivos participantes. Com isso, o modelo final pode ser utilizado para realizar a detecção de novos dados ainda não classificados com o intuito de identificar o comportamento de cada evento ainda desconhecido.

Portanto, é proposta uma arquitetura de NIDS que tem o objetivo de detectar anomalias na rede utilizando métodos como o *Federated Learning* com segurança por SSL. Na arquitetura, estarão presentes uma máquina alvo contendo um servidor *Web*, um modelo adversário representado por uma *botnet* sendo utilizada para realizar o tráfego malicioso, um componente de detecção que irá capturar os dados de rede, um servidor responsável por coordenar o processo de treinamento e dispositivos IoT que participarão do aprendizado. Neste tópico, serão abordadas cada estrutura da arquitetura de forma separada, tal que estes são os componentes que serão desenvolvidos posteriormente: o Modelo Adversário, Módulo de Detecção e Implementação do *Federated Learning*.

A representação geral do modelo lógico do sistema proposto, seguindo as definições apresentadas anteriormente, está presente na Figura 3.1.

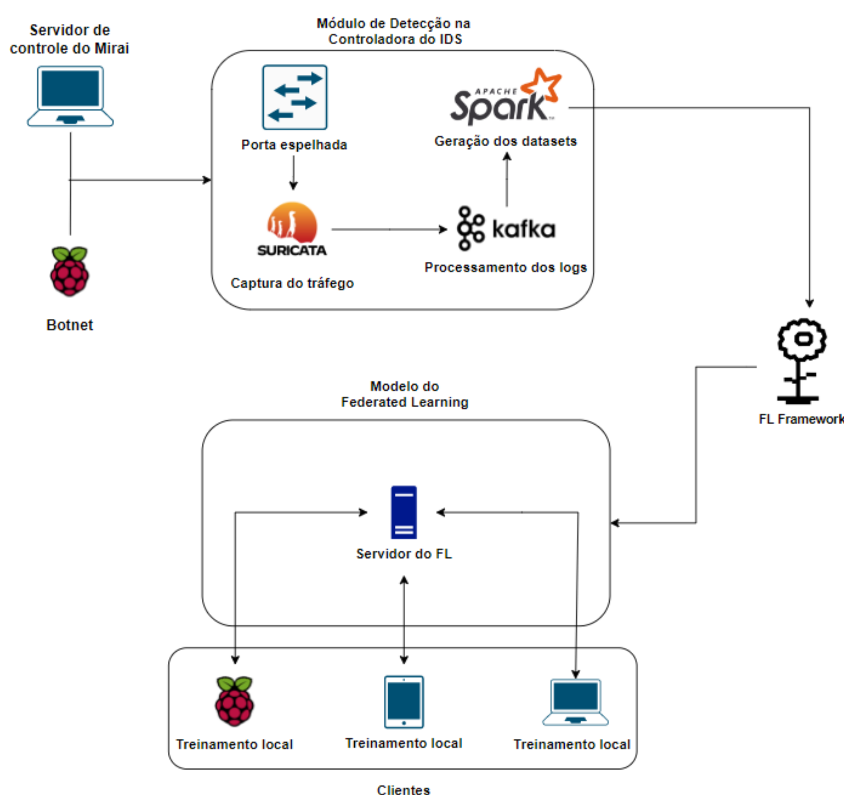

Figura 3.1 – Arquitetura proposta

Para sintetizar todos os passos do funcionamento da arquitetura proposta, é representada a estrutura lógica de forma procedural a partir de um fluxograma de raias na Figura 3.2. Todas as

etapas da arquitetura, com a descrição do funcionamento de cada componente definido no modelo lógico, são descritas de forma simplificada a seguir:

1. Primeiramente, é definido um Modelo Adversário contendo uma *botnet* para realizar ataques dentro de uma rede local. Esses ataques são feitos em um alvo de forma distribuída, ou seja, a partir de vários dispositivos IoT distintos que serão infectados antes da etapa de testes.

2. O dispositivo comutador da rede local, possuindo tecnologia de Espelhamento de Porta, irá enviar todos os tráfegos na rede (de todas as interfaces presentes) para um dispositivo específico, chamado de Controladora do IDS.

3. A controladora do IDS irá receber todo o tráfego da rede, incluindo os ataques feitos pela *botnet* durante a fase de testes, e irá capturá-los a partir de um software chamado de Suricata.

4. Após a captura ativa do tráfego, será utilizado um software chamado Apache Kafka para receber através de um *cluster* a captura e enviá-los como um fluxo de mensagens com destino a outra aplicação.

5. O fluxo de mensagens é enviado para um software chamado de Apache Spark, que irá processar todos os eventos e irá transformá-los em um modelo de aprendizado de máquina (*dataset*) utilizando a linguagem de banco de dados SQL.

6. Com o *dataset* finalmente estruturado, ocorre o ajuste do modelo através da montagem do padrão de classificação de cada evento utilizando as regras de captura do Suricata como base. Ou seja, são definidas todas as possíveis classificações de pacote que serão utilizadas no processo de detecção por anomalias.

7. Então, o *dataset* de ML referente ao tráfego capturado a ser utilizado é enviado a um ambiente de *Federated Learning* a partir de um *framework* chamado de Flower para incluí-lo no processo de aprendizado de máquina distribuído.

8. A partir da utilização de um servidor e de diversos clientes diferentes, é feita a etapa de treinamento do modelo de aprendizado obtido pela detecção da Controladora do IDS para realizar a identificação do comportamento de cada evento.

9. Ao fim do treinamento por FL, são feitas as análises de performance e desempenho da detecção de um conjunto de testes com base nas categorias de comportamento pré-selecionadas na etapa 6.

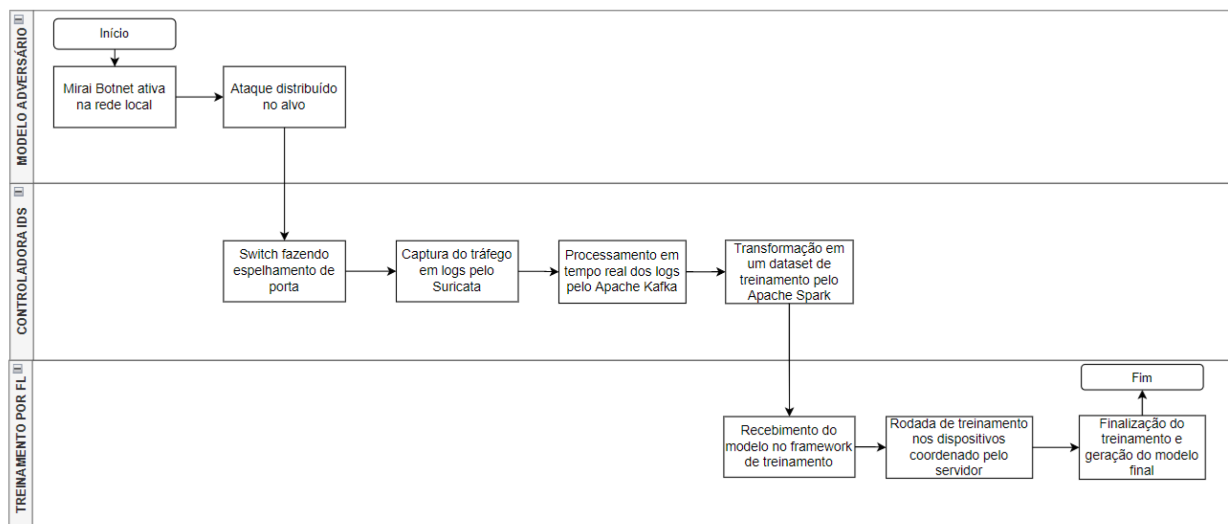

Figura 3.2 – Fluxograma da arquitetura

3.1 Modelo Adversário

3.1.1 Mirai Botnet

Para realizar a implementação do projeto de IDS, primeiramente deve-se ter um modelo adversário que será utilizado para fins de experimentação, análise e estudos. O modelo adversário tem a função de aplicar os tráfegos maliciosos na rede através de ataques e realizar infecções nos dispositivos da arquitetura para comprometê-los, causando distúrbios em toda a estrutura. Neste projeto, sendo uma rede IoT contendo equipamentos de microarquitetura com várias vulnerabilidades de sistema, o principal objetivo do modelo adversário será de infectar os dispositivos com um *malware* para roubar suas atividades e utilizá-los em uma *botnet* com o intuito de realizar ataques de negação de serviço (DoS) a outras redes e infraestruturas. Por diversos tipos de contaminações como instalações de binários, roubo de credenciais, injeção de aplicações, exploração de portas abertas, dentre outros, é possível utilizar um dispositivo como um *bot*, e por comandos por acesso remoto, fazê-lo aplicar ataques a um determinado alvo sem que este tenha controle de suas ações.

Então, a partir de um modelo contendo a utilização de um *malware* para realizar infecções e ataques, serão propagados dados e pacotes maliciosos na rede a ser estudada para introduzi-los no modelo de detecção pela IDS a ser apresentada, com o objetivo de coletar amostras de ataque a serem analisadas por *Machine Learning* e fazer a sua detecção ativa. Ou seja, o modelo adversário estará presente no projeto para servir de amostras a arquitetura, de tal forma que terão dispositivos vulneráveis na rede, e através das tentativas de infecção e ataque, os dados gerados serão capturados e utilizados no modelo de treinamento por *Federated Learning* para fins de identificação e detecção das anomalias na rede. Na arquitetura apresentada, o *malware* que será utilizado pelo modelo adversário é o Mirai através de uma controladora que possui um servidor de controle e ataque.

O Mirai, assim como descrito em [30] e no trabalho [42], é um *malware* que foi desenvolvido e primeiramente utilizado por volta de 2016, o qual verifica a presença de dispositivos IoT com processadores ARC (Linux, BSD, etc, que estão embutidos no sistema) vulneráveis na rede,

como dispositivos de fumaça, DVRs, câmeras, roteadores, Raspberry Pi, para então infectá-los e transformá-los em máquinas zumbis de uma *botnet* com o intuito de realizar possíveis ataques de DDoS (*Distributed Denial of Service*) a um alvo específico. O *malware* procura problemas de credenciais em dispositivos não atualizados, normalmente com as senhas padrões do fabricante, para então infectá-los carregando um payload malicioso no seu sistema e controlá-los remotamente com o objetivo de realizar ataques a outras redes. O Mirai utiliza *botnets* normalmente por servidor CNC (Servidor de Comando e Controle) o qual controla os dispositivos infectados remotamente por comandos em uma interface administrativa. Os ataques normalmente são de negação de serviço como DDoS por DNS, UDP, TCP, ACK, SYN *Flood*, dentre outros [30].

A estrutura do Mirai consiste dos seguintes segmentos: usuários, *botmaster*, servidor CNC, *bots*, ataques e *Loader*. Os usuários são aqueles que, por meio de uma API do Mirai, se conectam com o servidor CNC para realizar requisições de serviços como a aplicação de ataques ou visualização do número de clientes (dispositivos infectados) pertencentes à *botnet*. O *botmaster*, através da interface de admin, consegue gerenciar todo o servidor CNC para não só comandar o processo de infecção de dispositivos IoT, como também enviar comandos aos *bots* para realizarem os ataques disponíveis (estão presentes nos códigos do Mirai) assim como o processo de recrutamento de outros alvos para a *botnet* a partir daqueles que já estão infectados. Os *bots*, que possuem o binário de infecção injetado no sistema, são acessados remotamente pela porta 23 ou 101 e respondem aos comandos do servidor. O recrutamento de mais dispositivos ocorre com a presença do *Loader*, capaz de identificar dispositivos vulneráveis e realizar a infecção através de *wget* ou TFTP (*Trivial File Transfer Protocol*) do binário capaz de tornar o acesso remoto do dispositivo possível pela central de comando após a quebra do usuário e senha. Com isso, assim como é mostrado na Figura 3.3, é possível ver a estrutura básica do Mirai como foi descrita anteriormente.

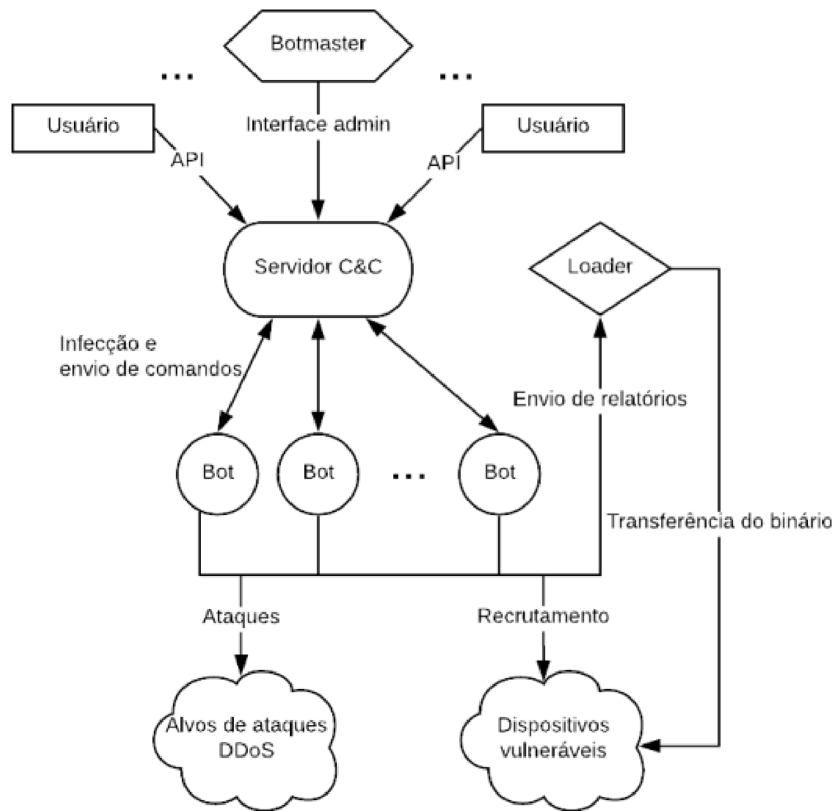

Figura 3.3 – Estrutura lógica do Mirai. Fonte: [52]

O Mirai possui diversos módulos e componentes no seu código que tornam possível a realização e aplicação da estrutura apresentada anteriormente. No seu código, existem os componentes do servidor CNC onde tem todos os componentes responsáveis pelo levantamento do servidor, implementação da interface de admin, gerenciamento do controle, aplicação da comunicação da porta 23 e 101 com a API do Mirai e a definição do banco de dados, que é utilizado para guardar os *logins* e senhas dos usuários do Mirai. Também é possível encontrar o código *attack-go* onde está presente todos os tipos de ataques relacionados a DDoS executáveis pela central de comando do Mirai. Além disso, o Mirai possui módulos como o *Killer*, sendo um código responsável por verificar a presença de outros *malwares* que possam estar sendo executados no alvo, além de outros serviços em portas, que são utilizadas pelo Mirai. Ou seja, antes do Mirai ser introduzido no dispositivo alvo, ocorre uma procura por *malwares* rivais e serviços em execução que atrapalharão a sua atuação no sistema caso não sejam finalizados.

Outro módulo aplicado é o *Scanner*, responsável pela procura de novos dispositivos a partir do *Telnet* para encontrar alvos vulneráveis. Através de uma geração aleatória de IPs (públicos ou privados), ocorre o teste de conexão na porta 23 (*Telnet*) para encontrar um dispositivo IoT que apresenta a vulnerabilidade de acesso remoto. Quando é encontrado esse dispositivo, os endereços IPs identificados juntamente dos usuários e senhas são enviados para um servidor nomeado *Scan-Listen* na porta 48101 com o objetivo de registrar o alvo para ser infectado consequentemente. Então, para ocorrer a infecção do Mirai no dispositivo, é aplicado o componente *Listen* progra-

mado na linguagem C, que introduz um binário de infecção através da porta 23 no alvo para este se juntar a *botnet* controlada pelo servidor CNC e consequentemente ser utilizado para realizar ataques de DDoS. Todos esses módulos e códigos referentes ao Mirai estão presentes nos trabalhos [30] e [42].

3.1.2 Implementação do ambiente de testes

Para realizar a implementação do IDS com uma Mirai *botnet* para aplicar os ataques desejados e então capturá-los ativamente, foi montado um ambiente de testes em uma rede isolada contendo todos os componentes necessários para aplicar e validar a arquitetura proposta anteriormente. Nesta infraestrutura estão presentes: um *switch* modelo Cisco Catalyst 2950 com espelhamento de porta ativo conectado a todos os outros dispositivos diretamente na rede 164.72.15.0/24, uma máquina Ubuntu chamada de Controladora do IDS com um servidor DNS em execução, uma máquina com um servidor Web em execução chamada de Alvo, uma máquina CentOS chamada de Controladora do Mirai contendo o servidor de controle da infecção e por último os Raspberry Pi infectados que fazem parte da *botnet*. A Figura 3.4 evidencia a estrutura citada, implementada no laboratório. Todos os componentes com suas funções dentro da topologia serão explicados a seguir.

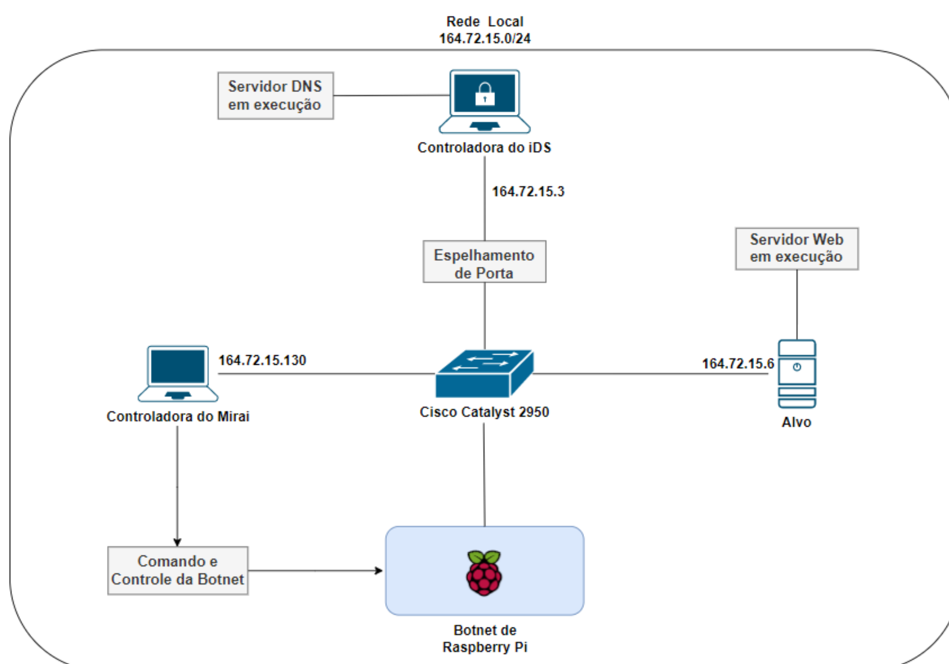

Figura 3.4 – Estrutura implementada no laboratório.

Para estabelecer a conexão entre todos os componentes, foi instalado um *switch* como um *backbone* para toda a rede se comunicar localmente sem ter nenhum tipo de conexão com a rede pública. Neste *switch*, foi configurado o espelhamento de porta de tal forma que a interface da Controladora do IDS recebe os dados referentes ao tráfego de todas as outras interfaces ativas, sendo altamente necessário para o processo de detecção, justificando a sua escolha para o projeto em questão. Foi configurada a rede de endereço 164.72.15.0/24 e sua presença é fundamental para

realizar a troca de informações entre cada um dos equipamentos.

A controladora do IDS é uma máquina com o Ubuntu 16.04.6 instalado e com endereço IPv4 164.72.15.3/24. A sua interface contém o espelhamento de todas as outras interfaces do *switch*, pelo fato de ser responsável por monitorar, analisar e avaliar o tráfego de toda a rede. A sua participação no projeto é crucial para realizar o processo de detecção, sendo o componente que irá agir como a parte da defesa contra o modelo adversário. Para fins de teste e pelo fato de ser importante para o funcionamento correto da *botnet*, foi levantado um servidor DNS (*Domain Name Server*) para resolver os domínios da rede. O Mirai, durante seu processo de escaneamento e infecção dos dispositivos identificados, necessita da atuação de um servidor de DNS para que os alvos possam resolver o nome do servidor de Comando e Controle. Por isso foi configurado um servidor BIND, assim como mostra a Figura 3.5, sendo também utilizado para realizar ataques de UDP e DNS *Flood* na porta 53 da controladora do IDS durante a captura.

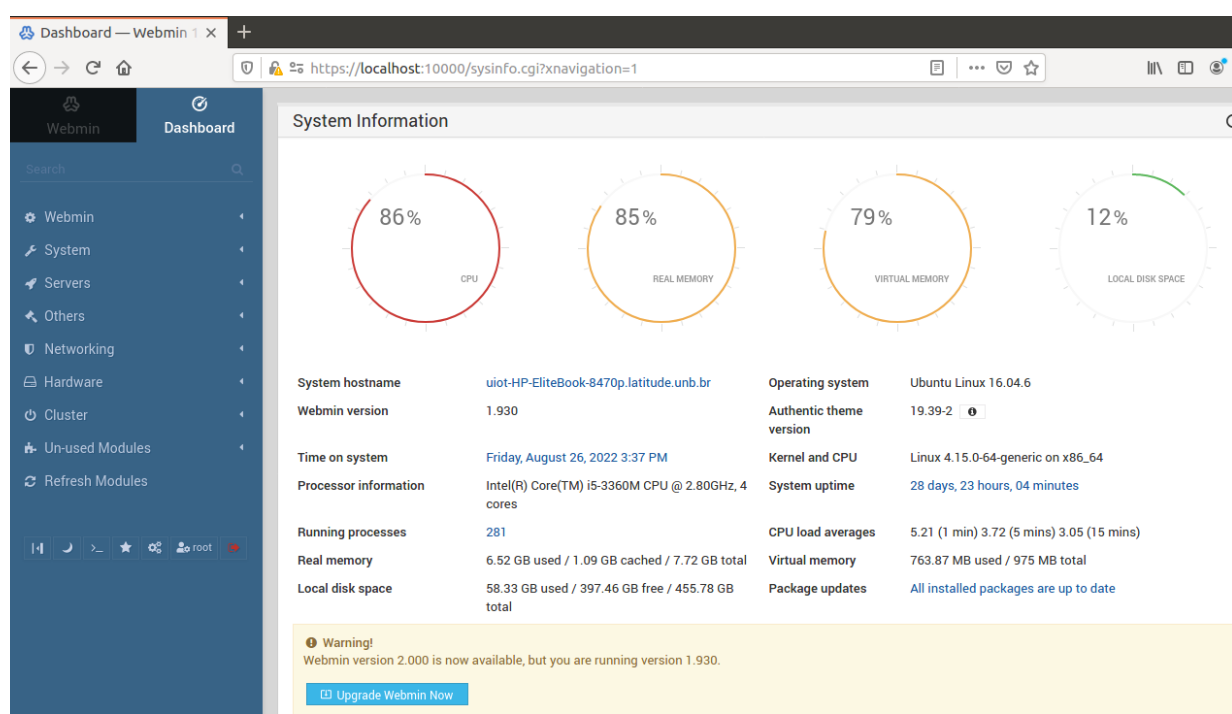

Figura 3.5 – DNS Server na controladora do IDS.

A máquina alvo, rodando o sistema operacional Linux, está configurada pelo IP 164.72.15.6/24 e ela possui um servidor *Web* em *Apache HTTP Server* sendo executada. A sua finalidade na arquitetura é para servir de ponto de ataques de DoS a partir da *botnet*, de tal forma que o objetivo é realizar ataques de UDP e SYN *Flood* na porta 80 conseguindo gerar um grande tráfego no serviço *Web* e introduzir uma abundante quantidade de requisições. De forma que o processamento de cada pacote aumente e consequentemente afete o desempenho do servidor. Ou seja, essa máquina é o ponto de destino de todos os ataques que serão comandados na rede, de tal forma que o servidor será monitorado e analisado com o intuito de se avaliar a quantidade de tráfego que os dispositivos infectados geram no alvo. A Figura 3.6 apresenta a página em HTML que foi preparada para o experimento.

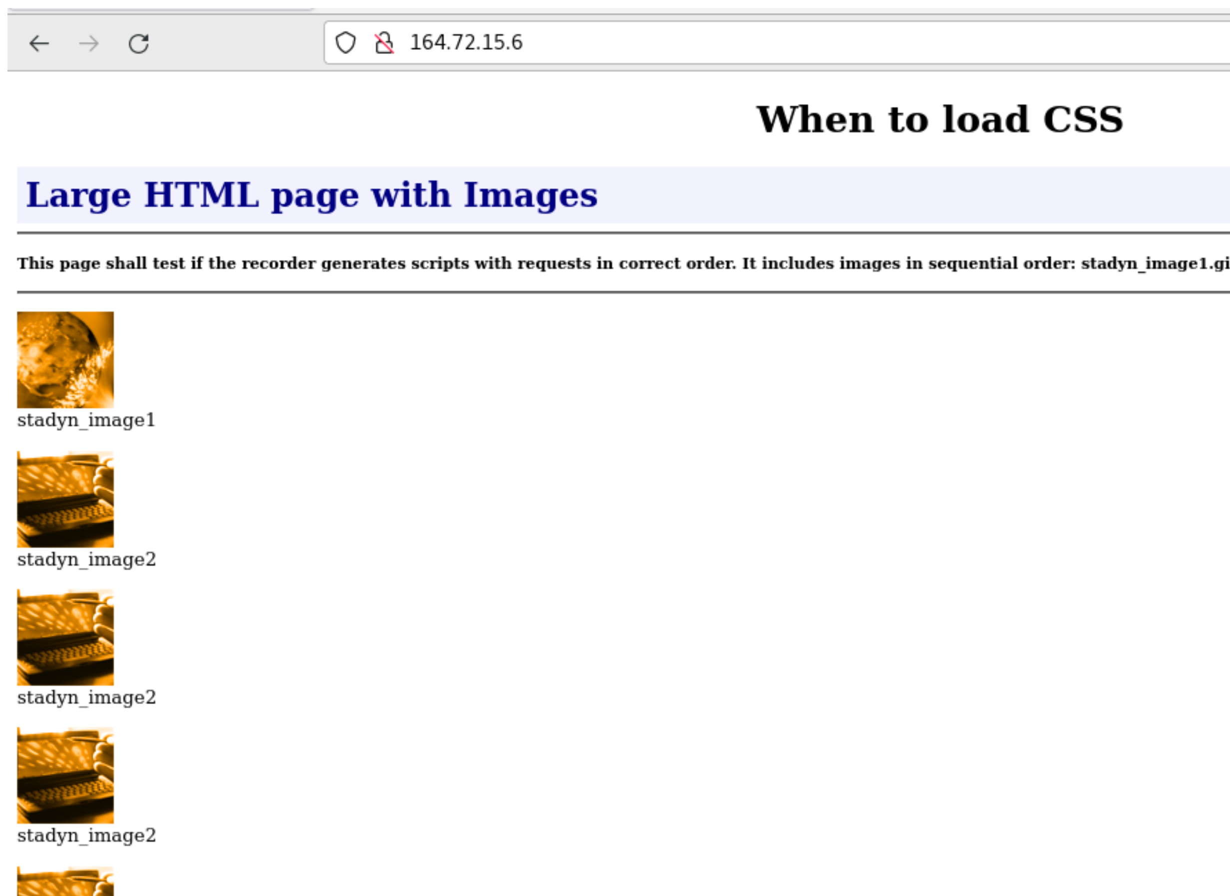

Figura 3.6 – *Web Server* levantado no alvo.

A Controladora do Mirai, sendo uma máquina executando CentOs e com endereço IP igual a 164.72.15.130/24, é o componente principal do modelo de adversário por ser responsável pelo processo de infecção, gerenciamento e administração dos ataques através do Mirai. Nele foi preparado e configurado o *malware* Mirai seguindo uma distribuição para fins educacionais nomeada de *Cosmic* [15], sendo uma modificação do código original do Mirai contendo diferentes ataques e comportamentos. A instalação do Mirai consiste na junção de todos os códigos e estruturas que já foram abordados, tendo o levantamento dos seguintes componentes: um servidor *Web* em *Apache HTTP Server*, um servidor de Comando e Controle e um servidor de *ScanListen*.

O servidor *Web* tem a finalidade de disponibilizar todos os arquivos de infecção executados nos dispositivos. Esses arquivos são binários seguindo a distribuição *sora* que é própria do Cosmic, de tal forma que estão disponíveis em vários formatos diferentes a depender de cada arquitetura que o Mirai consegue infectar. O principal binário que será utilizado no projeto é o *sora* seguindo o padrão de arquitetura de um Raspberry Pi 3 Modelo B. Esse binário será instalado em cada dispositivo e executado posteriormente para ocorrer a infecção. Ou seja, a página *Web* serve como um repositório para todos os arquivos de infecção do Mirai, e essa lista pode ser visualizada na Figura 3.7.

164.72.15.130/bins/

Index of /bins

Name	Last modified	Size	Description
Parent Directory		-	
sora.arm	2022-08-21 22:24	24K	
sora.arm5	2022-08-21 22:24	21K	
sora.arm6	2022-08-21 22:24	29K	
sora.arm7	2022-08-21 22:24	48K	
sora.m68k	2022-08-21 22:24	52K	
sora.mips	2022-08-21 22:24	26K	
sora.mpsl	2022-08-21 22:24	27K	
sora.ppc	2022-08-21 22:24	24K	
sora.sh4	2022-08-21 22:24	50K	
sora.spc	2022-08-21 22:24	60K	
sora.x86	2022-08-21 22:24	24K	

Figura 3.7 – Página *Web* com os binários.

O servidor de Comando e Controle (CNC) é a principal ferramenta do Mirai, sendo sua função gerenciar e controlar todos os dispositivos que estão na rede. Com a sua presença, é possível observar quantos *bots* adentraram a *botnet*, de tal forma que a comunicação entre a controladora e os *bots* ocorre de forma periódica, tendo uma verificação constante da atividade desses dispositivos infectados, e também é possível visualizar quais são os métodos de ataques disponíveis assim como aplicá-los em um alvo específico. Ao realizar a execução do servidor, é possível visualizar todos os dispositivos em que estão conectados na rede, quais são seus endereços e suas portas de comunicação, de tal forma que é necessário acessar o terminal por meio de *Telnet* no *localhost* e na porta 1312.

No terminal, ao digitar *methods*, são apresentados todos os ataques disponíveis, assim como uma instrução básica de como realizá-los. No comando de ataque, devem ser inseridos o IP de destino, o tempo de execução e a porta de destino. Com a inserção do comando, a controladora ativa cada um dos *bots* pela porta 23 com o objetivo deles realizarem o *Flood* escolhido. Então, quando o tempo de execução finalizar, os ataques também serão encerrados. A Figura 3.8 evidencia a inicialização do servidor com todos os dispositivos conectados e a Figura 3.9 mostra o terminal com todos os métodos de ataque (com as devidas instruções) e com o número de *bots* conectados na parte superior do console.

```
Cosmic started!

[Cosmic] Connected | IP Address: 164.72.15.11:55664 ()
[Cosmic] Connected | IP Address: 164.72.15.11:55666 ()
[Cosmic] Connected | IP Address: 164.72.15.11:55668 ()
[Cosmic] Connected | IP Address: 164.72.15.11:55670 ()
[Cosmic] Connected | IP Address: 164.72.15.12:54190 ()
[Cosmic] Connected | IP Address: 164.72.15.24:47012 ()
[Cosmic] Connected | IP Address: 164.72.15.24:47014 ()
[Cosmic] Connected | IP Address: 164.72.15.24:47016 ()
[Cosmic] Connected | IP Address: 164.72.15.24:47018 ()
[Cosmic] Connected | IP Address: 164.72.15.5:47950 ()
[Cosmic] Connected | IP Address: 164.72.15.5:47954 ()
[Cosmic] Connected | IP Address: 164.72.15.5:47952 ()
[Cosmic] Connected | IP Address: 164.72.15.5:47956 ()
[Cosmic] Connected | IP Address: 164.72.15.12:54192 ()
[Cosmic] Connected | IP Address: 164.72.15.12:54194 ()
```

Figura 3.8 – Mirai sendo executado.

```
Cosmic Net > Bots: [10] | User: [mirai] | Methods:[9] | Bypass:[2]
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
Zelensky is a clown

[mirai@Cosmic]
    methods

Usage: ![METHOD] [IP] [TIME] dport[PORT]
 .udp [IP] [TIME] dport=[PORT]
 .cosmic [IP] [TIME] dport=[PORT]
 .std [IP] [TIME] dport=[PORT]
 .xmas [IP] [TIME] dport=[PORT]
 .ovh [IP] [TIME] dport=[PORT]
 .udphex [IP] [TIME] dport=[PORT]
 .dns [IP] [TIME] dport=[PORT]
 .syn [IP] [TIME] dport=[PORT]
 .nfo [IP] [TIME] dport=[PORT]

[mirai@Cosmic]
```

Figura 3.9 – Servidor CNC na controladora do Mirai.

Na última etapa da montagem do Mirai, é levantado o servidor de *ScanListen* responsável por escutar os dispositivos em que estão infectados para ocorrer a inserção de suas credenciais em um banco de dados na linguagem SQL. Portanto, o ScanListen age como uma ponte entre o *bot* recém-infectado e o servidor de comando, para ocorrer a sua identificação e controle posterior pela controladora. A Figura 3.10 apresenta a execução em conjunto do servidor CNC e do *ScanListen* no terminal da controladora.

```
[root@cnc ~]# screen ./cnc
[detached from 17572.pts-0.cnc]
[root@cnc ~]# cd loader/
[root@cnc loader]# screen ./scanListen
[detached from 17615.pts-0.cnc]
[root@cnc loader]#
```

Figura 3.10 – Execução do Mirai e ScanListen.

Por último, são conectados os dispositivos do tipo Raspberry Pi 3 Modelo B que compõem a *botnet* do projeto. Eles também estão configurados na rede 164.72.15.0/24 cada um com um IP definido, de tal forma que eles também conversam com todos os outros dispositivos. Para realizar a infecção dos mesmos, todo o processo foi feito de forma manual ao invés de realizar o processo automático de detecção e inserção do *malware* por questões de praticidade e gerenciamento do *malware* em cada dispositivo. O processo é realizado a partir da página Web da controladora, de tal forma que foi instalado em cada dispositivo o binário no formato *arm7* pelo comando: *wget http://164.72.15.130/bins/sora.arm7*.

Por conseguinte, é feita a execução do arquivo no terminal do Raspberry Pi que foi acessado por *Telnet* e então este é conectado com o servidor de Comando e Controle confirmando a infecção. O processo de comunicação para realizar a ponte entre os dois é feita através de uma porta aleatória definida pelo próprio Mirai, porém, a forma de realização do comando de ataque do Mirai nos *bots* ocorre pela porta 23. A Figura 3.11 mostra a execução do binário e a Figura 3.12 apresenta a execução de um comando de lista de portas em escuta ou estabelecidas, evidenciando a comunicação entre o *bot* e a controladora do Mirai. O processo de montagem segue a estrutura definida nos trabalhos [30] e [42].

```
root@raspberrypi-5:~# ls
sora.arm7
root@raspberrypi-5:~# ./sora.arm7
Connected To CNC
root@raspberrypi-5:~#
```

Figura 3.11 – Binário sendo executado em um Raspberry Pi.

```
root@raspberrypi-5:~# netstat -ant
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:45263           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:50009           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:48667           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:45789           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:57661           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:56703           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:60449           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:59745           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:36929           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44485           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:37957           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44709           0.0.0.0:*               LISTEN
tcp        0      0 164.72.15.5:47240       164.72.15.130:1312      ESTABLISHED
tcp        0    136 164.72.15.5:23          164.72.15.3:48806       ESTABLISHED
tcp6       0      0 :::22                   :::*                    LISTEN
root@raspberrypi-5:~#
```

Figura 3.12 – Conexões do dispositivo infectado.

Então, é formado o ambiente de testes contendo o dispositivo de detecção e o modelo adversário que foi previamente definido. Todo o processo de atuação do modelo adversário pelo Mirai desde o surgimento da controladora até a realização dos ataques está descrita por um fluxograma de raias na Figura 3.13, e o fluxograma de toda a comunicação entre a *botnet* e a central de comando na arquitetura está representada na Figura 3.14.

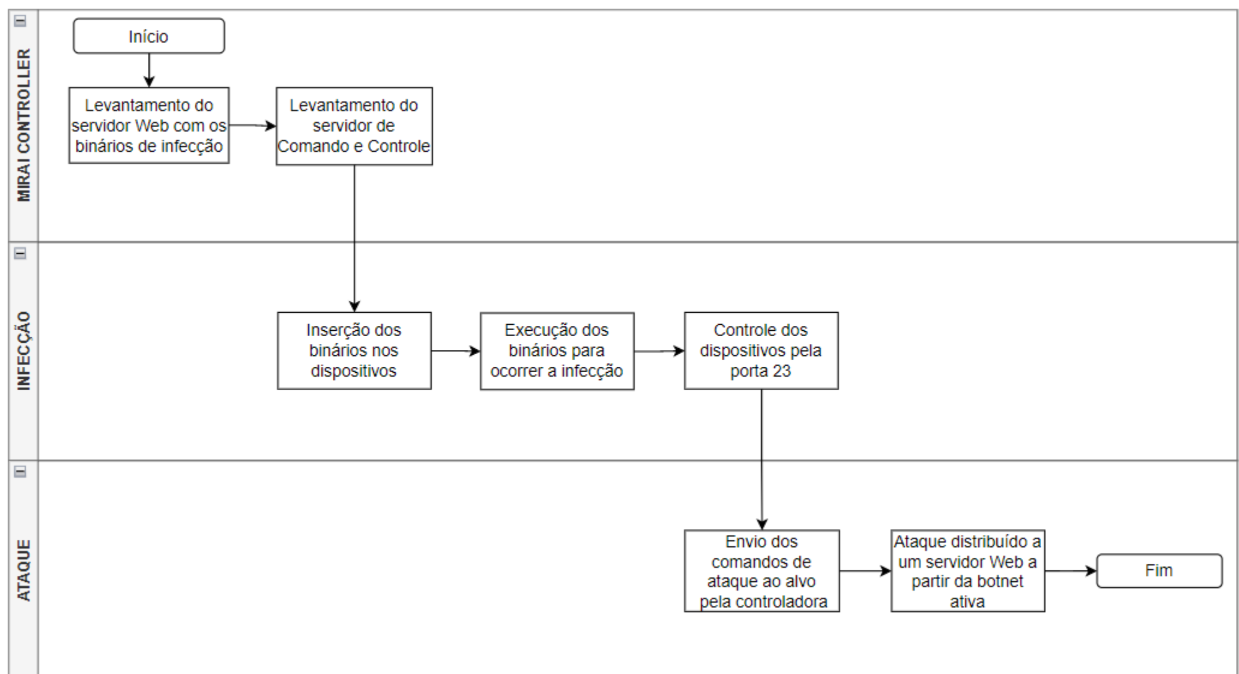

Figura 3.13 – Fluxograma do Mirai

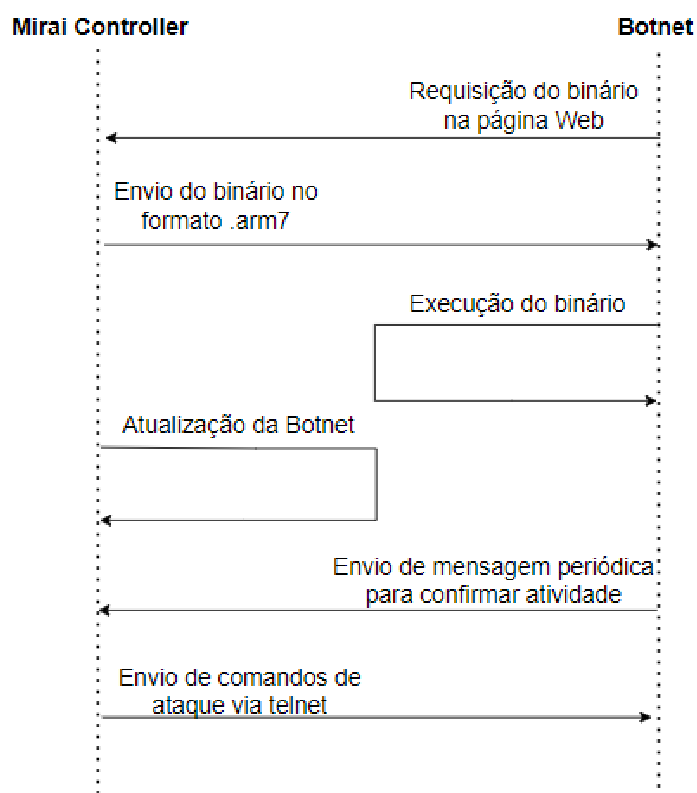

Figura 3.14 – Fluxograma da comunicação do Mirai com a *botnet*

3.2 Módulo de Detecção

3.2.1 Estrutura básica

A partir da proposta de arquitetura definida anteriormente, a etapa inicial da construção de um IDS consiste na forma de obtenção dos dados que serão analisados para identificar possíveis intrusões e ataques, especialmente em uma rede de Internet das Coisas que apresenta diversas vulnerabilidades. Com o objetivo de realizar a separação e a identificação dos tráfegos benignos e maliciosos, sendo estes gerados a partir do modelo adversário contendo uma *botnet* de dispositivos IoT infectados, é estruturado um modelo de detecção na rede local a ser estudada utilizando uma controladora de IDS com espelhamento de porta para observar toda a rede, possuindo então os componentes necessários para realizar as seguintes tarefas:

- Capturar o tráfego advindo do modelo adversário na porta espelhada da controladora de IDS a partir de regras de assinatura.
- Realizar a injeção da captura em uma aplicação de fluxo de mensagens que irá produzir os dados em um *cluster* seguindo um formato específico.
- Realizar a transformação da captura a partir do processamento em tempo real dos dados em uma aplicação de *streaming* estruturada.

- Estruturar um modelo de treinamento de aprendizado de máquina a partir do processamento dos dados.

- Enviar o modelo de treinamento finalizado para o servidor responsável pela implementação do *Federated Learning*.

Então, o processo de detecção consiste em um dispositivo central que receberá os tráfegos que serão identificados por um *software* contendo regras de assinatura de ataques e invasões. Os alertas gerados pelo *software* serão consumidos por um sistema de fluxo de mensagens para enviá-los a uma aplicação responsável por realizar o processamento em tempo real de cada mensagem com o intuito de estruturá-los no formato de um modelo de treinamento. Este modelo será utilizado posteriormente em um *framework* de treinamento distribuído por *Federated Learning* com o objetivo de classificar cada tráfego em benigno ou anômalo. Sendo assim, a estrutura formada pode ser classificada como um NIDS com detecção de anomalias.

Inicialmente, na rede local IoT a ser estudada, terá uma máquina com Ubuntu sendo responsável por gerenciar todo o processo de detecção e geração do modelo de treinamento, denominado de controladora do IDS. A controladora, através do *switch* responsável pelo funcionamento da rede local, possui espelhamento de porta (Port Mirror) de tal forma que os tráfegos que estão circulando por todas as outras portas do *switch* são observados e capturados pela controladora. Com isso, será possível obter as informações necessárias para identificar a atuação do Mirai e também de outros ataques ou *malwares*.

Seguindo como referência o artigo [24], será utilizado o *software* Suricata IDS/IPS, com funcionamento descrito em [21], a partir das suas bibliotecas de regras para capturar os alertas e dados de rede em formato de JSON. Contendo os *logs* referentes ao tráfego, será utilizado o Apache Kafka, também utilizado em [50], sendo um sistema de mensagens e eventos que funciona por *data streaming*, para receber os JSONs referentes ao Suricata. O recebimento dos JSONs pelo Kafka permite que estes sejam consumidos pelo Apache Spark com Structured Streaming [55] (através de uma integração entre o Kafka e o Spark) com o intuito de gerar um *DataFrame/Dataset* a partir das chaves e valores dos JSONs utilizando um programa feito em *Scala*, estruturando então um modelo de *Machine Learning* a ser utilizado por um servidor de FL.

A Figura 3.15 contém um fluxograma apresentando o módulo de detecção com todas as etapas definidas e ferramentas a serem utilizadas no processo. A implementação de cada material e a explicação de cada etapa do processo serão apresentadas com mais detalhes nos subtópicos: Suricata, Apache Kafka, Apache Spark com *Structured Streaming* e Modelo de Treinamento.

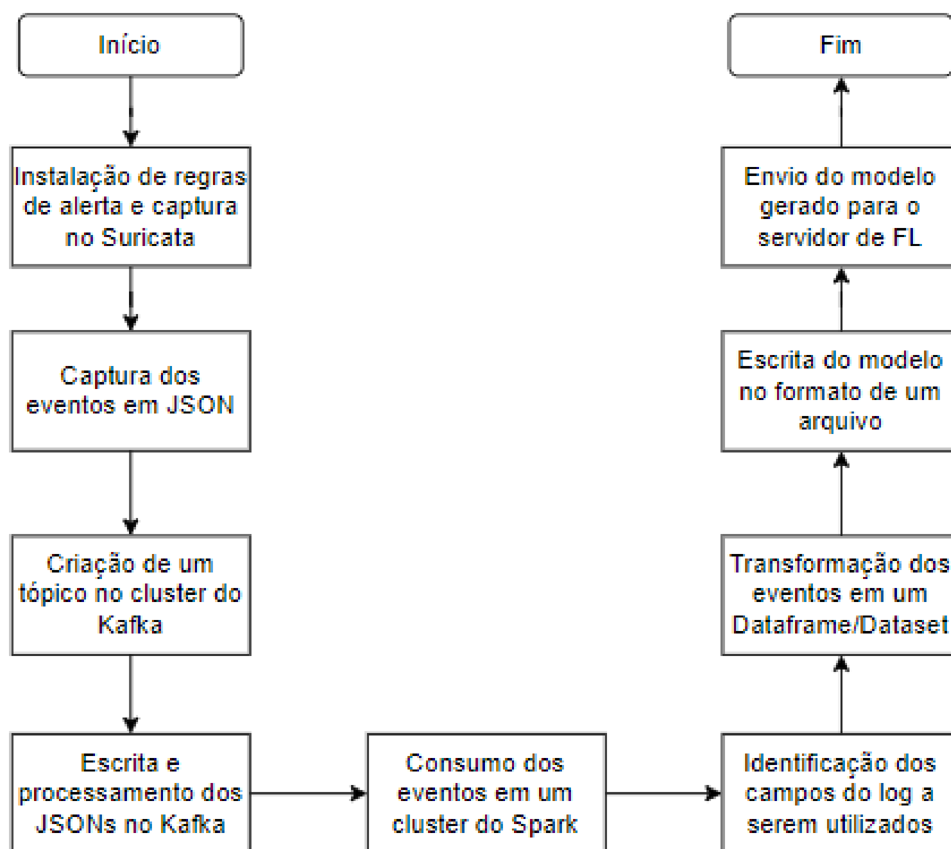

Figura 3.15 – Fluxograma do Módulo de Detecção

3.2.2 Suricata

O Suricata [21] é um *software Open Source* capaz de analisar e detectar ameaças em uma rede local a partir de um conjunto de regras pré-estabelecidas no seu funcionamento, agindo então como um suporte a IDS e IPS no monitoramento de tráfegos. Além disso, o Suricata realiza a inspeção profunda de pacotes com *multi-threading* de um alto volume de dados, possui integração com diversas aplicações diferentes, realiza a transação de protocolos e consegue realizar a extração de arquivos. Possui integração com o *Lua Scripting* que consegue estruturar ou gerenciar formas de capturas diferentes e mais complexas de forma mais customizável. Portanto, o Suricata é um ótimo *software* para ser utilizado neste projeto devido a suas possibilidades de integração e utilidade, sendo viável para montar um *dataset* de ML e FL assim como é evidenciado no artigo [24]. A escolha desse *software* de IDS é justificado pela sua ampla utilização em trabalhos relacionados como citado anteriormente, sendo facilmente integrável com ferramentas de consumo e processamento de dados em tempo real, trazendo vantagens e benefícios para o projeto em questão em termos de captura de eventos benignos e possivelmente maliciosos em uma infraestrutura de Internet das Coisas.

A sua presença na controladora ocorre de forma que o Suricata coleta todos os dados de rede ao monitorar ativamente a interface conectada no *switch* (destino do espelhamento de porta), assim como explicado no tópico anterior. No seu processo de instalação, ocorre a definição e o gerenciamento de todas as regras utilizadas durante o monitoramento, sendo regras relacionadas a

diversos tipos de ataques e atividades registradas que apresentam viabilidade na detecção. Essas regras são configuradas durante a instalação do *software* e pela aplicação chamada de *suricata-update* responsável por atualizar o programa, tendo conjuntos de regras públicas e privadas, assim como o *Emerging Threats* [17], sendo então customizáveis trazendo a possibilidade de configurações específicas e customizáveis a depender da estrutura da rede e do objetivo da captura.

A estruturação das regras é um passo importante para ocorrer a geração dos alertas necessários e para ter a realização de classificações aproximadas do que foi detectado, de tal forma que são criados diversos tipos de *logs* referentes ao tráfego monitorado na interface a qual o *software* está em execução. No projeto em questão, todas as regras de acesso gratuito foram habilitadas assim como demonstra a Figura 3.16, contendo regras básicas de rede e segurança, indo de requisições DNS até identificação de *botnets* e *trojans*. A importância e justificativa da utilização dessas regras se deve ao fato de que é necessário ter amostras tanto de tráfegos comuns e como também de tráfegos maliciosos, para que na etapa do aprendizado de máquina as classificações por AI ocorram de forma precisa.

A Figura 3.17 evidencia a lista geral de regras instaladas na controladora, estruturadas no formato definido pelo Suricata, contendo o tipo de evento, protocolo utilizado, sentido do fluxo (rede externa para rede interna ou vice e versa), além da mensagem de classificação da regra definindo então qual foi o ocorrido ao capturar o pacote que gerou o alerta. É possível criar as próprias regras locais a depender da necessidade do ambiente, de tal forma que a escrita das mesmas segue o mesmo padrão mostrado na Figura 3.17.

```
root@uiot-HP-EliteBook-8470p:~# suricata-update list-enabled-sources
26/8/2022 -- 15:39:58 - <Info> -- Using data-directory /usr/local/var/lib/surica
ta.
26/8/2022 -- 15:39:58 - <Info> -- Using Suricata configuration /usr/local/etc/su
ricata/suricata.yaml
26/8/2022 -- 15:39:58 - <Info> -- Using /usr/local/share/suricata/rules for Suri
cata provided rules.
26/8/2022 -- 15:39:58 - <Info> -- Found Suricata version 6.0.6 at /usr/local/bin
/suricata.
Enabled sources:
  - oisf/trafficid
  - sslbl/ja3-fingerprints
  - malsilo/win-malware
  - et/open
  - sslbl/ssl-fp-blacklist
  - tgreen/hunting
  - etnetera/aggressive
root@uiot-HP-EliteBook-8470p:~#
```

Figura 3.16 – Lista de sources habilitadas no Suricata.

```
  GNU nano 2.5.3            Arquivo: suricata.rules

alert http $EXTERNAL_NET any -> $HTTP_SERVERS any (msg:"ET WEB_SPECIFIC_APPS Ga$
alert dns $HOME_NET any -> any any (msg:"ET INFO DYNAMIC_DNS Query to a Suspici$
alert tcp [98.63.230.136,99.113.188.39,99.120.173.245,99.122.201.244,99.149.215$
alert http any any -> [$HOME_NET,$HTTP_SERVERS] any (msg:"ET EXPLOIT TerraMaste$
alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"ET EXPLOIT Metasploit Vario$
alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"ET WEB_CLIENT PDF With eval$
alert tls $EXTERNAL_NET any -> $HOME_NET any (msg:"SSLBL: Malicious SSL certifi$
alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"ET MALWARE Win32/Vidar Vari$
alert http $EXTERNAL_NET any -> $HTTP_SERVERS any (msg:"ET WEB_SPECIFIC_APPS Wo$
alert tcp $EXTERNAL_NET any -> $HOME_NET 5900:5920 (msg:"ET SCAN Potential VNC $
alert http any any -> $EXTERNAL_NET any (msg:"MalSilo MALWARE (formbook) Detect$
alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ET MALWARE ProxyBox -ProxyBo$
alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"ET MALWARE [PTsecurity] W32$
alert tls $EXTERNAL_NET any -> $HOME_NET any (msg:"SSLBL: Malicious SSL certifi$
alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"ET WEB_SERVER Possible SQL $
alert tls $EXTERNAL_NET any -> $HOME_NET any (msg:"SSLBL: Malicious SSL certifi$
alert tls $EXTERNAL_NET 443 -> $HOME_NET any (msg:"ET MALWARE ABUSE.CH SSL Blac$
alert tcp $HOME_NET any -> $EXTERNAL_NET 1024: (msg:"ET MALWARE Win32/HunterSte$
alert http $EXTERNAL_NET any -> $HTTP_SERVERS any (msg:"ET WEB_SPECIFIC_APPS Th$
                              [ 42665 linhas lidas ]
```

Figura 3.17 – Lista de regras do Suricata

Para que a captura funcione corretamente, de tal forma que a interface correta seja utilizada e que os *logs* sejam gerados conforme o desejado, o arquivo YAML do Suricata deve ser configurado apropriadamente. Os *logs* funcionam diferentemente, tendo aqueles referentes ao sistema, tráfego HTTP, TCP, apenas alertas, anomalias, dentre outros. Porém, o *log* a ser utilizado no projeto em questão é o Eve JSON [18], que estrutura todas as informações em um formato JSON (*JavaScript Object Notation*) sendo útil e viável para realizar a estrutura do modelo posteriormente, pelo fato do Apache Spark ter fácil integração com arquivos do tipo JSON e ter funções próprias de filtragem de campos. A escolha do Eve em detrimento de um *log* em PCAP segue pelo fato da montagem inicial do modelo de treinamento base necessitar da diferenciação dos fluxos normais e anômalos para a classificação e previsão de novos dados, além de que a estruturação do *dataset* ocorre com maior facilidade ao se utilizar o formato JSON.

No arquivo YAML, o Eve é configurado a partir de seus parâmetros e pode ser especificado o tipo de informação que se deseja coletar, como quais protocolos serão inseridos, com qual tipo de informação adicional e de, qual forma. Os campos relevantes no JSON da captura são: *timestamp*, *flow id*, *source IP*, *destination IP*, *source port*, *destination port*, *protocol*, *signature*, *signature id*, *severity*, *pkts toserver*, *pkts toclient*, *bytes toserver* e *bytes toclient*. De forma que os campos a serem utilizados no *dataset* serão explicados nos subtópicos Apache Spark e Modelo de Treinamento. Portanto, o *log* a ser utilizado será o Eve e a Figura 3.18 mostra um exemplo da captura de um alerta, contendo todos os campos escritos após o surgimento do evento, ocorrendo o destaque para aqueles citados anteriormente.

```
{
  "timestamp": "2022-09-02T17:52:10.262086-0300",
  "flow_id": 1662504211621876,
  "in_iface": "enp0s25",
  "event_type": "alert",
  "src_ip": "164.72.15.12",
  "src_port": 60577,
  "dest_ip": "164.72.15.3",
  "dest_port": 53,
  "proto": "UDP",
  "community_id": "1:LEz3HXjQpXkGOSYE080n+IM/Pc4=",
  "alert": {
    "action": "allowed",
    "gid": 1,
    "signature_id": 2101948,
    "rev": 8,
    "signature": "GPL DNS zone transfer UDP",
    "category": "Attempted Information Leak",
    "severity": 2,
    "metadata": {
      "created_at": [
        "2010_09_23"
      ],
      "updated_at": [
        "2010_09_23"
      ]
    }
  },
  "app_proto": "failed",
  "flow": {
    "pkts_toserver": 623,
    "pkts_toclient": 511,
    "bytes_toserver": 933082,
    "bytes_toclient": 27918,
    "start": "2022-09-02T17:52:08.239604-0300"
  }
}
```


Figura 3.18 – Exemplo de uma captura em JSON.

Tendo todas as regras instaladas, assim como o YAML do *software* e os parâmetros dos *logs* configurados, é possível realizar a captura do tráfego na interface então definida. A Figura 3.19 mostra a execução do monitoramento, de tal forma que cada tráfego capturado será adicionado no arquivo do *log* no formato apresentado na Figura 3.18. Então, com o tráfego sendo realizado em tempo real, o *log* final será enviado diretamente para um sistema de mensagens visando ser processado e estruturado em um modelo final. Esse processamento será explicado nos tópicos seguintes.

```
root@uiot-HP-EliteBook-8470p:~# suricata -c /etc/suricata/suricata.yaml -i enp0s
25 -D
26/8/2022 -- 15:42:01 - <Notice> - This is Suricata version 6.0.6 RELEASE runnin
g in SYSTEM mode
26/8/2022 -- 15:42:01 - <Warning> - [ERRCODE: SC_ERR_CONF_YAML_ERROR(242)] - App
-Layer protocol sip enable status not set, so enabling by default. This behavior
 will change in Suricata 7, so please update your config. See ticket #4744 for m
ore details.
26/8/2022 -- 15:42:01 - <Warning> - [ERRCODE: SC_ERR_CONF_YAML_ERROR(242)] - App
-Layer protocol mqtt enable status not set, so enabling by default. This behavio
r will change in Suricata 7, so please update your config. See ticket #4744 for
more details.
26/8/2022 -- 15:42:01 - <Warning> - [ERRCODE: SC_ERR_CONF_YAML_ERROR(242)] - App
-Layer protocol rdp enable status not set, so enabling by default. This behavior
 will change in Suricata 7, so please update your config. See ticket #4744 for m
ore details.
root@uiot-HP-EliteBook-8470p:~#
```

Figura 3.19 – Captura do Suricata em execução.

3.2.3 Apache Kafka

O Apache Kafka [4], tendo sua descrição de implementação em um IDS no artigo [50], é uma aplicação que funciona sendo um sistema de mensagens como registros ou eventos, contendo um serviço de fluxo de dados de tal forma que estes são processados no destino, ou seja, em tempo real. O Kafka funciona com *logs* tanto no quanto em seu armazenamento, trabalho com um grande volume de dados, sendo então escalável, distribuído e trabalhando em altas velocidades de transferência e disponibilidade. O seu funcionamento se baseia em escrita (publicação) de dados e leitura (inscrição) de dados em uma aplicação. Portanto, este projeto tem o objetivo de utilizar o Apache Kafka, tendo alta capacidade de integração com sistemas que utilizam *logs*, para realizar o processamento em tempo real da amostra de tráfegos capturados pela controladora e para enviá-los de forma contínua a uma aplicação de estruturação que irá transformar os dados em um modelo de aprendizado de máquina.

Sua arquitetura consiste no *cluster* de Kafka tendo servidores chamados de *brokers*, constituídos de tópicos (ou partições) que ou irão receber ou irão enviar dados durante o processamento. Os *producers* irão publicar e enviar os dados aos tópicos no *cluster*, tendo a escrita de mensagens a partir de alguma aplicação externa ao Kafka, e os *consumers* são aplicações que receberão as mensagens a partir dos tópicos ao realizarem a inscrição desses dados. Ou seja, o funcionamento do Kafka consiste na presença de um *producer* que irá gerar os dados e as mensagens a partir de uma aplicação, um *cluster* que irá registrar os dados em tópicos e posteriormente enviá-los para um *consumer* realizando o recebimento e processamento dos mesmos. A Figura 3.20 possui um diagrama representando a arquitetura básica do Apache Kafka.

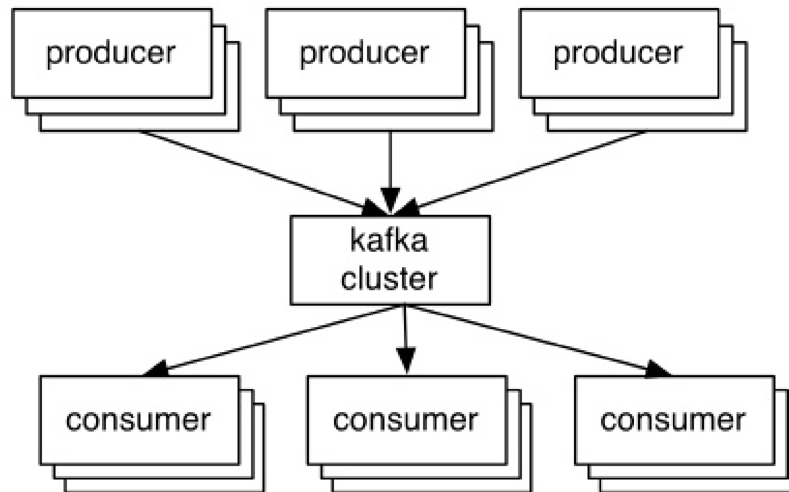

Figura 3.20 – Arquitetura do Apache Kafka. Fonte: [5]

Neste projeto, será então utilizado um tópico de entrada que receberá os dados do *log* gerado pelo Suricata agindo então como um *producer*, para ocorrer o consumo dos JSONs pelo Apache Spark. Ou seja, para realizar o processamento da captura do tráfego, a escrita e a publicação dos *logs* em JSON do Suricata será o *producer* e terá um tópico no Kafka *cluster* chamado de *test* que irá escrever os dados no Apache Spark integrado ao Kafka por *Structured Streaming*, de tal forma que este agirá como um *consumer*. Inicialmente, tendo a aplicação do Apache Kafka instalado e configurado na máquina, é criado então um tópico de entrada para armazenar os dados que serão publicados, isso sendo feito no *localhost* e na porta 9092 (utilizada pelo Kafka para levantar o *cluster*). A Figura 3.21 demonstra a criação do tópico em questão.

```
kafka@uiot-HP-EliteBook-8470p:~/kafka$ bin/kafka-topics.sh --crea
te --topic test --partitions 1 --replication-factor 1 --bootstrap
-server localhost:9092
```

Figura 3.21 – Criação do tópico de entrada.

Com o tópico criado, é executado o comando do Kafka *producer* para poder escrever e publicar dados de alguma aplicação no tópico criado anteriormente. Para fazer isso com os *logs* do Suricata, foi utilizado o comando *tail* com o objetivo de escrever os dados contidos no arquivo em tempo real

de monitoramento, sendo sensível a qualquer possível mudança no *log*. A Figura 3.22 demonstra a execução do comando do *producer* para ler os dados diretamente do arquivo em JSON, de tal forma que o *producer* fica ativo por tempo indeterminado esperando a inserção de novos dados.

```
kafka@uiot-HP-EliteBook-8470p:~/kafka$ tail -f /var/log/suricata/
eve.json | bin/kafka-console-producer.sh --bootstrap-server local
host:9092 --topic test
```

Figura 3.22 – Execução do Kafka *producer*.

Para fins de teste, foi feita a execução do Kafka *consumer* no tópico de entrada em outro terminal da máquina para verificar se a publicação e leitura dos dados está ocorrendo de forma correta. É possível ver na Figura 3.23 que os JSONs estão sendo capturados no formato de linhas (sem a formatação de um arquivo JSON) demonstrando e comprovando o funcionamento da arquitetura do Apache Kafka quanto ao fluxo de mensagens em tempo real entre duas instâncias diferentes.

```
kafka@uiot-HP-EliteBook-8470p:~/kafka$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test
{"timestamp":"2022-08-26T15:45:52.406110-0300","flow_id":1130214551728734,"in_iface":"enp0s25","event_type":"alert","src_ip":"164.72.15.130","s
rc_port":0,"dest_ip":"164.72.15.3","dest_port":0,"proto":"ICMP","icmp_type":8,"icmp_code":0,"community_id":"1:x0wqomIcFLYBXHRqV/oxaPtAYPs=","al
ert":{"action":"allowed","gid":1,"signature_id":2100366,"rev":8,"signature":"GPL ICMP_INFO PING *NIX","category":"Misc activity","severity":3}}
{"timestamp":"2022-08-26T15:45:52.406110-0300","flow_id":1130214551728734,"in_iface":"enp0s25","event_type":"alert","src_ip":"164.72.15.130","s
rc_port":0,"dest_ip":"164.72.15.3","dest_port":0,"proto":"ICMP","icmp_type":8,"icmp_code":0,"community_id":"1:x0wqomIcFLYBXHRqV/oxaPtAYPs=","al
ert":{"action":"allowed","gid":1,"signature_id":2100384,"rev":6,"signature":"GPL ICMP_INFO PING","category":"Misc activity","severity":3}}
{"timestamp":"2022-08-26T15:45:52.406149-0300","flow_id":1130214551728734,"in_iface":"enp0s25","event_type":"alert","src_ip":"164.72.15.3","src
_port":0,"dest_ip":"164.72.15.130","dest_port":0,"proto":"ICMP","icmp_type":0,"icmp_code":0,"community_id":"1:x0wqomIcFLYBXHRqV/oxaPtAYPs=","al
ert":{"action":"allowed","gid":1,"signature_id":2100408,"rev":6,"signature":"GPL ICMP_INFO Echo Reply","category":"Misc activity","severity":3}
}
```

Figura 3.23 – Execução do Kafka *consumer*.

Porém, não será utilizado um Kafka *consumer* para ler os dados que estão sendo produzidos pelo Kafka *producer*. O papel do *consumer* na arquitetura proposta será do Apache Spark com *Structured Streaming* que irá receber os JSONs propagados pelo Suricata com o objetivo de processá-los e transformá-los a partir dos campos selecionados. Portanto, os dados irão do Suricata para o Kafka, e do Kafka para o Apache Spark com o intuito de finalizar o processamento do fluxo de dados. Ou seja, o Apache Kafka serve como uma ponte entre a captura e a estruturação das informações que serão treinadas e analisadas posteriormente. A Figura 3.24 apresenta o diagrama do fluxo de dados do módulo de detecção, indicando a produção e o consumo especificado neste projeto.

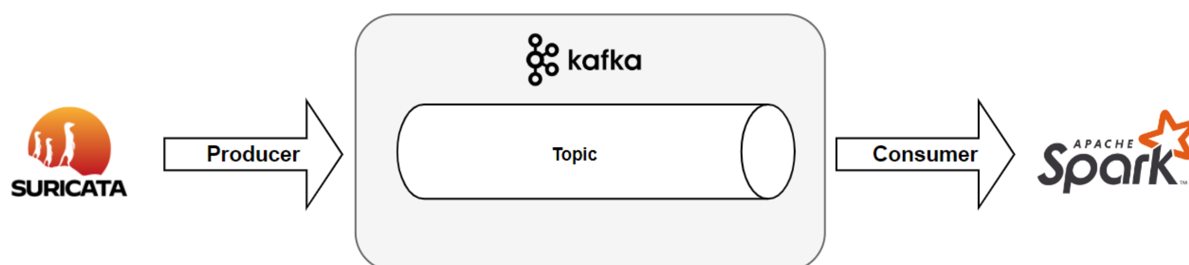

Figura 3.24 – Estrutura de *producer* e *consumer* do projeto.

3.2.4 Apache Spark com Structured Streaming

O Apache Spark [7], *software* utilizado em um IDS no trabalho [16], é uma ferramenta de processamento, de transformação e de análise de dados em tempo real e de forma distribuída, tendo a utilização de SQL, Data Science, processamento de grafos e aplicação de *Machine Learning* no fluxo de dados, os quais compõem os principais componentes do Apache Spark. Para realizar o processamento e as transformações, podem ser utilizadas três tipos de linguagens de programação como o Java, o *Python* e principalmente o *Scala*, sendo que este será utilizado neste projeto, trazendo a possibilidade de se realizar funções de filtragem, mapeamento, redução, dentre outros. A partir dessas ferramentas, é possível então gerar *Datasets/Dataframes* (seguindo o formato de colunas e valores em uma tabela) a partir de arquivos de vários formatos como o CSV (*Comma-separated Values*), o JSON, *Parquet*, dentre outros.

A arquitetura básica do Apache Spark envolve a presença de um *Cluster Manager*, responsável por administrar um grupo de *Workers* que são máquinas que realizam a execução de tarefas criadas e gerenciadas pelo componente que executa o processamento dos dados, chamado de *Driver Program*. Na estrutura da execução das tarefas gerenciado pelo *Driver Program*, existe um conjunto de objetos que armazenam o modelo de programação distribuídos no *cluster* chamados de *Resilient Distributed Dataset* (RDD), de tal forma que todas as transformações e ações nos dados a serem processados ocorrem nos RDDs. Portanto, o *cluster* armazena os RDDs contendo o modelo de programação do Spark para realizar a execução das tarefas de transformação dos dados a partir do gerenciamento do *Driver Program* e da atuação dos *Workers*. Com isso, é possível montar um *cluster* local na máquina para ocorrer o processamento dos dados que são consumidos de forma distribuída e paralela.

Então, o objetivo de se utilizar o Apache Spark neste projeto é de consumir os dados de captura da rede advindos do Kafka para utilizar o módulo de programação (em *Scala*) e transformar em tempo real as informações no formato JSON em um *Dataset/Dataframe*, de tal forma que cada campo do *log* será transformado em uma coluna contendo os valores de cada evento que foi registrado. Ou seja, o Spark tem a função de gerar um *dataset* a partir dos *logs* criados pelo Suricata. Essa etapa ocorre principalmente por uma integração realizada entre o Spark e a API do Kafka a partir do componente de *Structured Streaming* ao ler o fluxo contido no tópico do Kafka *cluster* em tempo real.

Para fazer essa integração e realizar o processamento e transformação dos *logs*, primeiramente deve ser criado um projeto na linguagem *Scala* pelo Apache Maven [6], que consiste em uma ferramenta de criação e gerenciamento de projetos assemelhando-se a um *Integrated Development Environment* (IDE). Com isso, é criado um projeto que deverá ter em sua configuração todos os detalhes de versões dos *softwares* utilizados, assim como as dependências que referenciam a integração do Apache Spark com o Apache Kafka, realizado pela modificação de um arquivo denominado POM (sigla referente a *Project Object Model*) em formato XML (*Extensible Markup Language*) que está presente no Anexo B.

Terminadas as configurações básicas no arquivo citado anteriormente, é criado então o arquivo no formato *Scala* contendo todo o código que irá ditar o funcionamento do processamento pelo

Apache Spark. Este código, estando presente no Anexo ??, contém todas as funções de leitura, transformação e escrita necessárias para gerar o *dataset*, de tal forma que ficará em execução por tempo indeterminado até a finalização do Apache Kafka *producer* pelo tópico utilizado.

Inicialmente, são inseridos os pacotes e as bibliotecas a serem utilizados no código, assim como a definição da classe principal. Após isso, é feita a função de leitura do fluxo, pela função *readStream* que coleta todos os dados presentes no tópico do Kafka (chamado de *test*)que está no localhost e na porta 9092 da controladora. Por conseguinte, com a leitura do JSON de cada evento, é atribuído os campos que serão escritos como colunas no *dataset* pela função *New Schema*. Definindo cada coluna a ser utilizada, são utilizadas algumas funções para fazer a associação dos campos com seus valores e então finalizar a estruturação do *dataset*. Por fim, é definida a função *writeStream* que irá escrever o *dataset* finalizado em alguma saída, podendo ser por console ou por um arquivo. Como o objetivo é gerar um modelo de treinamento, a escrita do *dataset* é feita no formato em CSV que distribui todos os valores na ordem de cada coluna definida anteriormente, separados por vírgulas. Tendo o modelo pronto, ocorrerá o seu envio para o servidor responsável pelo treinamento com FL.

Com o código finalizado, é feita a sua compilação para gerar um arquivo no formato JAR (*Java Archive*) visando ser executado no *cluster* do Spark. Então, para efetuar todo o procedimento, o Kafka *producer* deve estar em execução assim como mostra a Figura 3.9 e também deve ser executado o comando do Apache Spark descrevendo todas as dependências necessárias, a classe principal que está presente no código e também o caminho do arquivo no formato JAR. A Figura 3.25 mostra o comando sendo executado com todos os seus parâmetros necessários, e a Figura 3.26 apresenta no console da máquina a estrutura básica do *dataset* a ser gerado a partir da definição dos campos do JSON enquanto espera pelos dados que estão contidos no tópico do Kafka.

```
root@uiot-HP-EliteBook-8470p:/opt/spark/bin# ./spark-submit --packages org.apache.spark:
ples.spark.streaming.kafka.json.SparkStreamingConsumerKafkaJson /root/spark/target/spark
```

Figura 3.25 – Execução do Apache Spark.

```
root
 |-- flow_id: string (nullable = true)
 |-- src_ip: string (nullable = true)
 |-- dest_ip: string (nullable = true)
 |-- src_port: integer (nullable = true)
 |-- dest_port: integer (nullable = true)
 |-- proto: string (nullable = true)
 |-- hour: string (nullable = true)
 |-- minute: string (nullable = true)
 |-- seconds: string (nullable = true)
 |-- signature: string (nullable = true)
 |-- signature_id: integer (nullable = true)
 |-- severity: integer (nullable = true)
 |-- pkts_toserver: integer (nullable = true)
 |-- pkts_toclient: integer (nullable = true)
 |-- bytes_toserver: integer (nullable = true)
 |-- bytes_toclient: integer (nullable = true)
```

Figura 3.26 – Esqueleto do *dataset* no console.

Agora, o Apache Spark irá consumir os dados do Kafka e transformá-los em um *dataset*. A Figura 3.27 mostra a tabela gerada no console em tempo real, de tal forma que a cada novo evento produzido no Kafka, é gerada uma nova linha de valores referente a uma nova captura. No CSV, mesmo que podendo ser capturados em momentos diferentes, todos os eventos estarão no mesmo *dataset* pelo fato de ser utilizado o modo de escrita *append* em que os novos valores sempre serão escritos na última linha. Então, a Figura 3.28 apresenta o CSV montado após o término da execução do Spark com tráfegos de exemplo, ou seja, sem a atuação do Mirai.

```
+----------------+-------------+-------------+--------+---------+-----+----+------+-------+--------------------+------------+--------+---------
----+-------------+--------------+--------------+
|         flow_id|       src_ip|      dest_ip|src_port|dest_port|proto|hour|minute|seconds|           signature|signature_id|severity|pkts_tose
rver|pkts_toclient|bytes_toserver|bytes_toclient|
+----------------+-------------+-------------+--------+---------+-----+----+------+-------+--------------------+------------+--------+---------
----+-------------+--------------+--------------+
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     38|GPL ICMP_INFO PIN...|     2100366|       3|
   1|            0|            98|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     38|  GPL ICMP_INFO PING|     2100384|       3|
   1|            0|            98|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     39|GPL ICMP_INFO PIN...|     2100366|       3|
   2|            0|           196|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     39|  GPL ICMP_INFO PING|     2100384|       3|
   2|            0|           196|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     40|GPL ICMP_INFO PIN...|     2100366|       3|
   3|            0|           294|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     40|  GPL ICMP_INFO PING|     2100384|       3|
   3|            0|           294|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     41|GPL ICMP_INFO PIN...|     2100366|       3|
   4|            0|           392|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     41|  GPL ICMP_INFO PING|     2100384|       3|
   4|            0|           392|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     42|GPL ICMP_INFO PIN...|     2100366|       3|
   5|            0|           490|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     42|  GPL ICMP_INFO PING|     2100384|       3|
   5|            0|           490|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     43|GPL ICMP_INFO PIN...|     2100366|       3|
   6|            0|           588|             0|
|1991247268055159|164.72.15.130|  164.72.15.3|       0|        0| ICMP|  17|    05|     43|  GPL ICMP_INFO PING|     2100384|       3|
   6|            0|           588|             0|
|1060381826814308|164.72.15.130|  164.72.15.6|       0|        0| ICMP|  17|    05|     49|GPL ICMP_INFO PIN...|     2100366|       3|
   1|            0|            98|             0|
|1060381826814308|164.72.15.130|  164.72.15.6|       0|        0| ICMP|  17|    05|     49|  GPL ICMP_INFO PING|     2100384|       3|
   1|            0|            98|             0|
|1060381826814308|  164.72.15.6|164.72.15.130|       0|        0| ICMP|  17|    05|     49|GPL ICMP_INFO Ech...|     2100408|       3|
   1|            1|            98|            98|
|1060381826814308|164.72.15.130|  164.72.15.6|       0|        0| ICMP|  17|    05|     50|GPL ICMP_INFO PIN...|     2100366|       3|
```

Figura 3.27 – Saída do *dataset* no console.

```
1167891544121271,164.72.15.11,164.72.15.3,44922,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,1,0,74,0
1953846936904543,164.72.15.5,164.72.15.3,5661,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,2,0,148,0
726811287724199,164.72.15.11,164.72.15.3,62635,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,1,0,74,0
986064103650615,164.72.15.11,164.72.15.3,47354,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,1,0,74,0
915514970841271,164.72.15.11,164.72.15.3,9709,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,1,0,74,0
341988660458811,164.72.15.11,164.72.15.3,22956,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,1,0,74,0
1903539985031353,164.72.15.11,164.72.15.3,19204,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,1,0,74,0
1023099606644052,164.72.15.12,164.72.15.3,2250,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,1,0,74,0
1261633500324053,164.72.15.11,164.72.15.3,41811,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,1,0,74,0
1004704261674990,164.72.15.5,164.72.15.3,56351,53,TCP,18,07,43,SURICATA TCPv4 invalid checksum,2200074,3,1,0,74,0
```

Figura 3.28 – Saída final do *dataset* principal.

Então, foi possível gerar um *dataset* com o sistema apresentado no fluxograma da Figura 3.15, de tal forma que toda essas etapas foram feitas durante a simulação de um ataque distribuído com Mirai ao infectar os dispositivos IoT na rede para ocorrer a detecção do *malware* e das anomalias geradas após a etapa de treinamento por *Federated Learning*. A estrutura do *dataset* presente na Figura 3.26 e a sua versão de saída na Figura 3.28 corresponde a montagem do modelo principal de treinamento que servirá de base para classificar e prever novos eventos, ou seja, existem campos adicionais que foram coletados justamente para montar a classificação de cada tráfego manualmente e que serão retirados, como o *signature* e *signature id*. O procedimento de captura e estruturação do modelo será apresentado no subtópico Modelo de Treinamento, explicando a diferença da estrutura do modelo inicialmente gerado e das capturas posteriores. Na Figura 3.29 é apresentada a transformação dos *logs* em *datasets* no formato de um diagrama.

Figura 3.29 – Diagrama do processamento e transformação dos *logs*.

3.2.5 Modelo de Treinamento

Após a obtenção dos arquivos no formato CSV representando os dados de cada evento do Suricata, o modelo de treinamento está quase pronto para ser utilizado. Os campos que foram extraídos, podendo ser identificados na Figura 3.26, farão parte do conjunto de características (ou chamado de features) do *dataset* de treinamento e de teste no momento do aprendizado. A escolha de cada uma das colunas em questão tem base no trabalho [24] que utiliza do Suricata e do seu conjunto de alertas para fazer uma estruturação semelhante, e o artigo [22] que descreve os requisitos para se construir um *dataset* de NIDS eficiente.

Com base nos dois trabalhos, os campos foram selecionados tendo informações relacionadas a comunicação e o payload, como, por exemplo, os endereços e portas de origem e destino, a severidade do evento e informações de fluxo como os pacotes e bytes recebidos e enviados tanto

pelo servidor e cliente (origem e destino). Todas essas características foram determinadas para melhor treinar e classificar cada evento assim capturado durante a etapa de *Federated Learning*, visando associar os tipos de tráfegos com o padrão de comunicação que é realizado toda vez que são propagados na rede.

Porém, os campos apresentados na Figura 3.28 são referentes à montagem do modelo principal, que servirá como base para o treinamento. Os campos *signature* e *signature id* serão retirados do modelo final após o tratamento, pois a justificativa da captura dos mesmos é de realizar a classificação manual de cada fluxo com o objetivo de determinar o padrão para cada tipo de evento. Ou seja, para montar o modelo final, foram utilizados os dois campos anteriormente citados, e foi consequentemente construído um campo adicional de classificação. As capturas posteriores, que serão classificadas com base no modelo de treinamento, não possuirão esses dois campos pelo fato do projeto seguir a análise por anomalias e não por assinatura.

O novo campo adicionado é fundamental para ocorrer o treinamento de forma apropriada e consequentemente a detecção dos ataques na rede, sendo o campo de classificação. Com a obtenção do arquivo no formato CSV, foi realizado seu tratamento de forma manual para ajustar os valores e montar a nova coluna. Seguindo os padrões do *signature* e *signature id*, houve a atribuição manual seguindo os tipos de tráfegos realizados para se obter o modelo, ou seja, todas as classificações estão presentes no modelo construído. A Tabela 3.1 apresenta todas as classificações selecionadas.

Tabela 3.1 – Tipos de classificações dos eventos do modelo de treinamento.

Labels
Normal
Botnet
DoS
Scan
Bruteforce

A classificação Normal se refere aos tráfegos feitos sem apresentar nenhuma má intenção, como requisições, DNS, Ping, sincronização, dentre outros. A classificação Botnet se refere a toda atividade do *malware* utilizado para infectar os dispositivos na rede, apresentando alertas de *Trojan*, tentativa de infecção com a inserção de arquivos maliciosos, comandos de ataques aos *bots*, dentre outros. A classificação DoS se refere a todos os ataques de *Flood* que os dispositivos infectados farão no alvo após o comando pela controladora. O campo Scan se refere a tentativa de uma máquina externa a rede de tentar escanear o alvo procurando portas abertas e possibilidades de ataque e invasão, sendo que sua ocorrência é bem comum antes da realização das invasões. E por último, o campo Bruteforce se refere a tentativa de quebra de senhas e tentativas de *log*in não permitidos em diferentes serviços, de tal forma que para a construção do modelo foi utilizado o *software* Hydra em diferentes portas do alvo. Com o campo de classificação definido, é feito o tratamento do *dataset*. A descrição de cada coluna utilizada é explicada a seguir.

- *flow id:* do tipo inteiro, referente ao ID do fluxo em questão.
- *src ip:* do tipo *string*, referente ao endereço IP de origem do pacote.

- *dest ip:* do tipo *string*, referente ao endereço IP de destino do pacote.
- *src port:* do tipo inteiro, referente à porta de origem do pacote.
- *dest port:* do tipo inteiro, referente à porta de destino do pacote.
- *proto:* do tipo *string*, referente ao protocolo que o pacote está associado.
- *severity:* do tipo inteiro, referente ao nível de severidade do evento capturado.
- *hour:* do tipo inteiro, referente à hora de captura do evento.
- *minute:* do tipo inteiro, referente aos minutos de captura do evento.
- *seconds:* do tipo inteiro, referente aos segundos de captura do evento.
- *pkts to server:* do tipo inteiro, referente à quantidade de pacotes enviada ao servidor de destino.
- *pkts to client:* do tipo inteiro, referente à quantidade de pacotes enviada ao cliente de origem.
- *bytes to server:* do tipo inteiro, referente à quantidade de bytes enviada ao servidor de destino.
- *bytes to client:* do tipo inteiro, referente à quantidade de bytes enviada ao cliente de origem.
- *class:* do tipo *string*, referente à classificação do evento.

A Figura 3.30 evidencia um segmento do modelo de treinamento após o seu tratamento, contendo especialmente o campo de classificação. Assim como já foi explicado, os dois campos adicionais foram apenas utilizados para realizar a classificação, de tal forma que a estrutura de captura para os novos eventos está representada na Figura 3.31. É possível ver que o modelo a ser utilizado na etapa de testes e resultados possui 15 colunas ao todo, sendo 14 colunas de características e 1 coluna de classificação.

flow_id	src_ip	dest_ip	src_port	dest_port	proto	hour	minute	seconds	severity	pkts_toserver	pkts_toclient	bytes_toserver	bytes_toclient	class
0	164.72.15.3	164.72.15.5	0	0	ICMP	18	5	40	3	0	0	0	0	Botnet
13144473755082	164.72.15.12	164.72.15.6	173	80	UDP	18	3	44	3	3	0	3198	0	DoS
1422554963674900	164.72.15.6	164.72.15.3	0	0	ICMP	17	10	54	3	9	16	882	1568	Normal
2177543804866060	164.72.15.11	164.72.15.3	39916	53	UDP	17	52	8	2	336	278	504000	15228	DoS
95536978984890	164.72.15.12	164.72.15.6	64610	80	UDP	18	3	45	3	1	0	1066	0	DoS
214320741940090	164.72.15.11	164.72.15.6	22600	80	UDP	18	3	46	3	2	0	2132	0	DoS

Figura 3.30 – Modelo de treinamento após o tratamento dos dados.

```
root
 |-- flow_id: string (nullable = true)
 |-- src_ip: string (nullable = true)
 |-- dest_ip: string (nullable = true)
 |-- src_port: integer (nullable = true)
 |-- dest_port: integer (nullable = true)
 |-- proto: string (nullable = true)
 |-- hour: string (nullable = true)
 |-- minute: string (nullable = true)
 |-- seconds: string (nullable = true)
 |-- severity: integer (nullable = true)
 |-- pkts_toserver: integer (nullable = true)
 |-- pkts_toclient: integer (nullable = true)
 |-- bytes_toserver: integer (nullable = true)
 |-- bytes_toclient: integer (nullable = true)
```

Figura 3.31 – Estrutura de captura de novos eventos a serem classificados.

Com a obtenção do *dataset*, é finalizada a etapa de detecção do tráfego a partir do Modelo Adversário, de tal forma que ocorrerá o seu treinamento pelo método de *Federated Learning* que será explicado na seção seguinte.

3.3 Implementação do Federated Learning

Após a captura com sucesso do tráfego gerado na rede local, possuindo dados referentes a ações da Mirai *botnet*, ataques DoS, Scan, ataques por Bruteforce e comunicações benignas, é utilizado o modelo de treinamento formado anteriormente para implementar seu treinamento por aprendizado de máquina e utilizá-lo para realizar a classificação de novos pacotes capturados com o objetivo de identificar anomalias. A técnica de treinamento será feita por *Federated Learning* em um ambiente com um servidor e múltiplos clientes, tendo a finalidade de realizar todo o processo de aprendizagem de forma distribuída e com um maior nível de privacidade.

O *Federated Learning* realiza os treinamentos localmente em cada usuário pré-definido, ou seja, estes aplicam um treinamento individual, possuindo diferenças de desempenho entre os demais clientes, de tal forma que o modelo final possuirá contribuições de todos os participantes. Todo o processo de treinamento será coordenado por um servidor que possuirá todas as estratégias de aprendizado, ou seja, todos os métodos a serem utilizados, como a forma de agregação dos resultados, passagem de parâmetros, número de clientes que participarão, número de rodadas, número de épocas para cada cliente, dentre outras características a serem definidas posteriormente. Os clientes possuirão o modelo localmente, e a partir de métodos de divisão de conjuntos de treinamento e teste, aprendizado por Redes Neurais Convolucionais, realizarão todo o procedimento de treinamento e avaliação do *dataset*, de tal forma que todo o processo de montagem será descrito nos subtópicos seguintes.

Para definir a estrutura do FL tanto no servidor quanto nos clientes, será utilizado o *framework* chamado de *Flower* [23], sendo uma ferramenta open-source de fácil implementação que permite a

integração do *Python* com diversos tipos de linguagens diferentes de *Federated Learning* e métodos de aprendizado, como, por exemplo, o *TensorFlow* e o *PyTorch*. A justificativa da escolha deste framework consiste no fato de que este apresenta todos os materiais necessários para se realizar o treinamento desejado, sendo altamente modificável e customizável, apresentando diversas maneiras de implementação de diferentes estratégias. É um *framework* simples que apresenta uma grande variedade de implementações de forma de treinamento e avaliação de modelos, também podendo ser aplicado em infraestruturas de *Cloud Computing*, redes Mobile e em redes IoT, trazendo a capacidade de se aplicar treinamentos em dispositivos Android, Raspberry Pi e em NVIDIA Jetson.

A implementação do *Federated Learning* neste projeto é feito com o *Flower Framework*, uma máquina virtual em Ubuntu 20.04.06 rodando a aplicação do servidor que controla o treinamento e dois dispositivos IoT, especificamente Raspberry Pi, agindo como clientes. A estrutura do código de aprendizado em todos os componentes será por *TensorFlow* com *Keras*, incluindo bibliotecas como o Pandas e Skicit-Learn, tendo a presença de Redes Neurais Convolucionais (CNN) por Múltiplas Camadas a partir do *dataset* em CSV, sendo este dividido em um conjunto de treinamento e outro de teste, para tanto ocorrer a validação e avaliação do desempenho de cada cliente em termos de acurácia e perda. O servidor possuirá as estratégias para o funcionamento do FL, de tal forma que ele define o número de rodadas, envia os parâmetros iniciais e realizará a etapa de agregação dos parâmetros resultantes de cada cliente por *Federated Averaging*. A comunicação dos componentes é feita por gRPC por uma porta de aplicação específica, tendo a utilização de uma camada de segurança por SSL (*Secure Sockets Layer*) para aumentar o nível de privacidade do treinamento.

A estrutura dos componentes será explicada com detalhes nos subtópicos a seguir, de tal forma que a estrutura lógica do FL está representada na Figura 3.32.

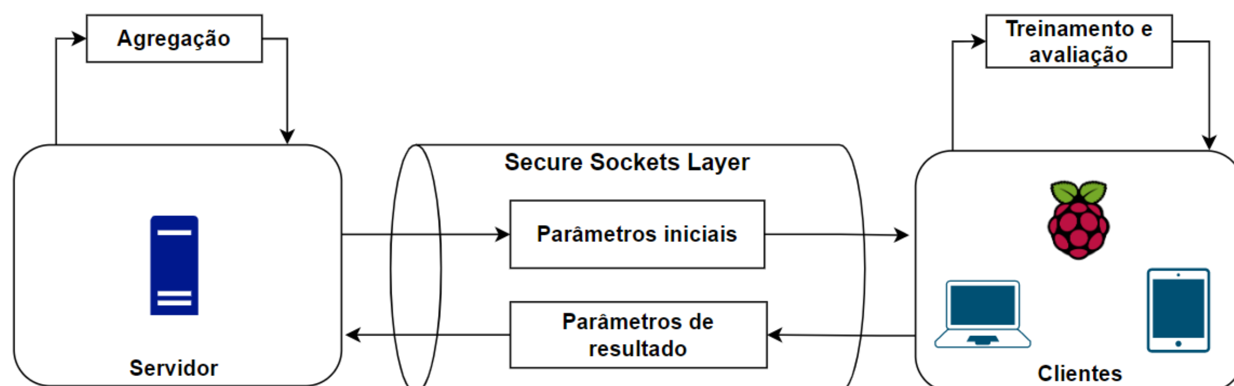

Figura 3.32 – Estrutura básica do *Federated Learning*.

O funcionamento do procedimento de FL do início ao fim está representado no fluxograma de raias está representado na Figura 3.33. O passo a passo do treinamento pode ser descrito da seguinte forma:

1. Ocorre inicialmente a definição dos parâmetros de execução dos treinamentos a partir do servidor: modelo a ser utilizado, número de rodadas, número de épocas, tamanho do *batch*

2. Os clientes fazem a divisão do dataset a partir de uma função de aleatorização, formando um conjunto de treinamento e um conjunto de testes.

3. Realiza a inicialização dos parâmetros de treinamento fazendo uma execução do modelo de aprendizado.

4. Cada cliente realiza o treinamento individual a partir do modelo pré-selecionado.

5. Fazem a avaliação dos seus treinamentos individuais a partir de métricas de acurácia e perda.

6. Enviam o treinamento finalizado em forma de pesos de resultado para o servidor.

7. O servidor obtém os pesos e faz uma junção de todos eles a partir de um algoritmo de agregação, gerando então um modelo final a partir da contribuição de todos os clientes.

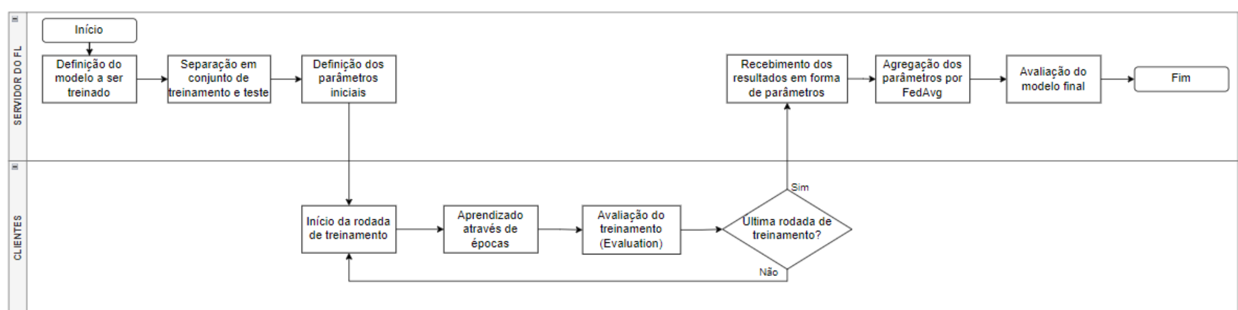

Figura 3.33 – Fluxograma do treinamento de FL.

3.3.1 Servidor

O servidor é a parte fundamental para que todo o processo ocorra de forma eficiente e segura. Seu papel, na estrutura de aprendizado, é: definir todas as estratégias de treinamento do processo, levantar um servidor gRPC para realizar a comunicação com os clientes, criar a camada de segurança por SSL ao gerar os certificados com pares de chaves, enviar parâmetros de treinamento com uma avaliação inicial e fazer a agregação dos parâmetros resultados dos treinamentos posteriores. O seu código é definido em *Python* com a biblioteca *TensorFlow* e *Keras* a partir da biblioteca do *Flower*, que apresenta a estrutura das estratégias que este *framework* disponibiliza. Todas as bibliotecas adicionais que foram utilizadas para implementar o modelo e realizar a leitura do *dataset* estão na Figura 3.34

Primeiramente, por uma máquina virtual com Ubuntu, foi utilizada a estrutura por *TensorFlow* para realizar as modificações necessárias visando treinar o modelo gerado neste projeto. Inicialmente, é definido a forma de inicialização do servidor, sendo necessário incluir o endereço IP, o número de rodadas aplicadas no processo, as estratégias a serem introduzidas e por último os certificados e pares de chaves utilizados para a comunicação ocorrer com a presença de SSL. A Figura 3.35 evidencia o fragmento do código referente à inicialização do servidor.

```python
from typing import Dict, Optional, Tuple
from pathlib import Path

from tensorflow import keras
import flwr as fl
import tensorflow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Flatten, Dense, Conv1D, MaxPool1D, Dropout, Input, Activation
import client
```

Figura 3.34 – Bibliotecas utilizadas no servidor.

```python
# Start Flower server (SSL-enabled) for four rounds of federated learning
fl.server.start_server(
    server_address="192.168.0.14:5050",
    config=fl.server.ServerConfig(num_rounds=5),
    strategy=strategy,
    certificates=(
        Path(".cache/certificates/ca.crt").read_bytes(),
        Path(".cache/certificates/server.pem").read_bytes(),
        Path(".cache/certificates/server.key").read_bytes(),
    ),
)
```

Figura 3.35 – Código referente à execução do servidor.

A definição das estratégias é uma etapa fundamental para o treinamento, pois é através delas que são determinadas as formas de aprendizado, avaliação e agregação. Para o projeto em questão, as seguintes estratégias serão utilizadas pelo servidor:

- O servidor realiza a inicialização dos parâmetros através do conjunto de testes e do modelo de treinamento gerado pelo lado do cliente. Os parâmetros iniciais são utilizados pelos clientes para inicializar a etapa de aprendizado.
- Aplica a agregação dos treinamentos de cada um dos clientes por *Federated Averaging* no fim da rodada, o qual realiza uma média entre todos os parâmetros de resultado para gerar uma versão final.
- Faz a avaliação do modelo agregado a cada rodada para realizar a sua atualização e então retorná-lo aos clientes.
- Define os parâmetros de configuração dos clientes, dizendo qual deve ser o tamanho do *batch* do treinamento e quantas rodadas devem ser implementadas por eles.

A Figura 3.36 mostra o fragmento do código que contém a definição das estratégias. É possível observar que também são definidos o número mínimo de clientes que devem estar disponíveis, assim como o número mínimo de clientes para se realizar o treinamento e avaliação. Portanto, o código

do cliente contém tanto a função de recebimento de configurações pelo servidor, e também a forma de gerenciamento do aprendizado federado entre os clientes durante todas as rodadas do processo.

```
strategy = fl.server.strategy.FedAvg(
    fraction_fit=1.0,
    fraction_evaluate=1.0,
    min_fit_clients=2,
    min_evaluate_clients=2,
    min_available_clients=2,
    evaluate_fn=get_evaluate_fn(model),
    on_fit_config_fn=fit_config,
    on_evaluate_config_fn=evaluate_config,
    initial_parameters=fl.common.ndarrays_to_parameters(model.get_weights()),
)
```

Figura 3.36 – Estratégias utilizadas.

Com o servidor já definido, é executado o seu código para que o processo se inicie. As etapas do processo de FL são definidas da seguinte forma: primeiramente ocorre a geração dos certificados para a comunicação em SSL pelo servidor, ocorre o início da comunicação por gRPC, o servidor inicializa os parâmetros de treinamento e de configuração, enviando-os para os clientes que se conectam ao servidor. Quando o número mínimo, pré-determinado, de clientes for atingido, estes iniciarão o treinamento do modelo chamado de *fit* que será explicado no subtópico dos clientes. Com o término dessa etapa, os pesos resultantes são recebidos pelo servidor realizando a agregação dos mesmos e consequentemente a sua avaliação chamado de *evaluation*. Após a avaliação, o modelo é atualizado e enviado de volta aos clientes, repetindo o processo até a ocorrência da última rodada. Os treinamentos e avaliações utilizam métricas de perda e acurácia para determinar o desempenho do aprendizado. A Figura 3.33 demonstra um resumo de todas as etapas citadas anteriormente.

Então, para inicializar o servidor, são gerados os pares de chaves e certificados, assim como mostra a Figura 3.37. Após isso, o servidor começa a funcionar com a inicialização dos parâmetros e a sua avaliação, assim como mostra a Figura 3.38. A partir de então ele começa o FL, esperando o surgimento de algum cliente para receber os dados e participar do processo.

```
Generating RSA private key, 4096 bit long modulus (2 primes)
..............................................................
..............................................................
..............................................................
e is 65537 (0x010001)
Generating RSA private key, 4096 bit long modulus (2 primes)
..............................................................
....................................................++++
e is 65537 (0x010001)
Signature ok
subject=C = DE, ST = HH, O = Flower, CN = 192.168.0.14
Getting CA Private Key
```

Figura 3.37 – Geração dos certificados do SSL.

```
INFO flower 2022-09-08 09:48:54,037 | app.py:119 | Starting Flower server, config: ServerConfig(num_rounds=5, round_timeout=None)
INFO flower 2022-09-08 09:48:54,237 | app.py:132 | Flower ECE: gRPC server running (5 rounds), SSL is enabled
INFO flower 2022-09-08 09:48:54,238 | server.py:86 | Initializing global parameters
INFO flower 2022-09-08 09:48:54,238 | server.py:266 | Using initial parameters provided by strategy
INFO flower 2022-09-08 09:48:54,238 | server.py:88 | Evaluating initial parameters
428/428 [==============================] - 4s 2ms/step - loss: 1.6189 - accuracy: 0.0251
INFO flower 2022-09-08 09:48:58,086 | server.py:91 | initial parameters (loss, other metrics): 1.6188786029815674, {'accuracy': 0.025116821750998497}
INFO flower 2022-09-08 09:48:58,086 | server.py:101 | FL starting
```

Figura 3.38 – Inicialização do servidor.

A seguir serão apresentados a estruturação e o funcionamento dos clientes, evidenciando a forma de leitura do *dataset* assim como a sua divisão nos conjuntos de treinamento e teste, além do modelo de aprendizado por Múltiplas Camadas que foi utilizado com o intuito de treinar a rede e realizar as classificações. O código de referência do servidor feito no *Flower framework* pode ser visualizado no Anexo D.

3.3.2 Clientes

Os clientes serão inicializados através dos seus códigos em *Python*, de tal forma que estarão em execução em cada Raspberry Pi utilizadas como experimento neste projeto, dispositivos diferentes daqueles que estão infectados pelo Modelo Adversário. Serão implementados dois clientes que receberão o *dataset* da Figura 3.30 e aplicarão o seu treinamento de forma individual sem que um interfira no outro. Para se conectarem com o servidor gRPC de forma segura, utilizarão o certificado que foi gerado pelo servidor quando este for executado pela primeira vez. Ao se conectarem, cada um receberá os parâmetros de inicialização e configuração do servidor para iniciarem os seus treinamentos individuais. Os clientes utilizarão várias bibliotecas para montar a estrutura do código a ser explicado, de tal forma que todas elas estão representadas na Figura 3.39.

```python
import argparse
import os
from pathlib import Path

from tensorflow import keras
import tensorflow as tf
import flwr as fl
import pandas as pd
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Flatten, Dense, Conv1D, MaxPool1D, Dropout, Input, Activation
from sklearn import preprocessing
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import SMOTE
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from imblearn.pipeline import Pipeline
from sklearn.utils import class_weight
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
```

Figura 3.39 – Bibliotecas do código dos clientes.

3.3.2.1 Divisão do conjunto de treinamento

Primeiramente, os clientes irão ler o *dataset* que está no diretório de execução do projeto através da biblioteca *Pandas* e irão montar dois tipos de conjuntos: um de treinamento e outro de teste. Os dois conjuntos são formados através da divisão do *dataset* completo, de tal forma que para este projeto foram definidas as proporções de 80 por cento para treino e 20 por cento para testes, e essa divisão é feita de forma aleatória a partir de uma função de aleatorização. As proporções foram feitas dessa forma pelo fato de ser necessário realizar um treinamento em grande parte do *dataset*, para não ocorrer aprendizagens em um mesmo conjunto de dados em todas as rodadas, de tal forma que iria causar um overfitting.

Cada um desses conjuntos organiza o *dataset* em colunas de características ou também chamado de *Features* e uma coluna de classificação que contém cada *Label* encontrado na Tabela 3.1, sendo utilizado para classificar cada linha representando um evento do tráfego capturado pelo Módulo de Detecção.

Pelo fato do modelo apresentar um certo desbalanceamento por conta da grande quantidade de pacotes DoS, foi feito tanto um *Under-Sampling* para reduzir os dados referentes ao DoS por um processo de remoção de linhas muito semelhantes, e também um *Over-Sampling* das outras classes a partir da geração de novas amostras sintéticas, com o objetivo de deixar o *dataset* mais balanceado. A razão de cada balanceamento será elaborada no tópico de resultados. A Figura 3.40 mostra a divisão do número de amostras para cada classe e a Tabela 3.2 faz o mapeamento de cada *Label* com o número da classe correspondente.

Tabela 3.2 – Número das classes referentes a cada Label.

Classes	Labels
0	Botnet
1	Bruteforce
2	DoS
3	Normal
4	Scan

```
Class=2, n=5202 (34.405%)
Class=0, n=2728 (18.042%)
Class=4, n=2219 (14.676%)
Class=3, n=2327 (15.390%)
Class=1, n=2644 (17.487%)
```

Figura 3.40 – Distribuição das classes.

Então, ocorre o ajuste dos dados por *Scaling* para refinar o formato dos dados de treinamento e para ajustar e potencializar o desempenho do aprendizado posterior. Além disso, as colunas de características que contêm *strings* foram convertidas para valores inteiros a partir de um mapeamento por *Label Encoding*, pelo fato de que o treinamento não funciona com dados no tipo *string*.

As colunas que passaram por esse processamento de mapeamento do tipo *string* para inteiro são: *src ip*, *dest ip*, *proto* e *class*. A Figura 3.41 mostra o resultado do *dataset* após esse procedimento, contendo apenas valores numéricos em todas as colunas.

	flow_id	src_ip	dest_ip	src_port	dest_port	proto	hour	minute	seconds	severity	pkts_toserver	pkts_toclient	bytes_toserver	bytes_toclient	class
0	0	446	5	0	0	75	18	5	40	3	0	0	0	0	0
1	13144473755082	444	6	173	80	103	18	3	44	3	3	0	3198	0	2
2	1422554963674900	448	4	0	0	75	17	10	54	3	9	16	882	1568	3
3	2177543804866060	443	4	39916	53	103	17	52	8	2	336	278	504000	15228	2
4	95536978984890	444	6	64610	80	103	18	3	45	3	1	0	1066	0	2

Figura 3.41 – Estrutura do *dataset* após o mapeamento.

Com a separação das colunas e dos conjuntos a partir de suas funções, é inicializado o processo de *fit* através do modelo de treinamento. O modelo de treinamento ocorre por Múltiplas Camadas totalmente conectadas utilizando a biblioteca do *Keras* para montar cada *Layer* de filtragem. A sua presença é importante para poder filtrar e realizar a amostragem de cada dado de treinamento, com o objetivo de aplicar a classificação final com base nas categorias já citadas. O modelo aplica as múltiplas camadas e finaliza com a classificação pela ativação de uma função chamada de *Softmax*, capaz de utilizar o vetor de categorias da coluna de classificação para atribuí-las a cada evento treinado.

Além de um simples treinamento por Redes Neurais com MLP, será implementado no segmento de testes, com o intuito de realizar comparações de desempenho, as Redes Neurais Convolucionais através de 1D-CNN. Este método aplica camadas convolucionais capazes de filtrar as características mais importantes do *dataset* e fazer um *Pooling* para separá-los e utilizá-los na classificação, feito em dados de uma única dimensão. Ocorre também a utilização de camadas totalmente conectadas para realizar a classificação dos dados com base nos 5 *Labels* do experimento, sendo um método de *Deep Learning* bem eficiente e útil para analisar os resultados do projeto.

O treinamento, a partir desse modelo, é feito por épocas, ou seja, rodadas de execução do cliente para aplicar o treinamento, contendo dados de tamanho do *batch* pré-definidos pelo servidor. Após o fim das épocas, o treinamento é finalizado, sendo geradas as métricas de acurácia, perda, precisão, dentre outras, para analisar o seu desempenho. Antes de enviar os resultados para o servidor, cada cliente realiza a avaliação do seu aprendizado a partir do conjunto de testes que foi gerado com o de treinamento.

Após o fim das avaliações individuais, o servidor receberá os parâmetros de resultado e irá agregá-los para gerar a versão final, assim como já foi explicado no subtópico do Servidor. Ou seja, cada cliente lê o *dataset* e estrutura os conjuntos necessários, para então aplicar o treinamento pelo modelo pré-definido e avaliá-lo. A Figura 3.42 apresenta a leitura e separação do *dataset*, além da aplicação do *Encoding*, *Scaling* e ajuste da proporção de cada classe. A Figura 3.43 apresenta o modelo de treinamento por MLP utilizado para realizar o *fit*, e a Figura 3.44 mostra o modelo de treinamento por 1D-CNN também a ser utilizado. Ambos modelos são inicializados com o formato de entrada igual ao valor 14, pelo fato de que são 14 colunas de características a serem treinadas.

Além disso, é possível observar o modo de compilação do modelo, de tal forma que é definida a métrica de acurácia para ser observada a cada época, o *Sparse Categorical Loss* representa o método de cálculo da perda, específico para o caso de um *dataset* de multicategoria, e por último o

modo de otimização por gradiente descendente estocástico chamado de *Adam*. É possível observar que é definida uma taxa de aprendizado (*learning rate*) de 0.0001 para o otimizador *Adam*. Essa configuração tem o objetivo de tornar melhor o aprendizado do modelo para ter menores taxas de perda e maiores taxas de acurácia ao longo do treinamento.

```
def load_partition():
    out = pd.read_csv("./trainmodel.csv", on_bad_lines='skip')
    out['src_ip'] = out['src_ip'].astype('category')
    out['dest_ip'] = out['dest_ip'].astype('category')
    out['proto'] = out['proto'].astype('category')
    out['class'] = out['class'].astype('category')
    out['src_ip'] = out['src_ip'].cat.codes
    out['dest_ip'] = out['dest_ip'].cat.codes
    out['proto'] = out['proto'].cat.codes
    out['class'] = out['class'].cat.codes
    x = out.drop('class', axis=1)
    y = out.iloc[:,-1].values
    over = SMOTE(sampling_strategy={0: 3400, 1: 3300, 3: 2900, 4: 2800})
    under = RandomUnderSampler(sampling_strategy={2: 6500})
    pipeline = Pipeline(steps=[('u', under),('o', over)])
    x, y = pipeline.fit_resample(x, y)
    x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.2,random_state=123)
    counter = Counter(y_train)
    for k,v in counter.items():
            per = v / len(y_train) * 100
            print('Class=%d, n=%d (%.3f%%)' % (k, v, per))
    scaler = preprocessing.StandardScaler()
    x_train=scaler.fit_transform(x_train)
    x_test=scaler.transform(x_test)
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

    return x_train, y_train, x_test, y_test
```

Figura 3.42 – Leitura do *dataset* nos clientes.

```
model = Sequential()
model.add(Input(shape=(14,)))
model.add(Dense(32,activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(16,activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(5,activation='softmax'))
opt = keras.optimizers.Adam(learning_rate=0.0001)
model.compile(opt, "sparse_categorical_crossentropy", metrics=["accuracy"])
```

Figura 3.43 – Estrutura do modelo de treinamento por MLP.

```
model.add(Conv1D(filters=16, kernel_size=(3,), activation='relu', input_shape = (14,1)))
model.add(Conv1D(filters=32, kernel_size=(3,), activation='relu', input_shape = (14,1)))
model.add(MaxPool1D(pool_size=(3,), strides=2, padding='same'))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(32,activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(16,activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(5,activation='softmax'))
opt = keras.optimizers.Adam(learning_rate=0.0001)
model.compile(opt, "sparse_categorical_crossentropy", metrics=["accuracy"])
```

Figura 3.44 – Estrutura do modelo de treinamento por 1D-CNN.

3.3.2.2 Definição dos Modelos de Redes Neurais

Os dois modelos estruturados são divididos por várias camadas montadas a partir da biblioteca do *Keras* integrado com o próprio *TensorFlow*, de tal forma que é possível obter um resumo com mais detalhes nas Figuras 3.45 e 3.46. É possível ver que ocorre a descrição das camadas, a apresentação do formato de saída após o processo de filtragem e os parâmetros de treinamento obtidos por cada uma das camadas.

O primeiro modelo é descrito apenas por MLP com camadas profundas e conectadas, pois possui 6 camadas de Redes Neurais de forma sequencial para poder realizar a classificação de um *dataset*. Após a camada visível *Input*, que determina o formato inicial do modelo a ser treinado, vem a camada escondida chamada de *Dense* que determina o modelo como *Deep Learning*, de forma que possui diversos neurônios recebendo as entradas e, por processos de multiplicação de matrizes e vetores, treinam e geram parâmetros de saída. O modelo possui três camadas de *Dense*, sendo uma gerando a saída com formato 32, a outra com formato 16 e a última com formato igual a 5. O *Dropout*, realizado entre camadas escondidas, serve para reduzir o *overfitting* após abandonar uma certa quantidade de conjuntos de neurônios que possam tornar o treinamento polarizado para garantir robustez ao modelo. Por último, tem a camada de ativação para realizar a classificação final, de tal forma que o formato de saída deve ser igual ao número de classificações possíveis, sendo igual a 5 para o caso deste projeto.

O resumo do 1D-CNN possui uma estrutura semelhante em relação ao modelo de Redes Neurais comum, porém com 3 camadas adicionais referentes ao processo de convolução. A camada Conv1D é responsável por filtrar todo o *dataset* a partir das dimensões do formato pré-definido, com o objetivo de observar e coletar os detalhes que trarão mais impacto no momento de classificação. O *MaxPooling1D* é responsável por simplificar as informações e escolher as principais características extraídas pela camada de convolução, ou seja, resumindo todas as filtragens obtidas para um número menor e mais concentrado de forma que diminui a quantidade de pesos que devem ser aprendidos nas próximas camadas. A camada *Flatten* serve para mudar o formato da estrutura após o *Pooling* de tal forma que irá transformar o mapeamento em uma coluna que entrará nas camadas totalmente conectadas de Redes Neurais, ou seja, os dados após o *Pooling* são transformados em um vetor de entrada para entrar nas camadas profundas e então prosseguir com o aprendizado. O restante das camadas segue o mesmo padrão do outro modelo, consistindo no processamento e classificação dos dados que já passaram pelas camadas convolucionais.

A justificativa da utilização do 1D-CNN neste projeto se deve pelo fato de ser uma técnica de aprendizagem profunda com a utilização de Redes Neurais para a identificação de comportamentos complexos. O seu método de funcionamento, se baseando em filtragem e extração de características fundamentais para a análise, contribui para a detecção mais detalhada de cada tipo de padrão presente nos eventos do *dataset*.

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 32)	480
dropout (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 16)	528
dropout_1 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 5)	85

Total params: 1,093
Trainable params: 1,093
Non-trainable params: 0

Figura 3.45 – Dados das camadas referente ao modelo comum.

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 12, 16)	64
conv1d_1 (Conv1D)	(None, 10, 32)	1568
max_pooling1d (MaxPooling1D)	(None, 5, 32)	0
dropout (Dropout)	(None, 5, 32)	0
flatten (Flatten)	(None, 160)	0
dense (Dense)	(None, 32)	5152
dropout_1 (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 16)	528
dropout_2 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 5)	85

Total params: 7,397
Trainable params: 7,397
Non-trainable params: 0

Figura 3.46 – Dados das camadas referente ao 1D-CNN.

Para o *fit* ser realizado, consistindo na utilização dos dois modelos citados acima para realizar o treinamento do *dataset*, é feito o comando *history* que consiste na definição dos parâmetros que serão necessários na execução do treinamento. O *history* importa o conjunto de treinamento, define o tamanho do *batch* informado pelo servidor, define o número de épocas também informado pelo servidor, o modo *verbose* que representa a forma de apresentação de finalização das épocas e por último a porcentagem de *split* da validação para obter as métricas desejadas. A Figura 3.47 evidencia a estrutura do *history* no código dos clientes.

```python
# Get hyperparameters for this round
batch_size: int = config["batch_size"]
epochs: int = config["local_epochs"]
# Train the model using hyperparameters from config
history = self.model.fit(
    self.x_train,
    self.y_train,
    batch_size,
    epochs,
    verbose=2,
    validation_split=0.3,
)
```

Figura 3.47 – Dados de configuração do aprendizado.

3.3.2.3 Definição de Parâmetros e Execução do Treinamento

Visando gerar resultados para fazer a análise do desempenho da estrutura de treinamento no tópico de Resultados e Análise, após o *fit* do modelo de treinamento pelo comando *history* que foi explicado anteriormente, é executado o segmento do código apresentado na Figura 3.48. Primeiro, será gerada a chamada Matriz de Confusão do treinamento contendo os valores de verdadeiros e falsos positivos assim como os verdadeiros e falsos negativos para cada uma das classes. Em segundo lugar, é feito o chamado *Classification Report* contendo os dados de Precisão, *Recall* e *F1-Score* de cada uma das classes. Esses resultados são gerados ao fim de uma rodada de cada um dos clientes participantes.

```python
self.y_pred = self.model.predict(self.x_test, batch_size=32, verbose=2)
self.y_pred_bool = np.argmax(self.y_pred, axis=-1)
confusion = confusion_matrix(self.y_test, self.y_pred_bool)
print('Confusion Matrix\n')
print(confusion)

print('\nClassification Report\n')
print(classification_report(self.y_test, self.y_pred_bool, target_names = ['Botnet', 'Bruteforce', 'DoS', 'Normal', 'Scan']))
```

Figura 3.48 – Resultados de saída do treinamento.

Com as estruturas definidas anteriormente, os clientes são executados (com identificações diferentes) a partir do apontamento do endereço de IP para o servidor já inicializado. Após a verificação dos certificados, ocorre a conexão e o início dos treinamentos. Os dois clientes, assim como mostram as Figuras 3.49 e 3.50, estão rodando Ubuntu Server para arquiteturas do tipo AArch64 sendo pertencente à família do ARM64, sendo referente a arquitetura de Raspberry Pi em 64 bits. A Figura 3.51 apresenta o segmento do código que revela a forma de inicialização dos clientes com o IP e porta de execução do servidor gRPC, com a leitura do certificado de SSL necessário. O terminal dos clientes é acessado via SSH na porta 22 visando realizar a execução do código do cliente.

```
root@rasp1:~# uname -a
Linux rasp1 5.4.0-1069-raspi #79-Ubuntu SMP PREEMPT Thu Aug 18 18:15:22 UTC 2022
 aarch64 aarch64 aarch64 GNU/Linux
root@rasp1:~#
```

Figura 3.49 – Identificação do cliente 1.

```
root@rasp2:~# uname -a
Linux rasp2 5.4.0-1069-raspi #79-Ubuntu SMP PREEMPT Thu Aug 18 18:15:22 UTC 2022
 aarch64 aarch64 aarch64 GNU/Linux
root@rasp2:~#
```

Figura 3.50 – Identificação do cliente 2.

```
# Start Flower client
client = CifarClient(model, x_train, y_train, x_test, y_test)

fl.client.start_numpy_client(
    server_address="192.168.0.14:5050",
    client=client,
    root_certificates=Path(".cache/certificates/ca.crt").read_bytes(),
)
```

Figura 3.51 – Configuração de inicialização dos clientes.

Então, o código em *Python* é executado no terminal de cada um dos clientes conectados por SSH, de tal forma que o servidor já deve estar em execução e apenas esperando a conexão de no mínimo dois clientes. Quando os participantes se conectam, é mostrada pela Figura 3.52 que é utilizado o certificado correto para abrir uma conexão segura por SSL, de tal forma que os clientes ficam prontos e no aguardo para receberem os parâmetros iniciais advindos do servidor com o objetivo de iniciar a primeira rodada de treinamento.

```
INFO flower 2022-09-08 13:55:04,744 | connection.py:99 | Opened secure gRPC conne
ction using certificates
DEBUG flower 2022-09-08 13:55:04,772 | connection.py:39 | ChannelConnectivity.IDL
E
DEBUG flower 2022-09-08 13:55:04,893 | connection.py:39 | ChannelConnectivity.CON
NECTING
DEBUG flower 2022-09-08 13:55:05,421 | connection.py:39 | ChannelConnectivity.REA
DY
```

Figura 3.52 – Cliente pronto para o treinamento.

Com a inicialização dos dois clientes, estes recebem os parâmetros do servidor e iniciam os seus treinamentos individuais. A Figura 3.53 mostra o início do treinamento pelo cliente 1 e a

Figura 3.54 mostra o início do treinamento pelo cliente 2. Nessas imagens, é possível ver que o treinamento das épocas ocorre com seu tempo de duração, número de passos que foram realizados, assim como os valores: *accuracy*, *loss*, *val loss* e *val accuracy*.

```
Epoch 1/5
592/592 - 18s - loss: 1.5382 - accuracy: 0.3152 - val_loss: 1.3322 - val_accuracy: 0.5570 - 18s/epoch - 30ms/step
Epoch 2/5
592/592 - 9s - loss: 1.2378 - accuracy: 0.5224 - val_loss: 0.9356 - val_accuracy: 0.6617 - 9s/epoch - 16ms/step
```

Figura 3.53 – Treinamento iniciado pelo cliente 1.

```
Epoch 1/5
592/592 - 16s - loss: 1.5333 - accuracy: 0.3295 - val_loss: 1.3225 - val_accuracy: 0.5478 - 16s/epoch - 26ms/step
Epoch 2/5
592/592 - 9s - loss: 1.2195 - accuracy: 0.5291 - val_loss: 0.9347 - val_accuracy: 0.6571 - 9s/epoch - 15ms/step
```

Figura 3.54 – Treinamento iniciado pelo cliente 2.

A Figura 3.55 mostra a agregação dos treinos e a avaliação feita pelo servidor na segunda rodada, sendo possível ver que no terminal é sempre informado qual é a etapa atual da rodada, quantos clientes participaram e se os resultados foram obtidos com sucesso ou não, de tal forma que ao fim é informada a avaliação da agregação em termos de perda e acurácia. A Figura 3.56 e a Figura 3.57 mostram a avaliação do treinamento pelos dois clientes, feito a partir do conjunto de testes formado na etapa de divisão do *dataset*. E por último, a Figura 3.58 mostra o momento do recebimento das avaliações no servidor para então prosseguir com uma nova rodada, de tal forma que essas avaliações são utilizadas para mostrar o resumo de todo o procedimento a partir da centralização dos resultados.

```
DEBUG flower 2022-09-16 08:24:46,560 | server.py:215 | fit_round 2: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-16 08:25:37,706 | server.py:229 | fit_round 2 received 2 results and 0 failures
106/106 [==============================] - 0s 2ms/step - loss: 0.3807 - accuracy: 0.8725
INFO flower 2022-09-16 08:25:38,064 | server.py:116 | fit progress: (2, 0.38070905208587646, {'accuracy': 0.8724852204322815}, 194.2828128500003)
```

Figura 3.55 – Agregação e avaliação pelo servidor.

```
5/5 [==============================] - 0s 9ms/step - loss: 1.0977 - accuracy: 0.8
375
```

Figura 3.56 – Avaliação dos treinamentos de cada cliente 1.

```
5/5 [==============================] - 0s 9ms/step - loss: 1.1037 - accuracy: 0.8
313
```

Figura 3.57 – Avaliação dos treinamentos de cada cliente 2.

```
DEBUG flower 2022-09-08 09:56:00,570 | server.py:165 | evaluate_round 2: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-08 09:56:01,077 | server.py:179 | evaluate_round 2 received 2 results and 0 failures
```

Figura 3.58 – Recebimento das avaliações no lado do servidor.

Então, está definido o ambiente de treinamento por *Federated Learning* em funcionamento com o objetivo de treinar o modelo de detecção. O modelo será treinado em várias rodadas pelos

clientes, para então ocorrer a análise de desempenho dos testes e validações para cada classificação além da ocorrência de predições de novos dados a partir do modelo já treinado. Todas as análises de desempenho, do treinamento completo, de predições e da validação da arquitetura serão feitas no tópico de Resultados e Análise. O código de referência dos clientes feito no *Flower framework* pode ser visualizado no Anexo C.

4 Resultados e Análise

Esta seção apresentará toda a análise de resultados proveniente da implementação da arquitetura que foi proposta neste projeto. A análise incluirá o funcionamento e propagação dos ataques feitos pelo Modelo Adversário na estrutura física que foi implementada e já evidenciada, demonstrando o volume de tráfego e o impacto que é gerado pelo Mirai ao servidor *Web* no alvo pré-selecionado durante a etapa de gerenciamento da *botnet*. Com isso, é observada a aplicação do modelo de detecção ao utilizar os dados de testes gerados pela divisão do modelo de treinamento, contendo informações referentes ao escaneamento da rede, ataques por Bruteforce, ataques por DoS e até mesmo tráfegos benignos.

A última etapa de resultados é a análise do treinamento do *dataset*, que foi previamente gerado, sendo executado na estrutura de *Federated Learning* que foi montada e apresentada na seção de Arquitetura Proposta. Serão descritas todas as métricas de avaliação de desempenho do treinamento e classificação a partir do modelo implementado, isso através da geração de tabelas e gráficos que apresentarão dados de acurácia, precisão, dentre outros. Então, será feita a validação da arquitetura tendo a utilização do modelo de treinamento para predizer e classificar dados de teste, com o objetivo de avaliá-los e de identificar o tipo de classificação da amostra referente a sua natureza, ou seja, se é um tráfego benigno ou malicioso. E com isso, é descrita as taxas de precisão e acurácia referente a detecção de cada categoria por meio de gráficos comparativos.

Então, o objetivo deste tópico é de definir o funcionamento da arquitetura demonstrando a conexão entre todos os componentes implementados, além de validar o modelo ao utilizá-lo para realizar a detecção de anomalias nos dados de teste. Para realizar toda a análise, foram utilizados: o Modelo Adversário com a Mirai Botnet para realizar as atividades maliciosas na implementação da arquitetura feita em um laboratório, o Módulo de Detecção presente na controladora do IDS que estrutura os tráfegos em um *dataset* de ML e o ambiente de aprendizado federado para treinar, avaliar e classificar o modelo proveniente da detecção.

A seção é dividida em duas partes: análise do volume de dados proveniente da *botnet* e a análise do desempenho de treinamento pelo *Federated Learning*.

4.1 Análise do tráfego do Modelo Adversário

Assim como já foi descrito no subtópico do Modelo Adversário na Arquitetura Proposta, a controladora do Mirai coordena os ataques de DoS ao alvo, este possuindo uma página *Web* na porta 80, que são realizados pelos 3 dispositivos IoT que foram infectados. O propósito desses ataques foi de gerar tráfegos maliciosos na rede para serem capturados e posteriormente classificados conforme o aprendizado de máquina. Dois tipos de ataques de *Flood* foram feitos na porta 80 do alvo, sendo eles: UDP e SYN *Flood*.

O UDP *Flood* é um ataque de larga escala consistindo no envio de um grande número de

pacotes do protocolo *User Datagram Protocol* para sobrecarregar o serviço pelo fato deste não ter a capacidade de processamento de responder todas as requisições que são realizadas. Ou seja, caso um serviço esteja em execução na porta de destino, o servidor irá gastar recursos para atendê-las. E no caso de não ter serviços em execução, o servidor envia uma resposta por *Ping* para avisar que o destino está inalcançável no momento.

O SYN *Flood* é um ataque que consiste na tentativa de inicializar uma conexão TCP com o servidor, de tal forma que este não finaliza o *Three-way Handshake*. Ou seja, o servidor gasta tempo de processamento computacional e consome recursos ao esperar pela finalização por parte do atacante. O atacante começa enviando um pacote de sincronização SYN, o servidor responde com um SYN e um pacote ACK confirmando a disponibilidade do serviço iniciar a conexão, porém o atacante não envia o pacote ACK para estabelecer a comunicação fazendo com que o servidor tenha um tempo de espera.

Esses dois tipos de ataques DoS foram aplicadas na rede implementada durante a etapa de experimentação e captura, de tal forma que não só foram utilizados para a montagem do *dataset*, como foi avaliado o seu impacto no alvo em termos de volume de tráfego. Foi utilizada a ferramenta *vnstat* [58] no Ubuntu para visualizar a quantidade total de *bytes* propagados ao destino, o volume do tráfego em Mbit/s, número total e médio de pacotes enviados.

É possível observar o volume do tráfego pelo ataque UDP na Figura 4.1, de tal forma que cada ataque teve a duração de 1 minuto aproximadamente. Como foi capturado a partir da perspectiva do alvo, o tráfego em estudo é observado na parte de recebimento (Rx). A quantidade de *bytes* no total se aproximou de 1 Gb em apenas 1 minuto com média de 90,44 *Mbits*/s, além de que foram enviados aproximadamente 500.000 pacotes a partir de apenas 3 *bots* simultâneos, tendo em média 7600 pacotes por segundo (cada *bot* gerando aproximadamente 3000 pacotes a cada segundo). Com poucos *bots* foi possível gerar um tráfego consideravelmente perigoso para a disponibilidade do serviço, mostrando que tendo a *botnet* em uma larga escala, é facilmente possível derrubar vários sistemas alvos diferentes, em especial aqueles na porta 80.

```
enp0s25  /  traffic statistics

                              rx        |       tx
----------------------------------------+------------------
 bytes                      706,54 MiB  |           9 KiB
----------------------------------------+------------------
          max             96,39 Mbit/s  |        3 kbit/s
      average             90,44 Mbit/s  |     1,08 kbit/s
          min                8 kbit/s   |        0 kbit/s
----------------------------------------+------------------
 packets                       492307   |             129
----------------------------------------+------------------
          max                8203 p/s   |           6 p/s
      average                7692 p/s   |           2 p/s
          min                  12 p/s   |           0 p/s
----------------------------------------+------------------
 time                     1,07 minutes
```

Figura 4.1 – Volume do tráfego por UDP *Flood*.

A Figura 4.2 mostra o volume do tráfego gerado pelo SYN *Flood*. É possível observar que a média de envio dos dados está em torno de 70,59 *Mbits*/s sendo menor que a taxa de transmissão do ataque UDP. Foram gerados em torno de 7.000.000 de pacotes com média de 115758 p/s também em 1 minuto. Ou seja, o ataque SYN apresentou um volume de dados muito maior, em termos de número de pacotes, enviados ao destino aproximadamente no mesmo tempo de execução do UDP *Flood*, mostrando que, mesmo com uma taxa menor de bits e uma quantidade menor de *bytes* enviados, este apresenta uma grande efetividade em um ataque de DDoS em uma *botnet* com maior escalabilidade pelo fato de enviar milhões de pacotes em um tempo muito pequeno. O fato deste enviar mais pacotes é pela forma de funcionamento do SYN *Flood* que consiste no envio de muitos pacotes de sincronização, tendo bem menos *bytes* que um pacote UDP, assim como é comprovado pelos dados obtidos.

```
enp0s25  /  traffic statistics

                              rx        |       tx
----------------------------------------+------------------
 bytes                      551,51 MiB  |           9 KiB
----------------------------------------+------------------
          max             77,85 Mbit/s  |        3 kbit/s
      average             70,59 Mbit/s  |     1,17 kbit/s
          min                4 kbit/s   |        0 kbit/s
----------------------------------------+------------------
 packets                      7408547   |             137
----------------------------------------+------------------
          max              127565 p/s   |           4 p/s
      average              115758 p/s   |           2 p/s
          min                   8 p/s   |           0 p/s
----------------------------------------+------------------
 time                     1,07 minutes
```

Figura 4.2 – Volume do tráfego por SYN *Flood*.

Com o volume do tráfego capturado para os dois ataques, foi estruturada uma tabela contendo todos os dados pertinentes para compará-los. Então, é possível observar que o pacote UDP é enviado em uma taxa menor de pacotes por segundo e apresenta mais *bytes* pelo fato do pacote UDP ser maior que o pacote de sincronização SYN. Mesmo a uma taxa menor de Mbit/s, o SYN gera bem mais pacotes no mesmo intervalo de tempo, de tal forma que os dois tipos de ataques são eficientes para fazer um ataque de DoS a um alvo, mas apresentam particularidades durante a transmissão. A Tabela 4.1 apresenta os dados coletados.

Tabela 4.1 – Volume do tráfego gerado pelos ataques de DoS.

DoS	Mbit/s	Pacotes/s	Total Bytes	Total Pacotes
UDP *Flood*	90,44	7692	706,54	492307
SYN *Flood*	70,59	115758	551,51	7408547

4.2 Análise de desempenho do Federated Learning

A etapa de *Federated Learning* é importante para realizar o treinamento do modelo utilizado para realizar as detecções por anomalias em uma rede de Internet das Coisas, de tal forma que foi realizada a partir de um servidor principal responsável por coordenar o processo e juntar os resultados em um só modelo final, e também de dois clientes que realizam o treinamento a cada rodada, gerando sempre um conjunto de outputs referentes a várias métricas e parâmetros de desempenho em que serão utilizados para avaliar o aprendizado. O servidor foi implementado em uma máquina virtual Ubuntu 20.04, enquanto os clientes são Raspberry Pi 3 Modelo B com Ubuntu Server, de tal forma que ambos possuem o *Flower Framework* configurado em linguagem *TensorFlow* e *Keras*.

Para fins de teste e obtenção de resultados neste projeto, assim como descrito no tópico de *Federated Learning*, serão aplicadas 3 rodadas de treinamento de 5 épocas cada com *batch size* igual a 16, de tal forma que o *dataset* formado a partir da captura é dividido em um conjunto de treinamento e um de teste a partir de um *split* de 80 para 20 por cento. O modelo é por multi-categoria, apresentado 5 categorias de classificação para definir o tipo de evento presente no *dataset* que serão utilizados para fins de detecções em dados novos que não participaram do processo de aprendizado. Para a ocorrência do treinamento por múltiplas camadas e por Redes Neurais Convolucionais de forma apropriada, o *dataset* é lido no formato CSV e separado de forma aleatória. E por último, é feito o seu *Scaling* para definir de forma correta e apropriada o formato dos vetores.

Durante a ocorrência dos treinamentos e das avaliações do modelo tanto pelos clientes quanto pelo servidor, são geradas, em tempo real, métricas de acurácia e perda para cada época, de tal forma que no fim de todo o procedimento é apresentada a centralização e média desses dois parâmetros a partir de todas as rodadas realizadas. Além disso, para fins de análise de desempenho do procedimento, no fim de cada rodada, serão geradas as seguintes métricas para avaliar os testes e o desempenho da classificação de cada classe presente:

- Geração da Matriz de Confusão que mostra o número de previsões corretas e incorretas

de cada classe apresentando os valores de Verdadeiro Positivo (TP), Falso Positivo (FP), Verdadeiro Negativo (TN) e Falso Negativo (FN)

- Realizar um Classification Report, consistindo em apresentar os valores de Precisão, *Recall* e *F1-Score* para cada classe, mostrando a média de cada um dos valores com e sem os pesos de classificação.

Com esses dados, coletados principalmente na primeira e terceira rodada, por cada cliente, para fins de comparação, serão feitas análises através das tabelas dos desempenhos a serem avaliados, assim como gráficos apresentando as curvas de Acurácia e Perda do modelo, agregado pelo servidor, assim como histogramas mostrando a comparação entre a Precisão, *Recall* e *F1-Score* de cada classe. O objetivo desses procedimentos é de observar a evolução do modelo ao fim do aprendizado federado. Com a utilização dessas métricas, é possível julgar a eficácia do modelo gerado neste projeto, além do nível de precisão e acurácia que este tem de identificar amostras ainda não classificadas.

Toda a análise será feita com a presença dos dois clientes em 3 rodadas de treinamento, tanto com a utilização do modelo de Redes Neurais por camadas totalmente conectadas e também com o modelo de 1D-CNN para fazer as devidas comparações. As etapas da obtenção dos resultados e das análises será feita como descrito a seguir:

1. Análise e justificativa do balanceamento do *dataset* a partir de cada classe com *Under-Sampling* e *Over-Sampling*.

2. Execução do treinamento completo e obtenção dos valores de acurácia e perda do modelo agregado pelo servidor, assim como a inicialização dos parâmetros.

3. Obtenção dos resultados de treinamento da primeira e última rodada de cada cliente a partir dos dois modelos utilizados.

4. Construção de gráficos para analisar o desempenho dos clientes quanto a classificação de cada categoria presente no *dataset*.

4.2.1 Modelo Agregado e Inicialização dos Parâmetros

Primeiramente, para prosseguir com os treinamentos a partir do *dataset* gerado pela controladora do IDS, foi feito o seu balanceamento [32]. O balanceamento é necessário para não ocorrer uma grande prevalência de uma classe em detrimento da outra, pois os resultados acabam ficando polarizados e são extremamente baixos para as classes com menos amostras. Então, o *dataset* montado inicialmente apresenta uma certa desproporcionalidade em relação às outras categorias, de tal forma que os eventos relacionados ao ataque de DoS apresentavam em torno de 80 por cento de todo o *dataset*. Isso se deve ao fato de que, utilizando três máquinas infectadas para realizar os ataques, o fluxo de dados por DoS é muito grande se comparado aos outros tipos de dados. Assim como foi mostrado no subtópico anterior, os ataques de DoS geraram em torno de 6000 a 9000 pacotes por segundo, totalizando 500.000 a 700.000 pacotes em 1 minuto. Ou seja, o número

de eventos por DoS vai ser naturalmente maior devido ao grande volume de dados gerado durante sua execução.

Por isso, foi utilizado principalmente o método de *Under-Sampling* [32] na categoria de DoS, para ocorrer uma diminuição drástica das amostras e que ocorra um melhor equilíbrio entre as outras categorias. Uma desvantagem deste método é a perda de informações, que é gerada após a sua aplicação, porém, no caso dos eventos para DoS, essa perda não é tão efetiva pelo fato de um ataque de DoS gerar pacotes muito similares e próximos quanto ao seu funcionamento. Ou seja, pelo fato dos ataques DoS possuírem pacotes muito repetidos com dados de fluxos semelhantes, o *Under-Sampling* não gera tantas perdas de informações em relação a essa categoria.

Além disso, para finalizar o balanceamento, foi feito um simples *Over-Sampling* utilizando a técnica de SMOTE (*Synthetic Minority Oversampling Technique*) [14], que consiste na geração de amostras sintéticas a partir dos eventos já existentes para aumentar a quantidade de eventos. Essa técnica é feita nas categorias minoritárias, e a justificativa da sua presença neste projeto é de equilibrar as outras categorias com a de DoS para ser mantido um melhor balanceamento no *dataset*. Uma desvantagem é a possibilidade de *overfitting*, e por isso não foi mantida uma proporção totalmente igualitária entre todas as categorias. Assim como mostra a Figura 3.40 no tópico de Arquitetura Proposta, a proporção é de 35 por cento de DoS para 15 por cento das outras classes, aproximadamente. Como a classe de DoS terá mais pacotes naturalmente, ainda foi mantida como a classe majoritária e não foi estabelecida a mesma proporção para as outras classes para não deixar o *dataset* muito artificial. A partir disso, os treinamentos foram feitos para que as precisões não fossem totalmente desbalanceadas e para que cada classe possa ser treinada de forma mais eficiente.

Com a execução do servidor e o acionamento dos dois clientes, é iniciado o processo de treinamento a partir da primeira rodada. Porém, antes dos clientes darem início, eles precisam dos parâmetros iniciais do modelo de treinamento a partir do servidor. Com a estratégia de inicialização estando presente, o servidor usa um *dataset* de validação gerado a partir do *split* feito na função do cliente e faz um treinamento inicial com a utilização de 0 parâmetros de suporte. O servidor faz a avaliação desse treinamento através do conjunto de validação e gera na saída o valor de acurácia obtido, o valor de perda e os parâmetros iniciais que serão enviados para os dispositivos participantes. As Figuras 4.3 e 4.4 apresentam o resultado da avaliação dos parâmetros iniciais para o modelo comum de Redes Neurais e para o CNN de 1 dimensão.

```
INFO flower 2022-09-08 20:48:47,436 | server.py:86 | Initializing global parameters
INFO flower 2022-09-08 20:48:47,436 | server.py:266 | Using initial parameters provided by strategy
INFO flower 2022-09-08 20:48:47,437 | server.py:88 | Evaluating initial parameters
119/119 [==============================] - 1s 3ms/step - loss: 1.5704 - accuracy: 0.3648
```

Figura 4.3 – Inicialização dos parâmetros no modelo comum.

```
INFO flower 2022-09-16 08:22:22,756 | server.py:86 | Initializing global parameters
INFO flower 2022-09-16 08:22:22,756 | server.py:266 | Using initial parameters provided by strategy
INFO flower 2022-09-16 08:22:22,756 | server.py:88 | Evaluating initial parameters
106/106 [==============================] - 1s 3ms/step - loss: 1.5738 - accuracy: 0.3027
```

Figura 4.4 – Inicialização dos parâmetros no 1D-CNN.

É possível observar que o resultado de ambas as avaliações geram acurácias baixas com menos de 50 por cento. Isso se deve ao fato de que o servidor realiza um treinamento inicial sem utilizar nenhuma outra base de aprendizado, ou seja, sem a utilização de pesos previamente coletados. Então, ele faz uma rodada básica de treinamento sem qualquer tipo de informação inicial. Essa etapa é importante para que os clientes possam realizar seus treinos com informações iniciais que contribuem para resultados mais altos em poucas rodadas de treinamento. A Tabela 4.2 contém as métricas citadas anteriormente para os dois modelos e o gráfico da Figura 4.5 mostra as proporções entre eles, evidenciando que o 1D-CNN apresenta maior acurácia e menor perda em relação ao modelo comum.

Tabela 4.2 – Métricas da inicialização dos pesos de cada modelo.

Modelo	Acurácia	Perda
Comum	0.1989	1.609
1D-CNN	0.3027	1.573

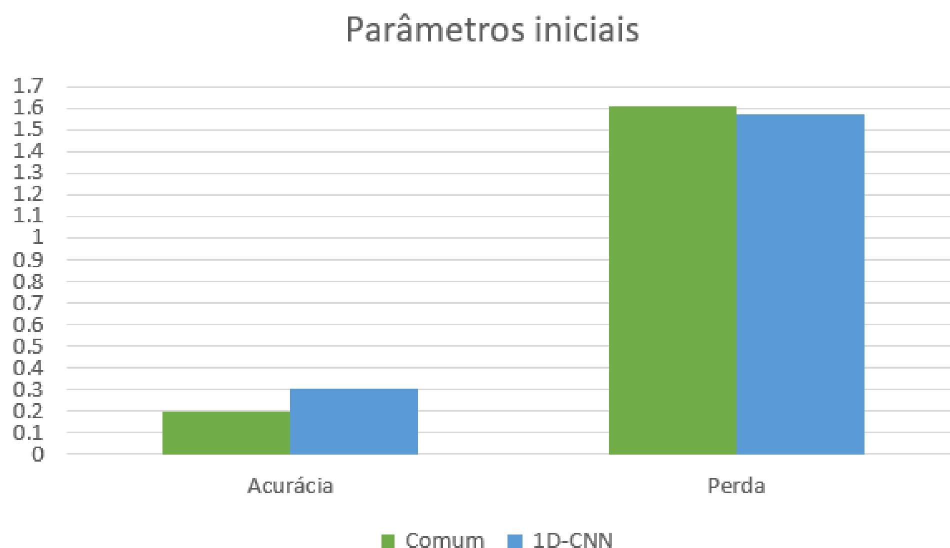

Figura 4.5 – Gráfico de métricas dos parâmetros iniciais dos dois modelos.

Os treinamentos foram feitos em 3 rodadas por 2 clientes, de tal forma que sempre é iniciado pelo processo de *fit*, o qual ocorre o aprendizado do modelo, tal que logo em seguida o servidor recebe os pesos de resultados e faz a agregação para gerar um modelo atualizado a partir dos dois clientes juntamente da sua avaliação. Após isso, os clientes avaliam os seus próprios treinamentos para retornar valores de acurácia, perda e também realizar o *Classification Report* que será explicado adiante. As Figuras 4.6 e 4.7 mostram todas as etapas do treinamento pela perspectiva do servidor dos dois modelos utilizados.

```
INFO flower 2022-09-08 20:48:48,271 | server.py:101 | FL starting
DEBUG flower 2022-09-08 20:49:46,984 | server.py:215 | fit_round 1: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-08 20:50:02,018 | server.py:229 | fit_round 1 received 2 results and 0 failures
WARNING flower 2022-09-08 20:50:02,038 | fedavg.py:243 | No fit_metrics_aggregation_fn provided
119/119 [==============================] - 0s 2ms/step - loss: 1.1722 - accuracy: 0.7653
INFO flower 2022-09-08 20:50:02,453 | server.py:116 | fit progress: (1, 1.1722006797790527, {'accuracy': 0.7653439044952393}, 74.18182175300171)
DEBUG flower 2022-09-08 20:50:02,453 | server.py:165 | evaluate_round 1: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-08 20:50:02,970 | server.py:179 | evaluate_round 1 received 2 results and 0 failures
WARNING flower 2022-09-08 20:50:02,970 | fedavg.py:274 | No evaluate_metrics_aggregation_fn provided
DEBUG flower 2022-09-08 20:50:02,970 | server.py:215 | fit_round 2: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-08 20:50:12,913 | server.py:229 | fit_round 2 received 2 results and 0 failures
119/119 [==============================] - 0s 4ms/step - loss: 1.0974 - accuracy: 0.8087
INFO flower 2022-09-08 20:50:13,628 | server.py:116 | fit progress: (2, 1.0973740816116333, {'accuracy': 0.8087301850318909}, 85.35672429599799)
DEBUG flower 2022-09-08 20:50:13,628 | server.py:165 | evaluate_round 2: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-08 20:50:14,075 | server.py:179 | evaluate_round 2 received 2 results and 0 failures
DEBUG flower 2022-09-08 20:50:14,075 | server.py:215 | fit_round 3: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-08 20:50:23,655 | server.py:229 | fit_round 3 received 2 results and 0 failures
119/119 [==============================] - 0s 2ms/step - loss: 1.0784 - accuracy: 0.8246
INFO flower 2022-09-08 20:50:23,964 | server.py:116 | fit progress: (3, 1.0784446001052856, {'accuracy': 0.8246031999588013}, 95.69290921100037)
DEBUG flower 2022-09-08 20:50:23,964 | server.py:165 | evaluate_round 3: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-08 20:50:24,417 | server.py:179 | evaluate_round 3 received 2 results and 0 failures
```

Figura 4.6 – Treinamento completo no modelo comum.

```
INFO flower 2022-09-16 08:22:23,781 | server.py:101 | FL starting
DEBUG flower 2022-09-16 08:23:44,246 | server.py:215 | fit_round 1: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-16 08:24:45,168 | server.py:229 | fit_round 1 received 2 results and 0 failures
WARNING flower 2022-09-16 08:24:45,178 | fedavg.py:243 | No fit_metrics_aggregation_fn provided
106/106 [==============================] - 1s 7ms/step - loss: 0.5264 - accuracy: 0.8157
INFO flower 2022-09-16 08:24:45,998 | server.py:116 | fit progress: (1, 0.5263617634773254, {'accuracy': 0.8156804442405701}, 142.21742228100038)
DEBUG flower 2022-09-16 08:24:45,999 | server.py:165 | evaluate_round 1: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-16 08:24:46,560 | server.py:179 | evaluate_round 1 received 2 results and 0 failures
WARNING flower 2022-09-16 08:24:46,560 | fedavg.py:274 | No evaluate_metrics_aggregation_fn provided
DEBUG flower 2022-09-16 08:24:46,560 | server.py:215 | fit_round 2: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-16 08:25:37,706 | server.py:229 | fit_round 2 received 2 results and 0 failures
106/106 [==============================] - 0s 2ms/step - loss: 0.3807 - accuracy: 0.8725
INFO flower 2022-09-16 08:25:38,064 | server.py:116 | fit progress: (2, 0.38070905208587646, {'accuracy': 0.8724852204322815}, 194.28281285000003)
DEBUG flower 2022-09-16 08:25:38,064 | server.py:165 | evaluate_round 2: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-16 08:25:38,593 | server.py:179 | evaluate_round 2 received 2 results and 0 failures
DEBUG flower 2022-09-16 08:25:38,593 | server.py:215 | fit_round 3: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-16 08:26:30,650 | server.py:229 | fit_round 3 received 2 results and 0 failures
106/106 [==============================] - 1s 7ms/step - loss: 0.3182 - accuracy: 0.8852
INFO flower 2022-09-16 08:26:31,487 | server.py:116 | fit progress: (3, 0.3181883990764618, {'accuracy': 0.8852071166038513}, 247.70555752500013)
DEBUG flower 2022-09-16 08:26:31,487 | server.py:165 | evaluate_round 3: strategy sampled 2 clients (out of 2)
DEBUG flower 2022-09-16 08:26:32,012 | server.py:179 | evaluate_round 3 received 2 results and 0 failures
```

Figura 4.7 – Treinamento completo do 1D-CNN.

Quando o treinamento finaliza, ou seja, ocorre a última rodada de treinamento e a avaliação dos clientes, o servidor realiza a última agregação e centraliza todas as acurácias e perdas de todas as rodadas para descrever o desempenho de todo o processo. Essa etapa é importante para verificar a eficiência do treinamento na totalidade. A Figura 4.8 apresenta as perdas e acurácias das agregações no modelo comum e a Tabela 4.3 estrutura esses valores para melhor visualização.

```
server.py:144 | FL finished in 96.1465215520002
app.py:180 | app_fit: losses_distributed [(1, 1.1988478899002075), (2, 1.1089169383049011), (3, 1.0904510617256165)]
app.py:181 | app_fit: metrics_distributed {}
app.py:182 | app_fit: losses_centralized [(0, 1.570357084274292), (1, 1.1722006797790527), (2, 1.0973740816116333), (3, 1.0784446001052856)]
app.py:183 | app_fit: metrics_centralized {'accuracy': [(0, 0.364814817905426), (1, 0.7653439044952393), (2, 0.8087301850318909), (3, 0.8246031999588013)]}
```

Figura 4.8 – Resultados da agregação no modelo comum.

Tabela 4.3 – Métricas de desempenho da agregação no modelo comum.

Rodadas	Acurácia	Perda
1	0.765	1.172
2	0.808	1.097
3	0.824	1.078

Assim como foi feito com o modelo comum, também foram coletados e estruturados os dados de métrica de desempenho para o 1D-CNN que foi executado pelos mesmos dois clientes. A Figura 4.9 mostra os resultados da agregação total na parte do servidor e a Tabela 4.4 contém esses mesmos valores para uma melhor visualização.

```
INFO flower 2022-09-16 08:26:32,012 | server.py:144 | FL finished in 248.23131892000038
INFO flower 2022-09-16 08:26:32,013 | app.py:180 | app_fit: losses_distributed [(1, 0.5640523433685303), (2, 0.4088147431612015), (3, 0.3471970409154892)]
INFO flower 2022-09-16 08:26:32,013 | app.py:181 | app_fit: metrics_distributed {}
INFO flower 2022-09-16 08:26:32,013 | app.py:182 | app_fit: losses_centralized [(0, 1.573767900466919), (1, 0.5263617634773254), (2, 0.38070905208587646), (3, 0.3181883990764618)]
INFO flower 2022-09-16 08:26:32,013 | app.py:183 | app_fit: metrics_centralized {'accuracy': [(0, 0.30266273021698), (1, 0.8156804442405701), (2, 0.872485204322815), (3, 0.885207116603851З)]}
```

Figura 4.9 – Resultados da agregação do 1D-CNN.

Tabela 4.4 – Métricas de desempenho da agregação no 1D-CNN.

Rodadas	Acurácia	Perda
1	0.815	0.526
2	0.872	0.380
3	0.885	0.318

Então, para comparar o desempenho dos dois modelos a partir do treinamento completo, foram utilizadas as tabelas estruturadas anteriormente para montar um gráfico comparativo de desempenho tanto para a acurácia e perda associada com o modelo agregado. Fazendo a divisão por rodadas, os gráficos foram criados a partir de duas curvas lineares para cada um dos modelos. A Figura 4.10 mostra o gráfico de acurácia por rodadas, e é possível observar que o modelo 1D-CNN atingiu valores maiores durante todo o intervalo. A Figura 4.11 mostra o gráfico de perda por rodadas, e é possível observar que o 1D-CNN apresentou menores perdas durante todo o procedimento quando comparado ao modelo comum.

A justificativa para esses resultados se deve pelo fato de que o 1D-CNN é um modelo mais preciso e eficiente, pois utiliza camadas convolucionais para extrair as melhores e mais importantes características do *dataset* antes de passar pelas Redes Neurais, ao contrário do modelo comum que realiza todo o aprendizado apenas por Redes Neurais. Então, foi validado que o modelo CNN é eficiente para realizar o treinamento do modelo proposto para o projeto, tanto pela sua capacidade de aprendizado quando comparado a um modelo comum quanto ao nível de acurácia que foi atingido, se aproximando de 90 por cento na última rodada. Além disso, é possível observar que a taxa de perda do 1D-CNN foi bem abaixo de 1.0 diferentemente do modelo comum, conseguindo atingir o valor de 0.318 na última rodada.

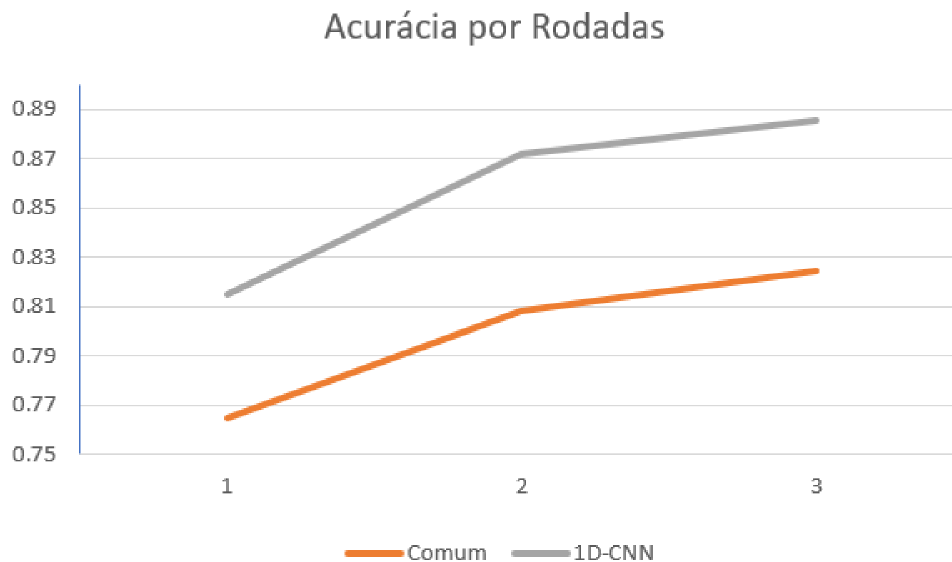

Figura 4.10 – Gráfico de desempenho da agregação em termos de acurácia.

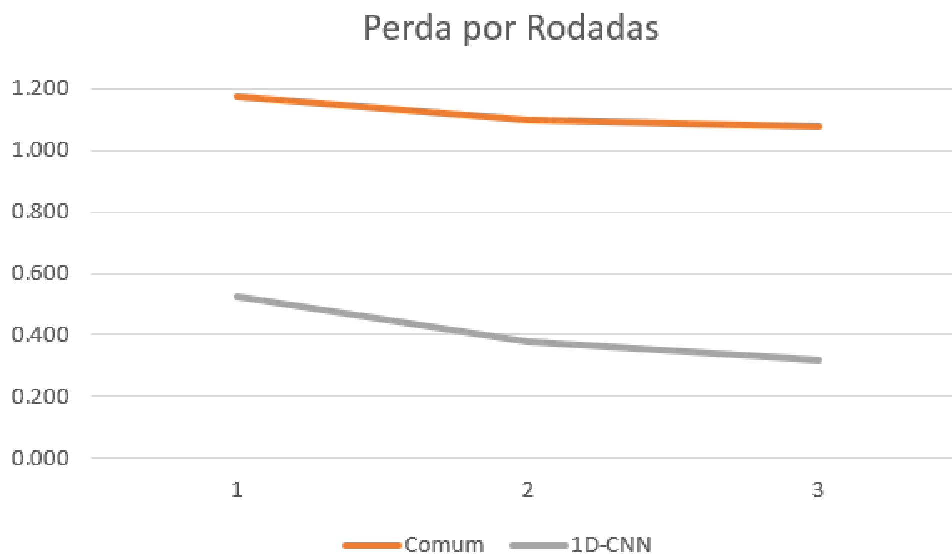

Figura 4.11 – Gráfico de desempenho da agregação em termos de perda.

4.2.2 Desempenho de Classificação das Categorias de Comportamento

Tendo analisado o desempenho do modelo agregado e completo, serão observadas as saídas de resultado provenientes do treinamento de cada cliente acerca do desempenho da classificação das categorias utilizadas neste projeto. No fim de cada rodada de treinamento, após a realização do *fit* de 5 épocas, são gerados os resultados de Matriz de Confusão e o chamado *Classification Report*. Então, para realizar análise do desempenho individual dos clientes quanto a classificação, foram coletados os resultados da primeira e terceira rodada nos dois modelos de aprendizado.

A Matriz de Confusão é uma tabela de dados onde todas as categorias de classificação são linhas e colunas simultaneamente. De tal forma, que através desta matriz, é possível observar os valores

de verdadeiro positivo, falso positivo, verdadeiro negativo e falso negativo. Esses parâmetros são referentes a predição durante a avaliação do modelo de treinamento no conjunto de testes, de tal forma que é revelado quais foram os acertos e quais foram os erros durante este momento após o treinamento. Na perspectiva de uma única categoria, o verdadeiro positivo (TP) se refere aos acertos de classificação que realmente pertencem à categoria em questão, o falso positivo (FP) é um erro onde ocorreu a classificação da categoria questão de forma errônea, o verdadeiro negativo (TN) são aqueles dados que foram corretamente classificados com base em outra categoria e os falsos negativos (TN) são aqueles dados que deveriam ser classificados com a categoria em questão, mas foi dado como um valor negativo.

Esses 4 tipos de valores são utilizados justamente para determinar o desempenho dos testes de predição e classificação de cada categoria no conjunto de dados ainda não classificados, para observar se o modelo consegue distingui-los uns dos outros quanto a sua natureza. Esses valores são utilizados para determinar as métricas de Precisão, *Recall* e *F1-Score* que serão gerados pelo *Classification Report*.

A Precisão é a relação dos verdadeiros positivos com todos os dados declarados positivos (TP e FP), determinando a taxa de acerto. Esta métrica é fundamental para os testes em questão, pois o objetivo é identificar o tráfego malicioso, então é necessário que ocorra mais verdadeiros positivos do que falsos positivos, sendo que este seria a classificação de um tráfego normal como malicioso.

O *Recall* é a métrica associada com quantos positivos são realmente positivos em relação aos falsos negativos quando estes têm um alto custo, ou seja, tem relação com dados que eram para ser positivos, mas foram declarados como negativos, no caso de se classificar um tráfego malicioso em benigno. Então, é uma métrica também fundamental para as análises posteriores, pois o objetivo é identificar o maior número de dados maliciosos possíveis.

O *F1-Score* é uma métrica para avaliar o balanceamento entre a Precisão e o *Recall* de uma determinada categoria, informando se o *dataset* precisa de uma distribuição melhor. Ou seja, quanto maior o *F1-Score*, melhor é a distribuição da classe e menores são os valores de falsos positivos e negativos identificados durante a sua avaliação.

O *Classification Report* utiliza da Matriz de Confusão para gerar as métricas citadas anteriormente, assim como a média da acurácia normal, a média macro e a média com pesos. A média macro se trata da média aritmética do desempenho de todas as classes sem a utilização de pesos, ou seja, todas as classes são tratadas da mesma forma sem distinção. A média com pesos trata cada classe diferentemente de acordo com sua contribuição e proporção no balanceamento do modelo. Ou seja, a média com pesos é o parâmetro principal para ser utilizado neste projeto, pois apresenta melhor avaliação de modelos que possuem um certo desbalanceamento. Portanto, com esses resultados, será observado o desempenho de cada classe e será utilizada a média com pesos para determinar o desempenho geral.

Então, os dois tipos de resultados foram gerados para o primeiro e último treinamento de cada cliente para os dois modelos presentes. Cada uma das métricas serão descritas e analisadas para identificar quais classes tiveram melhor desempenho em questão dos verdadeiros e falsos, positivos e negativos, além da comparação dos resultados entre os modelos. Para avaliar se a média de

F1-Score das classes com pesos foi satisfatória, foi observado se a taxa foi superior a 60 por cento.

As Figuras 4.12 e 4.13 indicam o primeiro e último treinamento do cliente 1 no modelo comum. É possível observar pelas saídas que a precisão da Botnet e do Scan tiveram um crescimento considerável em precisão, enquanto a categoria Normal e Bruteforce não apresentaram melhorias. O *Recall* também teve melhorias, de tal forma que o mais baixo é da categoria Scan pelo fato de apresentar muitos falsos negativos. A média com pesos foi superior a 80 por cento, indicando um bom desempenho para um *dataset* que ainda apresenta certo desbalanceamento quanto aos dados de DoS.

```
Confusion Matrix

[[ 483   27   48   70   44]
 [ 134  424   46   46    6]
 [  57    0 1241    0    0]
 [  50   55  129  336    3]
 [  35  180    1    0  365]]

Classification Report

              precision    recall  f1-score   support

      Botnet       0.64      0.72      0.68       672
  Bruteforce       0.62      0.65      0.63       656
         DoS       0.85      0.96      0.90      1298
      Normal       0.74      0.59      0.66       573
        Scan       0.87      0.63      0.73       581

    accuracy                           0.75      3780
   macro avg       0.74      0.71      0.72      3780
weighted avg       0.76      0.75      0.75      3780
```

Figura 4.12 – Primeiro treinamento do cliente 1 no modelo comum.

```
Confusion Matrix

[[ 496   45    2  110   19]
 [  72  528    3   46    7]
 [  19    0 1268   11    0]
 [  40   78    1  453    1]
 [  17  182    0    0  382]]

Classification Report

              precision    recall  f1-score   support

      Botnet       0.77      0.74      0.75       672
  Bruteforce       0.63      0.80      0.71       656
         DoS       1.00      0.98      0.99      1298
      Normal       0.73      0.79      0.76       573
        Scan       0.93      0.66      0.77       581

    accuracy                           0.83      3780
   macro avg       0.81      0.79      0.80      3780
weighted avg       0.84      0.83      0.83      3780
```

Figura 4.13 – Último treinamento do cliente 1 no modelo comum.

Ou seja, vários dados de Scan não estão sendo classificados corretamente, assim como o Bruteforce está classificando dados errados como positivos. Isso pode ser explicado pelo fato de que são categorias similares e que apresentam proximidades em termos de fluxo e severidade. Com

isso, mesmo que as taxas sejam maiores que 60 por cento, indicando um desempenho considerável, o modelo possui pequenos problemas ao tentar diferenciar o Bruteforce do Scan. Além disso, a categoria de DoS atingiu taxas próximas a 100 por cento pelo fato de ser o tipo de categoria mais simples de ser identificado, justificado pela alta semelhança entre os pacotes e devido às suas repetições. Mesmo que a precisão da categoria Normal não tenha tido tanta evolução, a taxa de *Recall* e *F1-Score* apresentaram melhorias indicando a eficiência de identificação da classe.

A seguir é apresentada a Tabela 4.5 referente aos dados de verdadeiros e falsos positivos e negativos de cada classe, comprovando a análise dita anteriormente. A classe Botnet apresentou resultados equilibrados e satisfatórios, possuindo mais quantidade de TP em relação ao FP e FN. A classe de Bruteforce apresentou uma grande quantidade de falsos positivos, justificando a baixa precisão, indicando que está ocorrendo a classificação de pacotes negativos como positivos. A classe de DoS apresentou uma pouca quantidade de FP e FN. A classe Normal teve um resultado satisfatório, sendo bem equilibrado quanto às métricas anteriores, indicando taxas de mais de 70 por cento. E por último a classe Scan que apresentou falsos negativos, justificando o *Recall* em torno de 60 por cento.

Tabela 4.5 – Análise da confusão de matriz final do cliente 1 no modelo comum.

Classe	TP	FP	TN	FN
Botnet	496	148	2960	131
Bruteforce	528	305	2819	56
DoS	1268	6	2506	30
Normal	453	167	3040	120
Scan	382	27	3172	199

Os mesmos tipos de resultados foram feitos para o caso do 1D-CNN, assim como mostram as Figuras 4.14 e 4.15, tanto na primeira quanto na última rodada. É possível identificar que a precisão de Botnet, do Normal, do Bruteforce e do Scan foram maiores, apresentando também uma evolução na média com pesos chegando em torno de 86 por cento. Mesmo apresentando os mesmos problemas referentes aos falsos positivos do Bruteforce e os falsos negativos do Scan, houve melhorias na classificação, com maiores taxas de acerto quanto à obtenção de dados que são verdadeiros positivos, além de uma clara redução na quantidade de resultados negativos.

```
Confusion Matrix

[[ 440   20  137   60   15]
 [ 139  441   14   38   24]
 [   2    1 1280   15    0]
 [  54   54    4  454    7]
 [  18  205    1    0  357]]

Classification Report

              precision    recall  f1-score   support

      Botnet       0.67      0.65      0.66       672
  Bruteforce       0.61      0.67      0.64       656
         DoS       0.89      0.99      0.94      1298
      Normal       0.80      0.79      0.80       573
        Scan       0.89      0.61      0.73       581

    accuracy                           0.79      3780
   macro avg       0.77      0.74      0.75      3780
weighted avg       0.79      0.79      0.78      3780
```

Figura 4.14 – Primeiro treinamento do cliente 1 no 1D-CNN.

```
Confusion Matrix

[[ 646   25    0    1    0]
 [  85  523    0   39    9]
 [   2    0 1280   16    0]
 [  58   56    0  457    2]
 [  12  208    0    0  361]]

Classification Report

              precision    recall  f1-score   support

      Botnet       0.80      0.96      0.88       672
  Bruteforce       0.64      0.80      0.71       656
         DoS       1.00      0.99      0.99      1298
      Normal       0.89      0.80      0.84       573
        Scan       0.97      0.62      0.76       581

    accuracy                           0.86      3780
   macro avg       0.86      0.83      0.84      3780
weighted avg       0.88      0.86      0.86      3780
```

Figura 4.15 – Último treinamento do cliente 1 no 1D-CNN.

A tabela 4.6 mostra o número de TP, FP, TN e FN do treinamento do cliente 1 com o 1D-CNN. O número de TP aumentou para a Botnet e o DoS, além de possuir menos FP para o caso do Bruteforce e para a classe Normal. Porém, o Bruteforce ainda possui um número considerável de falsos positivos, assim como o Scan possui um grande número de falsos negativos. Porém, é perceptível a melhoria dos resultados na totalidade quando é feita a comparação com o modelo comum de Redes Neurais, principalmente pelo aumento de dados verdadeiros e diminuição de dados falsos.

Após a análise dos resultados provenientes do cliente 1, serão observadas as saídas do cliente 2 nas mesmas condições. Pelas Figuras 4.16 e 4.17, é observado o início e o fim do treinamento, mostrando que houve uma evolução na classe de Botnet, DoS e Scan, de tal forma que o Bruteforce e o Normal se mantiveram constantes, tendo apenas evoluções quanto ao *Recall* e ao *F1-Score*. Porém, possível observar que a média com pesos evoluiu de 77 para 84 por cento, indicando uma

Tabela 4.6 – Análise da confusão de matriz final do cliente 1 no 1D-CNN.

Classe	TP	FP	TN	FN
Botnet	646	157	2951	26
Bruteforce	523	289	3028	124
DoS	1280	0	2500	18
Normal	457	56	3151	120
Scan	361	11	3188	220

melhoria considerável. O *F1-Score* de todas as categorias, estando acima de 70 por cento, indica que o balanceamento entre Precisão e *Recall* foi satisfatório para os pré-requisitos do projeto.

```
Confusion Matrix

[[ 510   29   43   73   23]
 [ 118  431   36   49    5]
 [  42    2 1240    0    0]
 [  41   50  111  383    3]
 [  38  180    1    0  372]]

Classification Report

              precision    recall  f1-score   support

      Botnet       0.68      0.75      0.71       678
  Bruteforce       0.62      0.67      0.65       639
         DoS       0.87      0.97      0.91      1284
      Normal       0.76      0.65      0.70       588
        Scan       0.92      0.63      0.75       591

    accuracy                           0.78      3780
   macro avg       0.77      0.73      0.75      3780
weighted avg       0.78      0.78      0.77      3780
```

Figura 4.16 – Primeiro treinamento do cliente 2 no modelo comum.

```
Confusion Matrix

[[ 509   59    3   99    8]
 [  70  513    2   47    7]
 [  11    1 1265    7    0]
 [  33   60    0  492    3]
 [  17  184    0    0  390]]

Classification Report

              precision    recall  f1-score   support

      Botnet       0.80      0.75      0.77       678
  Bruteforce       0.63      0.80      0.70       639
         DoS       1.00      0.99      0.99      1284
      Normal       0.76      0.84      0.80       588
        Scan       0.96      0.66      0.78       591

    accuracy                           0.84      3780
   macro avg       0.83      0.81      0.81      3780
weighted avg       0.86      0.84      0.84      3780
```

Figura 4.17 – Último treinamento do cliente 2 no modelo comum.

A Tabela 4.7 indica o número de TP, FP, TN e FN para o cliente 2 no modelo comum, apresentando resultados semelhantes aos do cliente 1. As classes tiveram uma boa distribuição de

verdadeiros e falsos, mantendo a proporção do Bruteforce e Scan quanto ao cliente 1, demonstrando uma similaridade na forma de treinamento, porém com proporções ainda diferenciáveis. O Botnet e o Normal foram equilibrados, mostrando a boa divisão entre o tráfego malicioso e benigno que circulou no ambiente de testes.

Tabela 4.7 – Análise da confusão de matriz final do cliente 2 no modelo comum.

Classe	TP	FP	TN	FN
Botnet	509	131	2971	169
Bruteforce	513	203	2897	126
DoS	1265	5	2510	19
Normal	492	153	3132	96
Scan	390	18	3372	201

E por último, nas Figuras 4.18 e 4.19, estão o primeiro e último treinamento do cliente 2 com 1D-CNN. É possível observar que o resultado apresenta melhores acurácias em comparação ao modelo comum, tendo boas saídas de *F1-Score* em todas as classes, indicando boas proporções de Precisão e *Recall*. A média com pesos foi de 86 por cento se comparado ao do modelo comum de 84 por cento, demonstrando uma vantagem em termos de desempenho. O *Recall* do Scan não foi tão alto como no caso do cliente 1, porém conseguiu atingir 60 por cento, indicando uma taxa satisfatória para a avaliação do modelo neste projeto.

```
Confusion Matrix

[[ 432   36  175   30    5]
 [ 124  416   13   55   31]
 [   2    1 1271   10    0]
 [  45   49    9  485    0]
 [  14  241    0    0  336]]

Classification Report

              precision    recall  f1-score   support

      Botnet       0.70      0.64      0.67       678
  Bruteforce       0.56      0.65      0.60       639
         DoS       0.87      0.99      0.92      1284
      Normal       0.84      0.82      0.83       588
        Scan       0.90      0.57      0.70       591

    accuracy                           0.78      3780
   macro avg       0.77      0.73      0.74      3780
weighted avg       0.79      0.78      0.77      3780
```

Figura 4.18 – Primeiro treinamento do cliente 2 no 1D-CNN.

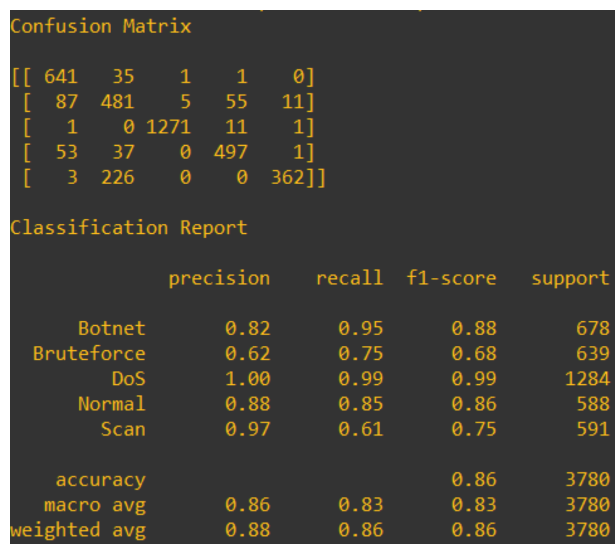

Figura 4.19 – Último treinamento do cliente 2 no 1D-CNN.

A Tabela 4.8 indica os últimos dados de verdadeiros e falsos positivos e negativos. É possível observar que a Botnet apresenta um bom equilíbrio entre os verdadeiros e falsos positivos, tendo uma baixa quantidade de falsos negativos. O Bruteforce ainda contém um alto número de falsos positivos, ainda conseguindo manter uma taxa de Precisão maior que 60 por cento. O DoS teve um desempenho excelente, tendo a melhor taxa de identificação em todos os treinamentos realizados. A categoria Normal apresentou uma grande redução de falsos positivos e negativos, atingindo uma alta taxa de acurácia e precisão. Por último, o Scan teve resultados bem semelhantes com o do cliente 1, tendo uma alta quantidade de falsos negativos, fazendo com que o *Recall* apresente uma taxa de 60 por cento. Então, foi validado que o modelo apresentado mostra uma taxa satisfatória de acurácia e precisão em termos de detecção para todas as classes, apenas apresentando taxas menores quanto às categorias de Bruteforce e Scan pelo fato de serem categorias com dados de fluxo próximos uns dos outros, facilitando a obtenção de classificações falsas.

Tabela 4.8 – Análise da confusão de matriz final do cliente 2 no 1D-CNN.

Classe	TP	FP	TN	FN
Botnet	641	144	2958	37
Bruteforce	481	298	3168	158
DoS	1271	6	2503	13
Normal	497	91	3216	67
Scan	362	13	3405	229

Antes da visualização da distribuição final das métricas de desempenho de todas as categorias utilizadas para realizar o treinamento do modelo geral construído neste projeto, é montada uma relação de todas as médias do *F1-Score*, incluindo a média com pesos, médias macro e acurácia dos treinamentos finais dos dois clientes. A média com pesos do *F1-Score* é um dos parâmetros mais importantes para se avaliar as proporções de Precisão e *Recall* do balanceamento de todas as classes, indicando a taxa de qualidade do modelo apresentado em termos de equilíbrios de dados e obtenção de valores verdadeiros seguindo o nível de influência de cada classe durante o aprendizado.

A média macro sem a presença de pesos também será observada assim como a acurácia padrão, mesmo que não sejam tão impactantes na análise final. A Tabela 4.9 indica o resultado das médias para o modelo de Redes Neurais simples na última rodada e a Tabela 4.10 indica o resultado das médias apontadas para o modelo 1D-CNN também na última rodada.

Tabela 4.9 – Relação final das médias do *F1-Score* para o modelo comum.

Cliente	Acurácia	Média Macro	Média com pesos
1	0.83	0.80	0.83
2	0.84	0.81	0.84

Tabela 4.10 – Relação final das médias do *F1-Score* para o 1D-CN.

Cliente	Acurácia	Média Macro	Média com pesos
1	0.86	0.84	0.86
2	0.86	0.83	0.86

É possível observar pelas Tabelas anteriores que os dois clientes tiveram desempenhos semelhantes e que o 1D-CNN apresentou melhores resultados em todos os quesitos quando comparado com o modelo comum. Portanto, é validado que o modelo 1D-CNN [33] apresenta resultados de alta eficiência para a arquitetura de IDS com *Federated Learning*, revelando a importância da utilização de *Deep Learning* com Redes Neurais para ambientes de detecção de atividades benignas ou maliciosas por aprendizado de máquina.

Além disso, apesar da diversidade de técnicas e de fluxo de funcionamento que o fazem ter menos desempenho que uma solução de Machine Learning comum, o *Federated Learning* apresentou resultados satisfatórios de desempenho de detecção para as categorias utilizadas, principalmente pelo fato de se utilizar de outros dispositivos com *datasets* distintos para realizar o treinamento. Essa descentralização do treinamento apresenta benefícios de privacidade e segurança, pois os clientes não compartilham informações entre si e estão conectados com o servidor de forma segura por métodos como o SSL.

Cada um dos clientes Raspberry Pi realizou o treino com alta performance sem problemas de utilização de memória e custo computacional, soluções viáveis para se realizar um treinamento de classificação distribuído em uma arquitetura de IDS. Para melhorar o desempenho do modelo e tornar a arquitetura mais viável para aplicações em ambientes com ataques não controlados, é importante considerar a escalabilidade da rede e aplicar mais métodos de privacidade como o *Differential Privacy*, além de métodos para reduzir o custo computacional e custo de comunicação como *Cloud* e *Fog Computing*.

4.2.3 Gráficos de Desempenho

Por último, para concatenar todos os resultados anteriores, são feitos 4 gráficos em forma de histogramas montando um comparativo das taxas para cada métrica anteriormente avaliada nos dois clientes. As Figuras 4.20 e 4.21 indicam o gráfico referente ao cliente 1 no modelo normal e no CNN, mostrando a clara evolução de praticamente todas as classes ao serem utilizadas as camadas

convolucionais para realizar a filtragem das características e o treinamento por Redes Neurais de camadas totalmente conectadas.

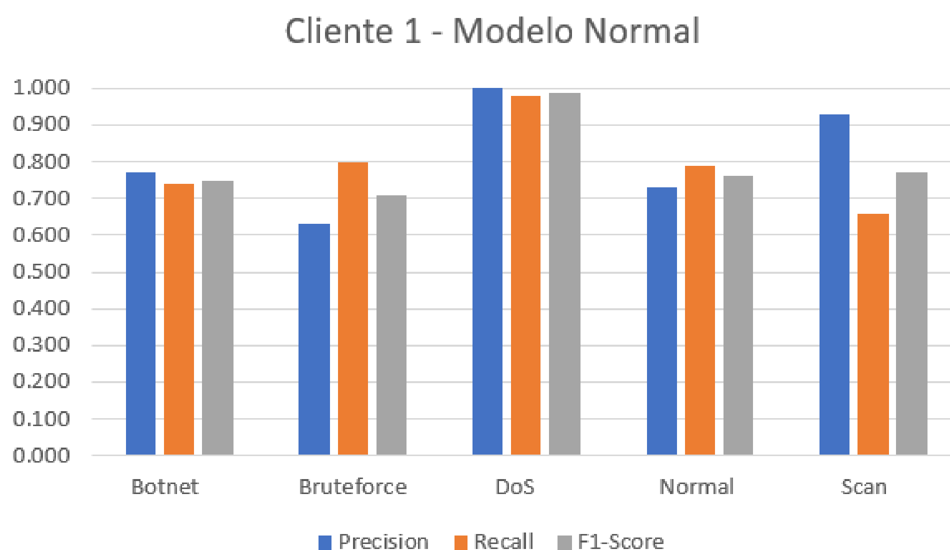

Figura 4.20 – Gráfico de desempenho de cada classe no modelo comum do cliente 1.

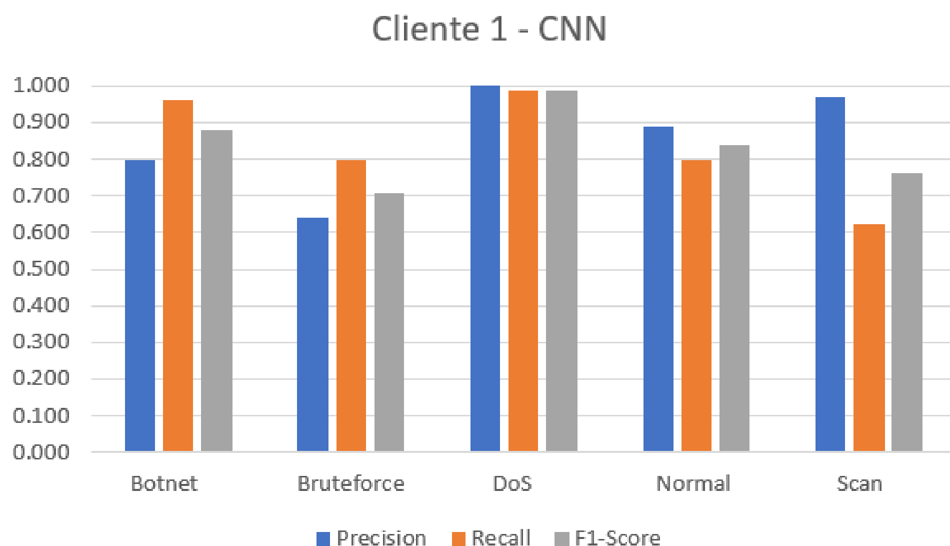

Figura 4.21 – Gráfico de desempenho de cada classe no 1D-CNN do cliente 1.

As Figuras 4.22 e 4.23 indicam os mesmos histogramas feitos para o cliente 2 no modelo normal e CNN. É possível observar que os resultados em geral foram próximos do cliente 1, porém com pequenas diferenças de taxas e distribuições devido à diferença de divisão dos conjuntos de treinamento e testes para indicar a separação dos treinamentos. Assim como no cliente 1, o modelo CNN indicou melhores resultados que o modelo normal, apresentando então uma maior eficiência para se realizar o treinamento de um *dataset* relacionado a tráfegos de uma rede local com a presença de uma Botnet aplicando ataque de negação de serviço distribuído.

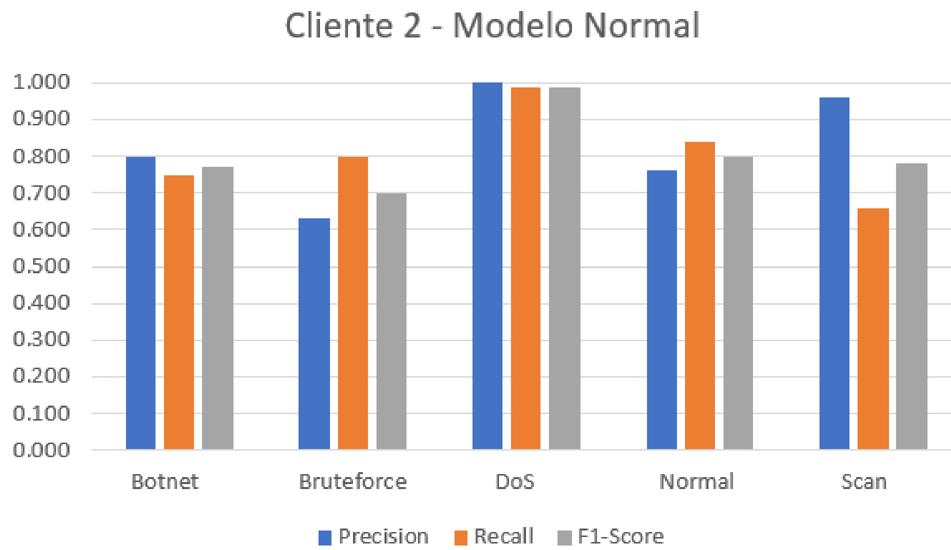

Figura 4.22 – Gráfico de desempenho de cada classe no modelo comum do cliente 2.

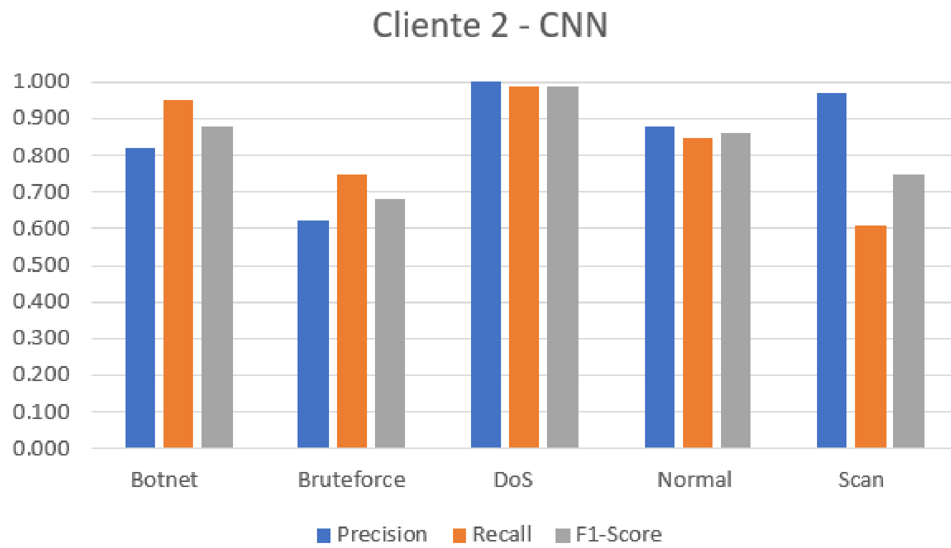

Figura 4.23 – Gráfico de desempenho de cada classe no 1D-CNN do cliente 2.

5 Conclusões e Trabalhos Futuros.

O objetivo deste trabalho foi de implementar uma arquitetura de detecção de intrusão em redes de Internet das Coisas por identificação de anomalias na rede a partir do tráfego capturado em uma infraestrutura local. O projeto utilizou-se de técnicas baseadas em conceitos referentes a redes IoT, detecção de intrusão, aprendizado de máquina distribuído por Redes Neurais Convolucionais.

Pelo fato de redes de Internet das Coisas apresentarem altos índices de vulnerabilidade devido ao motivo de que são utilizados dispositivos de microarquitetura simples, foi desenvolvido um NIDS (*Network Intrusion Detection System*) para se observar o comportamento do tráfego ao longo de toda a estrutura desde o *backbone* até os *endpoints*. As técnicas de aprendizado de máquina aplicadas, sendo utilizado especificamente o *Federated Learning* a partir dos dispositivos IoT, serviram para avaliar o padrão da comunicação referente a cada tipo de dado gerado durante o experimento com o objetivo de classificá-los conforme a sua natureza.

Na arquitetura proposta, foi utilizada uma *botnet* real chamada Mirai para atacar um alvo por DoS enquanto ocorre a captura deste tráfego por uma estrutura contendo o *software* Suricata, Apache Kafka e Apache Spark para fazer seu processamento em tempo real. Com o processamento finalizado, o modelo resultante é utilizado em uma infraestrutura de aprendizado federado contendo dispositivos do tipo Raspberry Pi na seção de clientes para realizar o seu respectivo treinamento. Posteriormente, com o fim do treinamento, é observado o nível de desempenho de previsão e detecção de novos dados a partir do reconhecimento dos padrões de cada pacote analisado.

Para realizar os testes e obter os resultados da validação da arquitetura, foram feitos dois tipos de treinamentos utilizando dois modelos distintos: um modelo de Redes Neurais simples e um modelo por Redes Neurais Convolucionais de uma dimensão (1D-CNN). Os treinamentos foram feitos por diversas rodadas, sendo obtidos os resultados do processo de agregação dos modelos a partir dos pesos de cada cliente. O 1D-CNN apresentou um melhor desempenho que o modelo comum, de tal forma que cada categoria utilizada no processo de classificação apresentou altas taxas de precisão de acerto durante os testes, comprovando a eficácia do modelo proposto.

Desta forma, o trabalho conseguiu obter resultados satisfatórios para cumprir o seu objetivo de detectar os diferentes padrões de comportamento de pacotes durante o tráfego em uma rede IoT com a presença de um *malware*. Cada categoria foi identificada de forma consideravelmente eficaz, apresentando resultados plausíveis de detecção de anomalias dentro de suas limitações.

Pelo fato do aprendizado federado ainda apresentar dificuldades quanto ao seu desempenho de treinamento para gerar modelos mais precisos, é possível realizar trabalhos futuros em cima dos resultados obtidos neste projeto visando escalar a estrutura de aprendizado, incluir mais métodos de segurança como a *Blockchain* e métodos de privacidade como o *Differential Privacy*. Além disso, uma sugestão de trabalho futuro é de evoluir o trabalho para agir como uma arquitetura de detecção e prevenção de intrusão, utilizando os resultados do aprendizado de máquina para realizar ações preventivas durante a identificação de um possível ataque.

Bibliografia

[1] Sajad Ahmadian e Alireza Khanteymoori. “Training back propagation neural networks using asexual reproduction optimization”. Em: mai. de 2015. DOI: <10.1109/IKT.2015.7288738>.

[2] Umar Ahsan e Abdul Bais. “A Review on Big Data Analysis and Internet of Things”. Em: out. de 2016, pp. 325–330. DOI: <10.1109/MASS.2016.048>.

[3] Ulrich Matchi Aïvodji, Sébastien Gambs e Alexandre Martin. “IOTFLA : A Secured and Privacy-Preserving Smart Home Architecture Implementing Federated Learning”. Em: *2019 IEEE Security and Privacy Workshops (SPW)*. 2019, pp. 175–180. DOI: <10.1109/SPW.2019.00041>.

[4] *Apache Kafka*. URL: <https://kafka.apache.org/> (acesso em 10/08/2022).

[5] *Apache Kafka Cluster*. URL: <https://blog.geekhunter.com.br/apache-kafka/> (acesso em 11/08/2022).

[6] *Apache Maven*. URL: <https://maven.apache.org/> (acesso em 11/08/2022).

[7] *Apache Spark*. URL: <https://spark.apache.org/> (acesso em 11/08/2022).

[8] Muhammad Asad, Ahmed Moustafa e Dr-Muhammad Aslam. “CEEP-FL: A comprehensive approach for communication efficiency and enhanced privacy in federated learning”. Em: *Applied Soft Computing* (fev. de 2021). DOI: <10.1016/j.asoc.2021.107235>.

[9] Parvaneh Asghari, Amir Rahmani e Hamid Haj Seyyed Javadi. “Internet of Things applications: A Systematic Review”. Em: *Computer Networks* 148 (dez. de 2018). DOI: <10.1016/j.comnet.2018.12.008>.

[10] Lirim Ashiku e Cihan Dagli. “Network Intrusion Detection System using Deep Learning”. Em: *Procedia Computer Science* 185 (2021). Big Data, IoT, and AI for a Smarter Future, pp. 239–247. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2021.05.025>.

[11] Keith Bonawitz et al. “Towards Federated Learning at Scale: System Design”. Em: *Proceedings of Machine Learning and Systems*. Ed. por A. Talwalkar, V. Smith e M. Zaharia. Vol. 1. 2019, pp. 374–388. URL: <https://proceedings.mlsys.org/paper/2019/file/bd686fd640be98efaae0091fa301e613-Paper.pdf>.

[12] Ayan Chatterjee e Bestoun Ahmed. “IoT Anomaly Detection Methods and Applications: A Survey”. Em: *Internet of Things* (jun. de 2022).

[13] Sayan Chatterjee e Manjesh Hanawal. “Federated Learning for Intrusion Detection in IoT Security: A Hybrid Ensemble Approach”. Em: (jun. de 2021).

[14] Nitesh Chawla et al. “SMOTE: Synthetic Minority Over-sampling Technique”. Em: *J. Artif. Intell. Res. (JAIR)* 16 (jun. de 2002), pp. 321–357. DOI: <10.1613/jair.953>.

[15] *Cosmic Mirai*. URL: <https://github.com/hoaan1995/Cosmic-Mirai> (acesso em 03/08/2022).

[16] Priyanka Dahiya e Devesh Kumar Srivastava. “Network Intrusion Detection in Big Dataset Using Spark”. Em: *Procedia Computer Science* 132 (2018). International Conference on Computational Intelligence and Data Science, pp. 253–262. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2018.05.169>.

[17] *Emergint Threats.* URL: <https://rules.emergingthreats.net/> (acesso em 01/08/2022).

[18] *Eve JSON.* URL: <https://suricata.readthedocs.io/en/suricata-6.0.0/output/eve/eve-json-output.html> (acesso em 07/08/2022).

[19] Ilhem Fajjari, Fouad Tobagi e Yutaka Takahashi. “Cloud edge computing in the IoT”. Em: *Annals of Telecommunications* 73 (ago. de 2018), pp. 413–414. DOI: <10.1007/s12243-018-0651-6>.

[20] Awatef Ali Yousef R Fares et al. “DoS Attack Prevention on IPS SDN Networks”. Em: *2019 Workshop on Communication Networks and Power Systems (WCNPS)*. IEEE. 2019, pp. 1–7.

[21] Roman Fekolkin. “Intrusion Detection and Prevention Systems: Overview of Snort and Suricata”. Em: (jan. de 2015).

[22] Andrey Ferriyan et al. “Generating Network Intrusion Detection Dataset Based on Real and Encrypted Synthetic Attack Traffic”. Em: *Applied Sciences* 11.17 (2021). ISSN: 2076-3417. DOI: <10.3390/app11177868>. URL: <https://www.mdpi.com/2076-3417/11/17/7868>.

[23] *Flower Framework.* URL: <https://flower.dev/> (acesso em 20/08/2022).

[24] Konstantina Fotiadou et al. “Network Traffic Anomaly Detection via Deep Learning”. Em: *Information* 12.5 (2021). ISSN: 2078-2489. DOI: <10.3390/info12050215>.

[25] Anmin Fu et al. “VFL: A Verifiable Federated Learning With Privacy-Preserving for Big Data in Industrial IoT”. Em: *IEEE Transactions on Industrial Informatics* 18.5 (2022), pp. 3316–3326. DOI: <10.1109/TII.2020.3036166>.

[26] Daniel G. V. Gonçalves et al. “IPS architecture for IoT networks overlapped in SDN”. Em: *2019 Workshop on Communication Networks and Power Systems (WCNPS)*. 2019, pp. 1–6. DOI: <10.1109/WCNPS.2019.8896297>.

[27] Enzo Grossi e Massimo Buscema. “Introduction to artificial neural networks”. Em: *European journal of gastroenterology hepatology* 19 (jan. de 2008), pp. 1046–54. DOI: <10.1097/MEG.0b013e3282f198a0>.

[28] Muhammad Habib ur Rehman et al. “Towards Blockchain-Based Reputation-Aware Federated Learning”. Em: fev. de 2020. DOI: <10.1109/INFOCOMWKSHPS50562.2020.9163027>.

[29] Cao Hui et al. “IFed: A novel federated learning framework for local differential privacy in Power Internet of Things”. Em: *International Journal of Distributed Sensor Networks* 16 (mai. de 2020), p. 155014772091969. DOI: <10.1177/1550147720919698>.

[30] “IoT Botnet Forensics: A Comprehensive Digital Forensic Case Study on Mirai Botnet Servers”. Em: *Forensic Science International: Digital Investigation* 32 (2020), p. 300926. ISSN: 2666-2817. DOI: <https://doi.org/10.1016/j.fsidi.2020.300926>.

[31] Guilherme de O Kfouri et al. “Design of a Distributed HIDS for IoT Backbone Components”. Em: *FedCSIS (Communication Papers)*. Leipzig, Germany, 2019, pp. 81–88. DOI: <10.15439/2019F329>.

[32] Haesik Kim. “Machine Learning”. Em: *Design and Optimization for 5G Wireless Communications*. 2020, pp. 151–193. DOI: <10.1002/9781119494492.ch5>.

[33] Serkan Kiranyaz et al. “1-D Convolutional Neural Networks for Signal Processing Applications”. Em: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 8360–8364. DOI: <10.1109/ICASSP.2019.8682194>.

[34] Sachin Kumar, Prayag Tiwari e Mikhail Zymbler. “Internet of Things is a revolutionary approach for future technology enhancement: a review”. Em: *Journal of Big Data* 6 (dez. de 2019). DOI: <10.1186/s40537-019-0268-2>.

[35] Francis Quintal Lauzon. “An introduction to deep learning”. Em: *2012 11th International Conference on Information Science, Signal Processing and their Applications (ISSPA)*. 2012, pp. 1438–1439. DOI: <10.1109/ISSPA.2012.6310529>.

[36] Jianhua Li et al. “FLEAM: A Federated Learning Empowered Architecture to Mitigate DDoS in Industrial IoT”. Em: *IEEE Transactions on Industrial Informatics* 18.6 (2022), pp. 4059–4068. DOI: <10.1109/TII.2021.3088938>.

[37] Tian Li et al. “Federated Learning: Challenges, Methods, and Future Directions”. Em: *IEEE Signal Processing Magazine* 37.3 (2020), pp. 50–60. DOI: <10.1109/MSP.2020.2975749>.

[38] Hung-Jen Liao et al. “Intrusion detection system: A comprehensive review”. Em: *Journal of Network and Computer Applications* 36 (jan. de 2013), pp. 16–24. DOI: <10.1016/j.jnca.2012.09.004>.

[39] Umer Majeed e Choong Seon Hong. “FLchain: Federated Learning via MEC-enabled Blockchain Network”. Em: *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 2019, pp. 1–4. DOI: <10.23919/APNOMS.2019.8892848>.

[40] Aaisha Makkar et al. “FedLearnSP: Preserving Privacy and Security Using Federated Learning and Edge Computing”. Em: *IEEE Consumer Electronics Magazine* 11.2 (2022), pp. 21–27. DOI: <10.1109/MCE.2020.3048926>.

[41] Dapeng Man et al. “Intelligent Intrusion Detection Based on Federated Learning for Edge-Assisted Internet of Things”. Em: *Security and Communication Networks* 2021 (out. de 2021), pp. 1–11. DOI: <10.1155/2021/9361348>.

[42] Joel Margolis et al. “An In-Depth Analysis of the Mirai Botnet”. Em: *2017 International Conference on Software Security and Assurance (ICSSA)*. 2017, pp. 6–12. DOI: <10.1109/ICSSA.2017.12>.

[43] H. McMahan et al. “Federated Learning of Deep Networks using Model Averaging”. Em: (fev. de 2016).

[44] Arpit Sunilkumar Modi. “Review Article on Deep Learning Approaches”. Em: *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*. 2018, pp. 1635–1639. DOI: <10.1109/ICCONS.2018.8663057>.

[45] Viraaji Mothukuri et al. "Federated-Learning-Based Anomaly Detection for IoT Security Attacks". Em: *IEEE Internet of Things Journal* 9.4 (2022), pp. 2545–2554. DOI: <10.1109/JIOT.2021.3077803>.

[46] Thien Duc Nguyen et al. "DÏoT: A Federated Self-learning Anomaly Detection System for IoT". Em: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 756–767. DOI: <10.1109/ICDCS.2019.00080>.

[47] Thien Duc Nguyen et al. "Poisoning Attacks on Federated Learning-based IoT Intrusion Detection System". Em: 2020.

[48] Oscar Novo. "Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT". Em: *IEEE Internet of Things Journal* 5.2 (2018), pp. 1184–1195. DOI: <10.1109/JIOT.2018.2812239>.

[49] Keiron O'Shea e Ryan Nash. "An Introduction to Convolutional Neural Networks". Em: *ArXiv e-prints* (nov. de 2015).

[50] Mohamed Ouhssini et al. "Distributed intrusion detection system in the cloud environment based on Apache Kafka and Apache Spark". Em: out. de 2021, pp. 1–6. DOI: <10.1109/ICDS53782.2021.9626721>.

[51] Animesh Patcha e Jung-Min Park. "An overview of anomaly detection techniques: Existing solutions and latest technological trends". Em: *Computer Networks* 51.12 (2007), pp. 3448–3470. ISSN: 1389-1286. DOI: <https://doi.org/10.1016/j.comnet.2007.02.001>.

[52] Marcelo Alves Prado. "Análise experimental da botnet IoT Mirai". Em: (2018).

[53] Nurul Ashikin Samat, Mohd Salleh e Haseeb Ali. "The Comparison of Pooling Functions in Convolutional Neural Network for Sentiment Analysis Task". Em: jan. de 2020, pp. 202–210. ISBN: 978-3-030-36055-9. DOI: <10.1007/978-3-030-36056-6_20>.

[54] Momina Shaheen et al. "Applications of Federated Learning; Taxonomy, Challenges, and Research Trends". Em: *Electronics* 11.4 (2022). DOI: <10.3390/electronics11040670>.

[55] *Structured Streaming Kafka Integration.* URL: <https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html> (acesso em 11/08/2022).

[56] *Suricata IDS/IPS.* URL: <https://suricata.io//> (acesso em 07/08/2022).

[57] Hein Tun et al. "Selection the perimeter protection equipment in security systems". Em: jan. de 2018, pp. 1504–1508. DOI: <10.1109/EIConRus.2018.8317383>.

[58] *vnStat.* URL: <https://humdi.net/vnstat/> (acesso em 02/09/2022).

[59] Zhen Wang e Dan Zhang. "HIDS and NIDS Hybrid Intrusion Detection System Model Design". Em: *Advanced Engineering Forum* 6-7 (set. de 2012), pp. 991–994. DOI: <10.4028/www.scientific.net/AEF.6-7.991>.

[60] Kang Wei et al. "Federated Learning With Differential Privacy: Algorithms and Performance Analysis". Em: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 3454–3469. DOI: <10.1109/TIFS.2020.2988575>.

[61] Bin Xia et al. “PETs: A Stable and Accurate redictor of Protein-Protein Interacting Sites Based on xtremely-Randomized ree”. Em: *IEEE Transactions on NanoBioscience* 14 (nov. de 2015), pp. 1–1. DOI: <10.1109/TNB.2015.2491303>.

[62] Tuo Zhang et al. “Federated Learning for Internet of Things: A Federated Learning Framework for On-device Anomaly Data Detection”. Em: *CoRR* abs/2106.07976 (2021). URL: <https://arxiv.org/abs/2106.07976>.

[63] Xinyou Zhang, Chengzhong Li e Wenbin Zheng. “Intrusion prevention system design”. Em: *The Fourth International Conference onComputer and Information Technology, 2004. CIT '04*. 2004, pp. 386–390. DOI: <10.1109/CIT.2004.1357226>.

[64] Ruijie Zhao et al. “Intelligent intrusion detection based on federated learning aided long short-term memory”. Em: *Physical Communication* 42 (jun. de 2020), p. 101157. DOI: <10.1016/j.phycom.2020.101157>.

[65] Chunyi Zhou et al. “Privacy-Preserving Federated Learning in Fog Computing”. Em: *IEEE Internet of Things Journal* 7.11 (2020), pp. 10782–10793. DOI: <10.1109/JIOT.2020.2987958>.

ANEXO A – Scala Code

```
package com.sparkbyexamples.spark.streaming.kafka.json

import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{col, from_json}
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types.{IntegerType, StringType, StructType}

object SparkStreamingConsumerKafkaJson {

  def main(args: Array[String]): Unit = {

    val spark: SparkSession = SparkSession.builder()
      .master("local[3]")
      .appName("SparkByExample")
      .getOrCreate()

    spark.sparkContext.setLogLevel("ERROR")

    val df = spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "localhost:9092")
        .option("subscribe", "test")
        .option("failOnDataLoss", "false")
        .option("startingOffsets", "earliest") // From starting
        .load()

    val schema = new StructType()
      .add("timestamp",StringType)
      .add("flow_id",StringType)
      .add("src_ip",StringType)
      .add("dest_ip",StringType)
      .add("src_port",IntegerType)
      .add("dest_port",IntegerType)
      .add("proto",StringType)
      .add("alert",
         new StructType()
           .add("severity",IntegerType)
```

```
          )
        .add("flow",
           new StructType()
             .add("pkts_toserver",IntegerType)
             .add("pkts_toclient",IntegerType)
             .add("bytes_toserver",IntegerType)
             .add("bytes_toclient",IntegerType)
             )

    val person = df.selectExpr("CAST(value AS STRING)")
    .select(from_json(col("value"), schema).as("data"))
      .select("data.*")
      .withColumn("hour",col("timestamp").substr(12,2))
      .withColumn("minute",col("timestamp").substr(15,2))
      .withColumn("seconds",col("timestamp").substr(18,2))
      .withColumn("severity",col("alert").getItem("severity"))
      .withColumn("pkts_toserver",col("flow").getItem("pkts_toserver"))
      .withColumn("pkts_toclient",col("flow").getItem("pkts_toclient"))
      .withColumn("bytes_toserver",col("flow").getItem("bytes_toserver"))
      .withColumn("bytes_toclient",col("flow").getItem("bytes_toclient"))
      .drop("flow").drop("alert").drop("timestamp")

    person.printSchema()

    val query = person
      .writeStream
      .format("csv")
      .option("format", "append")
      .option("csv.block.size", 1024)
      .option("path","/tmp/datasetcsv/")
      .option("checkpointLocation", "/checkpointad")
      .outputMode("append")
      .start()
      .awaitTermination()
  }
}
```

ANEXO B – POM File

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>spark</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>${project.artifactId}</name>
  <description>My wonderfull scala app</description>
  <inceptionYear>2018</inceptionYear>
  <licenses>
    <license>
      <name>My License</name>
      <url>http://....</url>
      <distribution>repo</distribution>
    </license>
  </licenses>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <encoding>UTF-8</encoding>
    <scala.version>2.12.15</scala.version>
    <spark.version>3.3.0</spark.version>
    <scala.compat.version>2.12</scala.compat.version>
    <spec2.version>4.2.0</spec2.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.scala-lang</groupId>
      <artifactId>scala-library</artifactId>
      <version>${scala.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.12</artifactId>
```

```
      <version>${spark.version}</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql-kafka-0-10_2.12</artifactId>
      <version>${spark.version}</version>
    </dependency>

    <!-- Test -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.scalatest</groupId>
      <artifactId>scalatest_${scala.compat.version}</artifactId>
      <version>3.0.5</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.specs2</groupId>
      <artifactId>specs2-core_${scala.compat.version}</artifactId>
      <version>${spec2.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.specs2</groupId>
      <artifactId>specs2-junit_${scala.compat.version}</artifactId>
      <version>${spec2.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <sourceDirectory>src/main/scala</sourceDirectory>
    <testSourceDirectory>src/test/scala</testSourceDirectory>
    <plugins>
      <plugin>
        <!-- see http://davidb.github.com/scala-maven-plugin -->
```

```
    <groupId>net.alchim31.maven</groupId>
    <artifactId>scala-maven-plugin</artifactId>
    <version>3.3.2</version>
    <executions>
      <execution>
        <goals>
          <goal>compile</goal>
          <goal>testCompile</goal>
        </goals>
        <configuration>
          <args>
            <arg>-dependencyfile</arg>
            <arg>${project.build.directory}/.scala_dependencies</arg>
          </args>
        </configuration>
      </execution>
    </executions>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.21.0</version>
    <configuration>
      <!-- Tests will be run with scalatest-maven-plugin instead -->
      <skipTests>true</skipTests>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.scalatest</groupId>
    <artifactId>scalatest-maven-plugin</artifactId>
    <version>2.0.0</version>
    <configuration>
      <reportsDirectory>${project.build.directory}
      /surefire-reports</reportsDirectory>
      <junitxml>.</junitxml>
      <filereports>TestSuiteReport.txt</filereports>
      <!-- Comma separated list of JUnit test class names to execute -->
      <jUnitClasses>samples.AppTest</jUnitClasses>
    </configuration>
    <executions>
      <execution>
        <id>test</id>
```

```
                        <goals>
                            <goal>test</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

ANEXO C – Client Code

```
import argparse
import os
from pathlib import Path

from tensorflow import keras
import tensorflow as tf
import flwr as fl
import pandas as pd
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Flatten, Dense, Conv1D, MaxPool1D, Dropout,
Input, Activation
from sklearn import preprocessing
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import SMOTE
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
from imblearn.pipeline import Pipeline
from sklearn.utils import class_weight
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
# Make TensorFlow logs less verbose
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"


# Define Flower client
class CifarClient(fl.client.NumPyClient):
    def __init__(self, model, x_train, y_train, x_test, y_test):
        self.model = model
        self.x_train, self.y_train = x_train, y_train
        self.x_test, self.y_test = x_test, y_test
        self.y_pred = 0
        self.y_pred_bool = 0
        self.y_pred_b = 0
        self.visualizer = 0
    def get_properties(self, config):
```

```
        """Get properties of client."""
        raise Exception("Not implemented")

    def get_parameters(self, config):
        """Get parameters of the local model."""
        raise Exception("Not implemented
        (server-side parameter initialization)")

    def fit(self, parameters, config):
        """Train parameters on the locally held training set."""

        # Update local model parameters
        self.model.set_weights(parameters)

        # Get hyperparameters for this round
        batch_size: int = config["batch_size"]
        epochs: int = config["local_epochs"]
        self.class_weight = {i : self.class_weight[i] for i in range(5)}
       # Train the model using hyperparameters from config
        history = self.model.fit(
            self.x_train,
            self.y_train,
            batch_size,
            epochs,
            verbose=2,
            validation_split=0.3,
        )

        self.y_pred = self.model.predict(self.x_test,
        batch_size=32, verbose=2)
        self.y_pred_bool = np.argmax(self.y_pred, axis=-1)
        confusion = confusion_matrix(self.y_test, self.y_pred_bool)
        print('Confusion Matrix\n')
        print(confusion)

        print('\nClassification Report\n')
        print(classification_report(self.y_test,
        self.y_pred_bool,
        target_names =
        ['Botnet', 'Bruteforce', 'DoS', 'Normal', 'Scan']))

        # Return updated model parameters and results
```

```
        parameters_prime = self.model.get_weights()
        num_examples_train = len(self.x_train)
        results = {
            "loss": history.history["loss"][0],
            "accuracy": history.history["accuracy"][0],
            "val_loss": history.history["val_loss"][0],
            "val_accuracy": history.history["val_accuracy"][0],
        }
        return parameters_prime, num_examples_train, results

    def evaluate(self, parameters, config):
        self.model.set_weights(parameters)

        # Get config values
        steps: int = config["val_steps"]

        # Evaluate global model parameters
        on the local test data and return results
        loss, accuracy =
        self.model.evaluate(self.x_test, self.y_test, 32, steps=steps)
        num_examples_test = len(self.x_test)
        return loss, num_examples_test, {"accuracy": accuracy}


def main() -> None:
    # Parse command line argument `partition`
    parser = argparse.ArgumentParser(description="Flower")
    parser.add_argument("--partition",
    type=int, choices=range(0, 10),
    required=True)
    args = parser.parse_args()

    # Modelo comum
#    model = Sequential()
#    model.add(Input(shape=(14,)))
#    model.add(Dense(32,activation='relu'))
#    model.add(Dropout(0.2))
#    model.add(Dense(16,activation='relu'))
#    model.add(Dropout(0.2))
#    model.add(Dense(5,activation='softmax'))

    #1D-CNN
```

```
    model.add(Conv1D(filters=16, kernel_size=(3,),
    activation='relu', input_shape = (14,1)))
    model.add(Conv1D(filters=32, kernel_size=(3,),
    activation='relu', input_shape = (14,1)))
    model.add(MaxPool1D(pool_size=(3,), strides=2, padding='same'))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(32,activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(16,activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(5,activation='softmax'))
    opt = keras.optimizers.Adam(learning_rate=0.0001)
    model.compile(opt, "sparse_categorical_crossentropy", metrics=["accuracy"])

    # Load a subset of CIFAR-10 to simulate the local data partition
    x_train, y_train, x_test, y_test = load_partition()

    # Start Flower client
    client = CifarClient(model, x_train, y_train, x_test, y_test)

    fl.client.start_numpy_client(
        server_address="0.0.0.0:5050",
        client=client,
        root_certificates=Path(".cache/certificates/ca.crt").read_bytes(),
    )


def load_partition():
    out = pd.read_csv("./trainmodel.csv", on_bad_lines='skip')
    out['src_ip'] = out['src_ip'].astype('category')
    out['dest_ip'] = out['dest_ip'].astype('category')
    out['proto'] = out['proto'].astype('category')
    out['class'] = out['class'].astype('category')
    out['src_ip'] = out['src_ip'].cat.codes
    out['dest_ip'] = out['dest_ip'].cat.codes
    out['proto'] = out['proto'].cat.codes
    out['class'] = out['class'].cat.codes
    x = out.drop('class', axis=1)
    y = out.iloc[:,-1].values
    over = SMOTE(sampling_strategy={0: 3400, 1: 3300, 3: 2900, 4: 2800})
    under = RandomUnderSampler(sampling_strategy={2: 6500})
```

```
    pipeline = Pipeline(steps=[('u', under),('o', over)])
    x, y = pipeline.fit_resample(x, y)
    x_train, x_test, y_train, y_test = train_test_split(x,y,
    test_size=0.2,random_state=80)
    counter = Counter(y_train)
    for k,v in counter.items():
            per = v / len(y_train) * 100
            print('Class=%d, n=%d (%.3f%%)' % (k, v, per))
    scaler = preprocessing.StandardScaler()
    x_train=scaler.fit_transform(x_train)
    x_test=scaler.transform(x_test)
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1], 1)

    return x_train, y_train, x_test, y_test

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

ANEXO D – Server Code

```
from typing import Dict, Optional, Tuple
from pathlib import Path

import flwr as fl
import tensorflow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Flatten, Dense, Conv1D, MaxPool1D,
Dropout, Input, Activation
import client

def main() -> None:
#     model = Sequential()
#     model.add(Input(shape=(14,)))
#     model.add(Dense(32,activation='relu'))
#     model.add(Dropout(0.2))
#     model.add(Dense(16,activation='relu'))
#     model.add(Dropout(0.2))
#     model.add(Dense(5,activation='softmax'))

    #1D-CNN
    model = Sequential()
    model.add(Conv1D(filters=16, kernel_size=3,
    activation='relu', input_shape = (14,1)))
    model.add(Conv1ND(filters=32, kernel_size=3,
    activation='relu', input_shape = (14,1)))
    model.add(MaxPool1D(pool_size=(3,), strides=2, padding='same'))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(32,activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(16,activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(5,activation='softmax'))
    opt = keras.optimizers.Adam(learning_rate=0.0001)
    model.compile(opt, "sparse_categorical_crossentropy", metrics=["accuracy"])
    model.summary()
```

```
    # Create strategy
    strategy = fl.server.strategy.FedAvg(
        fraction_fit=1.0,
        fraction_evaluate=1.0,
        min_fit_clients=2,
        min_evaluate_clients=2,
        min_available_clients=2,
        evaluate_fn=get_evaluate_fn(model),
        on_fit_config_fn=fit_config,
        on_evaluate_config_fn=evaluate_config,
        initial_parameters=fl.common.ndarrays_to_parameters(
        model.get_weights()),
    )

    # Start Flower server (SSL-enabled) for four rounds of federated learning
    fl.server.start_server(
        server_address="0.0.0.0:5050",
        config=fl.server.ServerConfig(num_rounds=3),
        strategy=strategy,
        certificates=(
            Path(".cache/certificates/ca.crt").read_bytes(),
            Path(".cache/certificates/server.pem").read_bytes(),
            Path(".cache/certificates/server.key").read_bytes(),
        ),
    )


def get_evaluate_fn(model):

    _, _, x_val, y_val = client.load_partition()
    def evaluate(
        server_round: int,
        parameters: fl.common.NDArrays,
        config: Dict[str, fl.common.Scalar],
    ) -> Optional[Tuple[float, Dict[str, fl.common.Scalar]]]:
        model.set_weights(parameters)
        loss, accuracy = model.evaluate(x_val, y_val)
        return loss, {"accuracy": accuracy}

    return evaluate
```

```
    config = {
        "batch_size": 16,
        "local_epochs": 5 if server_round < 2 else 5,
    }
    return config


def evaluate_config(server_round: int):

    val_steps = 5 if server_round < 4 else 10
    return {"val_steps": val_steps}


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