Utilização de Inteligência Artificial para Detecção de Dispositivos em Redes e Proteção Contra Modelos Adversariais

Aluno: Matheus Ferreira Coelho
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
matheus-coelho@ufmg.br

Orientadora: Michele Nogueira Lima
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
michele@dcc.ufmg.br

***Abstract*—This paper addresses the extraction of traffic features from a packet capture database, in order to label which devices inside the network are generating a given traffic. After the extraction, artificial intelligence and machine learning algorithms are used to train models capable of classifying these devices, based on the extracted features. After being created and their initial accuracy are verified, we will test adversarial attacks in these models, in order to check how its accuracy is modified after being attacked.**

***Resumo*—Este artigo aborda a extração de características de tráfego em bancos de dados de captura de pacotes, com o objetivo de rotular quais são os dispositivos dentro da rede que estão gerando um dado tráfego. Após a extração, são utilizados algoritmos de inteligência artificial e aprendizado de máquina para treinar modelos capazes de classificar estes dispositivos, baseado nas características extraídas. Assim que criados e verificadas suas eficácias iniciais, são testados ataques adversariais nestes modelos, de forma a verificar como sua acurácia é alterada depois de ser atacado.**

***Palavras-chave*—tráfego, redes, classificação, modelos, adversários**

I. Introdução

Em redes de computadores, normalmente prezamos pela segurança dos dispositivos que se encontram em determinado parque tecnológico. Frequentemente vemos notícias e artigos relatando ataques cibernéticos à redes e infraestruturas de pequenas, médias e grandes empresas. Muitas vezes, o que encontramos para proteção deste ambiente é algo que denominamos como firewall. O firewall pode ser explicado como um dispositivo de segurança que monitora o tráfego naquela rede, verificando os pacotes de entrada e saída e baseado em regras definidas pelo administrador, bloqueia ou permite que passem determinadas conexões. Este mecanismo pode ser tanto virtualizado como um equipamento físico.

Lendo sobre este conceito, é fácil de perceber como um firewall é essencial para o parque tecnológico de uma empresa, que pode possuir dados confidenciais que precisam de ser protegidos. Nestes dispositivos, é possível determinar o bloqueio de um endereço IP específico, junto de uma porta específica, e incluir estas regras de bloqueio/permissão o quanto for necessário.

Apesar do firewall trazer mais segurança para a empresa, relatos de empresas sofrendo ciberataques continuam surgindo, devido aos atacantes evoluírem rapidamente os seus métodos de ataque. O que vemos muitas corporações utilizando são appliances com um sistema de firewall open-source, que precisam ter regras manualmente inseridas pelo administrador. Com toda certeza, esses sistemas trazem segurança, mas seria interessante se houvesse uma forma de definir essas regras também de outra maneira.

Conforme os dias passam, os atacantes estão desenvolvendo maneiras de infiltrar nessas redes de formas cada vez mais complexas, na maioria dos casos envolvendo inteligência artificial e aprendizado de máquina. Os firewalls open-source que necessitam de administração manual provavelmente estarão sujeitos a serem burlados por estes ataques, pois alguns deles usam implementações antigas, e mesmo os que são frequentemente atualizados podem não estar atentos a estas novas tecnologias.

O objetivo deste trabalho é o estudo e desenvolvimento de um modelo de inteligência artificial que consiga detectar dispositivos dentro de uma rede, analisando o tráfego, aprendendo a classificar estes tráfegos, e a partir disso sermos capazes de definir o que pode estar na nossa rede ou não. Após esta tarefa de classificação, temos o próximo objetivo de validar o quanto o nosso modelo é seguro, quando ele é confrontado com ataques adversariais, que são perturbações que buscam "sabotar" modelos já estabelecidos.

Alguns conceitos de aprendizado de máquina e mineração de dados, como classificação e *dataframes*, serão fundamentais para esse desenvolvimento. Temos um objetivo geral de estudar e aprender mais sobre inteligência artificial e suas várias aplicações nos dias de hoje, e um objetivo mais específico de entender mais a fundo como análise de pacotes de redes, classificação supervisionada, redes neurais e ataques cibernéticos funcionam.

Na primeira parte da monografia, desenvolvemos um script capaz de extrair características de arquivos de pacotes de rede, que foram previamente coletados usando uma biblioteca. Estes arquivos foram enviados e disponibilizados em uma

máquina virtual hospedados no laboratório de computação da UFMG. Foram realizados vários testes e extraídos diferentes características destes pacotes para garantir que o script estava funcionando como deveria. Após isso, realizamos alguns ajustes de pós-tratamento, que não foram realizados no script, para poder visualizar detalhes relacionados ao que extraímos (como por exemplo, média, mediana e desvio padrão). Inicialmente, o objetivo da monografia seria utilizar esse modelo de classificação para definir regras de firewall, mas definimos que este modelo será utilizado para outros fins (detecção de dispositivos e "confronto" com os modelos adversariais).

II. Trabalhos Correlatos

A. Modelos Adversariais

O principal trabalho relacionado ao que foi desenvolvido, no qual temos uma grande inspiração, é o "*Adversarial Network Traffic: Towards Evaluating the Robustness of Deep Learning-Based Network Traffic Classification*", publicado em 2021 [1]. Nesse artigo, é estudado como algoritmos de aprendizado de máquina e aprendizado profundo demonstram uma grande eficiência na classificação de tráfego de rede.

O foco desse trabalho relacionado era verificar a forma como estes classificadores lidavam com modelos adversariais (chamados também de ANT, (*Adversarial Network Traffic*) que nada mais são do que modelos desenvolvidos para atacar a classificação original do tráfego, de forma a induzir ao erro, rotulando incorretamente. São mostradas várias formas que os ANTs podem ser utilizados, atacando a categorização de pacotes, fluxo de rede e o fluxo temporal de rede (*time series*), trazendo gráficos que mostram as diferentes mudanças na eficiência dos modelos originais, dependendo de como as modificações são injetadas (se é feito no começo, meio ou final do pacote, se são alterados os números das camadas de transporte ou outras características da base analisada).

B. MITRA

Outro artigo que se relaciona com o que propomos possui o título "*A Dynamic Method to Protect User Privacy Against Traffic-based Attacks on Smart Home*", publicado em 2023 [2]. Este trabalho desenvolveu um método MITRA para ajudar a reduzir os ataques em redes domésticas envolvendo IoT (*Internet of Things*), injetando dados falsos de forma a não serem facilmente identificados por modelos de classificação, dificultando o roubo ou interceptação de dados reais por um possível atacante. Os passos do funcionamento do MITRA começam pela coleta de tráfego, seguido pelo pré-processamento desta coleta e a identificação dos dispositivos, se assemelhando muito com o que foi desenvolvido nesta monografia. Os próximos passos do método, que são a criação de dispositivos fictícios e geração de tráfego falso, é análogo aos modelos adversariais citados anteriormente.

C. Outros artigos

Outros artigos que foram estudados no início das pesquisas incluem o "*Network Traffic Anomaly Detection via Deep Learning*", publicado em 2021 [3], em que foi utilizado uma implementação de firewall open-source do tipo FreeBSD chamado *pfSense*. Este é um sistema operacional bastante difundido, e teve o papel de coletar logs de tráfego de rede, para então tentar classificar esses logs em maliciosos ou não com a ajuda de aprendizado profundo.

Mais artigos interessantes que foram pesquisados são o "*Design and Implementation of an Artificial Intelligence-based Web Application Firewall Model*" [4], além do "*Building New Generation Firewall Including Artificial Intelligence*" [5]. Em ambos, foram desenvolvidas aplicações de firewall com forte envolvimento de inteligência artificial.

III. Metodologia

A. Ambiente do laboratório

Para início do desenvolvimento do trabalho, foram solicitadas credenciais de acesso à VPN dos laboratórios da UFMG. Isso foi necessário para poder nos conectarmos à rede da universidade e acessar uma máquina virtual destinada justamente para a execução das tarefas. A realização das atividades em um ambiente virtual e remoto traz a vantagem de não precisar se atentar tanto com o consumo de recursos por parte dos códigos sendo executados, tirando uma parcela dessa carga da nossa máquina local.

Após conectar na VPN com *login* e senha fornecidos, é feita então uma conexão SSH com a VM (*Virtual Machine*, forma resumida de chamar a máquina virtual) que está hospedada em uma máquina dos laboratórios, com um IP fixo. Já conectado na VM, temos disponíveis um ambiente de desenvolvimento já preparado, com auxílio do Anaconda. Na pasta raiz (home) da máquina virtual, navegamos para a pasta FAPESP onde contém os arquivos necessários para estudo. Além disso, temos o comandos *conda activate teste* usado para trocar o ambiente Python atualmente ativo, visto que só no "teste" temos todas as bibliotecas essenciais.

B. Tshark e PCAPs

O primeiro conceito que precisa ser explicitado é dos arquivos de extensão *PCAP*. Basicamente, os PCAPs são arquivos de captura de pacotes de rede. Estes documentos possuem informações essenciais do que está dentro daquela rede gerando tráfego, como por exemplo: data, IP de origem e destino, endereço MAC de origem e destino, protocolo de transporte, tamanho do pacote, entre outras características. São inúmeros dados que podem ser extraídos, e eles podem ser gerados dentro de qualquer rede utilizando uma ferramenta chamada *Wireshark*, também conhecida como *Tshark*. O Tshark tem como função o análise de pacotes para análise de comportamento e problemas de redes. Assim como possui a capacidade de gerar estes PCAPs, essa ferramenta também pode extrair os dados destes arquivos e convertê-los em .CSV, uma extensão amplamente utilizada para *datasets* em ciência de dados. Nessa nossa etapa de extração de características, o resumo do que vamos fazer é a construção de um script em Python capaz de gerar comandos do Tshark e pegar estes aspectos de cada PCAP que temos.

Os comandos do Tshark vão extrair os dados que queremos dos PCAPs e transformá-los em CSV. Criamos três pastas auxiliares: uma para armazenar os PCAPs originais, outra para armazenar os CSVs convertidos (pasta *tshark*) e outra para os CSVs já tratados na parte de pré-processamento (pasta *csvfilter*). Outrossim, foram colocados na raiz os CSVs contendo os nomes dos dispositivos existentes na rede capturada. Cada dispositivo possui o seu respectivo MAC, e será feita essa correspondência no processo.

Uma biblioteca importante do Python que é utilizada no código é a *multiprocessing*, que traz ferramentas importantes para o desempenho das extrações. Os arquivos de captura são bem extensos, então a adição destes métodos beneficia na eficiência das pesquisas e conversões das capturas. Além dessa biblioteca, temos também a *tqdm*, importante para a visualização do progresso das extrações conforme o código vai sendo executado, mostrando a porcentagem de conclusão de cada arquivo. Iniciando o script, nós percorremos cada PCAP disponível na pasta, para cada um será executado um comando Tshark que retira as características, cada trabalho dos comandos é adicionado em um vetor de *tasks* que serão executados em sequência, após todos os comandos para cada captura serem armazenados, ao fim do loop.

Vamos analisar então a função que constrói o comando Tshark: ele recebe o caminho do diretório onde está o arquivo de captura, inicia o comando com "*tshark -r*, com o parâmetro -r indicando que será feita uma leitura (read) do PCAP. No parâmetro *tsharkOptions*, é recebido uma lista contendo todas as partes que compõem o comando Tshark final, declarada como *TSHARK_OPT* algumas linhas abaixo. O caminho do arquivo PCAP é então utilizado para criar um novo caminho onde CSV convertido será armazenado, adicionando ele ao final do comando. Segue o exemplo do comando Tshark completo para um dos PCAPs:

```
tshark -r
"/home/mcoelho/FAPESP/pcaps/
16-09-23.pcap"
-t ad -T fields -E separator=,
-e _ws.col.Time
-e ip.src -e ip.dst -e ip.proto
-e tcp.len -e udp.length
-e eth.src -e eth.dst
-e tcp.srcport -e tcp.dstport
-e udp.srcport -e udp.dstport >
"/home/mcoelho/FAPESP/pcaps/tshark/
16-09-23.pcap.csv"
```

O que quer dizer cada parte do comando:

- *-t ad*: diz ao Tshark o formato que ele deve retornar o *timestamp* (data e hora do pacote).
- *-T fields*: indica que queremos extrair os *fields*, nossas características.
- *-E separator*: indica qual caractere está separando cada coluna em cada entrada dos dados coletados. Normalmente, este caractere é a vírgula ou de tabulação.
- *-e*: a partir daqui, cada -e é uma característica que queremos extrair. Os arquivos de captura contém inúmeros dados que podemos retirar deles, e é nessa parte do código que escolhemos o que precisamos. Cada um possui a sintaxe específica, como por exemplo o *_ws.col.Time* para o timestamp, *ip.dst* para o IP de destino, e assim por diante.

Logo em baixo, temos a lista *PACKET_COLUMNS*, que contém o nome das colunas para as quais vamos renomear na próxima etapa (para não precisar ter nomes incomuns e pouco explicativos como o *_ws.col.Time* do timestamp). Pode-se perceber que cada nome de coluna está exatamente na mesma ordem das *features* que estão no comando.

Após a extração para o primeiro CSV, vamos então para a etapa de pré-processamento que realizará algumas tratativas para este CSV gerado. Isso é feito com uma função *executeCommand*, que é enviada para uma instância da *multiprocessing*, ajudando no consumo de recursos para cada subprocesso. Nesta função, cada um dos CSV gerados será lido e transformado em um *dataframe* do Pandas, uma biblioteca amplamente conhecida para análise de dados em Python. As colunas do dataframe são renomeadas para os títulos que definimos em *PACKET_COLUMNS*, os possíveis valores vazios são substituídos por 0, além de ser realizada a soma das portas TCP e UDP do pacote, tanto para origem quanto destino. Fazemos isso para poder ter apenas uma coluna mostrando a porta do pacote, visto que a número resultante da soma já vai nos dizer se o protocolo é TCP ou UDP. Os pacotes dos dois protocolos também são somados, seguindo a mesma lógica, e os de tamanho 0 são deletados, já que não possuem relevância para análises. Todas estas colunas que separavam os dois protocolos, tanto para o tamanho quanto portas de origem/destino, são deletadas para manter uma coluna só para cada uma dessas informações.

É feito um tratamento para não incluir dispositivos de *broadcast* no CSV. Esse tipo de dispositivo é um host que envia um pacote para todos os outros hosts pertencentes àquela rede. Para evitar de termos muitas entradas relacionadas a um dispositivo com este MAC de *broadcast*, ele é removido da coleta. Após isso, utilizamos o nosso *device_list.csv*, contendo a identificação de cada equipamento naquela rede, para atribuir os nomes no nosso CSV sendo processado. É usada a função *map* do Python que compara o MAC do dispositivo com aqueles que têm disponíveis no device_list, e se forem iguais, adiciona em uma coluna *device_src_name* a identificação do aparelho. A função completa de pré-processamento está na próxima página.

A utilização dessa função é útil para já adiantar a fase de estudo do CSV, fazendo o tratamento agora antes mesmo de termos nosso arquivo final. Depois de tudo isso, finalmente o resultado do nosso pré-processamento será salvo em um novo CSV daquele PCAP originalmente analisado, dentro da pasta *csvfilter*. Finalizando o nosso script, todos estes CSVs tratados são lidos, transformados em um dataframe, retiradas possíveis colunas que não vamos mais precisar caso necessário, e adicionados em um vetor de dataframes. Por fim,

Código 1. Função de pré-processamento

```
def executeCommand(command, outfilename, rotulos, salveFilter,
rotul):
sem.acquire()

subprocess.call(command, shell=True, stdout=subprocess.PIPE)
df = pd.read_csv(outfilename, error_bad_lines=False,
warn_bad_lines=False)
df.columns = PACKET_COLUMNS

df.fillna(0, inplace = True)

df['src_port'] = df['tcp_srcport'] + df['udp_srcport']
df['dst_port'] = df['tcp_dstport'] + df['udp_dstport']

# delete packets with size equal to 0
df['len'] = df['len_tcp'] + df['len_udp']
df = df[df['len'] !=0]

df = df.drop(columns=['len_tcp','len_udp','tcp_srcport',
'tcp_dstport', 'udp_srcport', 'udp_dstport'])

# No broadcast
df.drop(df[(df['mac_dst'] == 'ff:ff:ff:ff:ff:ff') |
(df['ip_dst'] == '255.255.255.255')].index, inplace=True)

# add o rotulo do device por meio do mac (usa arq csv)
df['device_src_name'] = df['mac_src'].map(rotul.set_index('mac')
['device_name'])

df.to_csv(salveFilter+rotulos+'.csv', index=False)
print('done writing packet to : ' + salveFilter+rotulos+'.csv')
sem.release()
```

é feito um merge desse vetor em um dataframe só, e salvo em um dataframe final nomeado *df_total.csv*.

C. Execução do código e base resultante

Utilizamos então, no terminal do Putty, um comando Python para execução do script e extração das *features*. A biblioteca *tqdm* mencionada antes é perfeitamente visível no terminal enquanto o código é executado, mostrado na figura 1.

Podemos então, visualizar o nosso dataframe final que contém as características já tratadas e com os dispositivos já rotulados, conforme figura 2.

Utilizamos o Google Colab para visualizar melhor estes dados. Temos o timestamp, que nos informa a data e hora do pacote, o protocolo (6 para TCP e 17 para UDP), os MACs de origem e destino, o tamanho do pacote além do dispositivo que transmitiu o pacote em si, categorizado com o CSV auxiliar *device_list*.

Uma das dificuldades encontradas nesta extração foi a correta formatação da característica de timestamp. O primeiro *field* que tentamos extrair da captura com o Tshark foi o *frame.time*, que também contém as informações de data. Porém, na formatação padrão do *frame.time*, a data possui uma vírgula separando o dia e mês do restante da string. Visto que no nosso comando Tshark a vírgula estava definida para separar cada coluna do CSV, a característica de timestamp estava sendo separada em duas colunas diferentes, que não era a nossa saída esperada. Foi possível contornar este problema extraindo o *_ws.col.Time* e alterando o formato retornado pelo comando, definido pelo parâmetro -t.

Para fins de estudo e para familiarizar com a estrutura do código, foi feito um novo script alterando alguns parâmetros do comando Tshark e também rotulando um dos CSVs com uma lista de dispositivos ”falsa”, diferente da lista definida anteriormente. Neste novo script, extraímos apenas o tamanho dos pacotes, o MAC de origem e destino e o timestamp, dessa vez utilizando o *frame.time_epoch*. Já no comando, adicionamos novos parâmetros de forma a já incluir no CSV apenas os pacotes que são TCP ou UDP, visto que a captura pode ter outros protocolos. Segue o novo comando Tshark que será gerado:

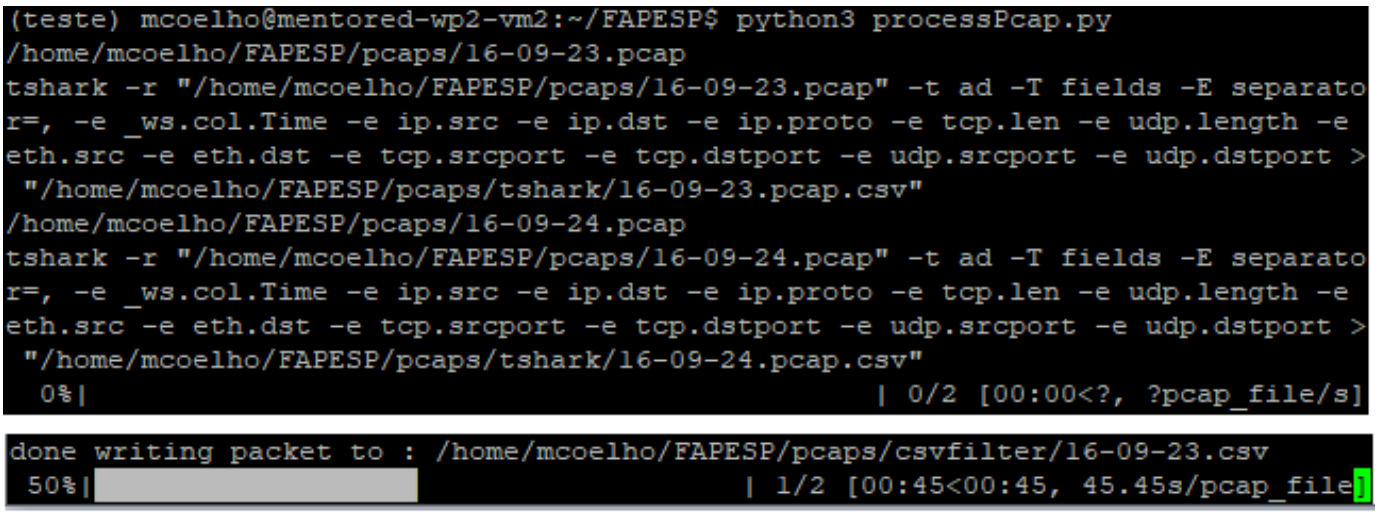

Fig. 1. Código de extração dos PCAPs sendo executado

	timestamp	proto	mac_src	mac_dst	len	device_src_name
0	2016-09-22 14:00:02.383174	6.0	d0:52:a8:00:67:5e	14:cc:20:51:33:ea	1.0	Device1
1	2016-09-22 14:00:02.447911	17.0	70:ee:50:18:34:43	14:cc:20:51:33:ea	54.0	Camera1
2	2016-09-22 14:00:02.555796	6.0	14:cc:20:51:33:ea	18:b7:9e:02:20:44	2.0	Gateway
3	2016-09-22 14:00:02.821282	17.0	14:cc:20:51:33:ea	70:ee:50:18:34:43	9.0	Gateway
4	2016-09-22 14:00:03.044023	17.0	14:cc:20:51:33:ea	70:ee:50:18:34:43	213.0	Gateway
...	...	...	...	...	...	...
858797	2016-09-24 13:59:57.586715	6.0	30:8c:fb:2f:e4:b2	14:cc:20:51:33:ea	90.0	Camera4
858798	2016-09-24 13:59:57.699441	6.0	00:16:6c:ab:6b:88	14:cc:20:51:33:ea	122.0	Camera3
858799	2016-09-24 13:59:57.859937	6.0	14:cc:20:51:33:ea	00:16:6c:ab:6b:88	138.0	Gateway
858800	2016-09-24 13:59:57.938215	6.0	08:21:ef:3b:fc:e3	14:cc:20:51:33:ea	91.0	Tablet1
858801	2016-09-24 13:59:58.502097	6.0	30:8c:fb:2f:e4:b2	14:cc:20:51:33:ea	90.0	Camera4

858802 rows × 6 columns

Fig. 2. Visualização das entradas do dataframe final

```
tshark -r "/home/mcoelho/FAPESP/pcaps/
16-09-23.pcap"
-t e -T fields -E separator=,
-e frame.time_epoch
-e tcp.len -e udp.length -e eth.src
-e eth.dst
-Y "(ip.proto==6)||(ip.proto==17)" >
"/home/mcoelho/FAPESP/pcaps/tshark/
16-09-23.pcap.csv"
```

Assim, antes de iniciar o loop inicial de percorrer os PCAPs da pasta auxiliar, foi chamada a função de construção do comando separadamente, para incluir na lista de tasks e ser executado primeiro, e as próximas tasks serão a extração dos PCAPS originais, que já havíamos analisado anteriormente. Depois, para evitar que os dados deste PCAP falso fossem incluídos no DF total que terá os dados "reais", o CSV já filtrado é movido para a pasta raiz da VM, pois o final do script faz um *merge* de todos os CSVs que estão na pasta *csvfilter*. Assim, o arquivo do PCAP falso é isolado e o *df_total* ainda tem os originais, juntando a lista de dataframes em um só. Visualizaremos uma parte do dataframe isolado na figura 3.

	timestamp	mac_src	mac_dst	len	device_src_name
0	1.635168e+09	64:32:a8:d1:99:e5	14:cc:20:51:33:ea	62.0	device-fake1
1	1.635168e+09	64:32:a8:d1:99:e5	14:cc:20:51:33:ea	62.0	device-fake1
2	1.635168e+09	64:32:a8:d1:99:e5	14:cc:20:51:33:ea	62.0	device-fake1
3	1.635168e+09	64:32:a8:d1:99:e5	14:cc:20:51:33:ea	62.0	device-fake1
4	1.635168e+09	64:32:a8:d1:99:e5	14:cc:20:51:33:ea	62.0	device-fake1
...	...	...	...	...	...
3594	1.635168e+09	e8:d8:19:9b:f9:33	14:cc:20:51:33:ea	62.0	device-fake6
3595	1.635168e+09	e8:d8:19:9b:f9:33	14:cc:20:51:33:ea	62.0	device-fake6
3596	1.635168e+09	e8:d8:19:9b:f9:33	14:cc:20:51:33:ea	62.0	device-fake6
3597	1.635168e+09	e8:d8:19:9b:f9:33	14:cc:20:51:33:ea	62.0	device-fake6
3598	1.635168e+09	e8:d8:19:9b:f9:33	14:cc:20:51:33:ea	62.0	device-fake6

3599 rows × 5 columns

Fig. 3. Visualização do novo CSV

Temos as mesmas colunas do CSV anterior, com exceção da coluna de protocolo. A coluna de timestamp possui um formato diferente, deixado dessa forma para visualizar uma outra abordagem possível para essa característica. É nítido ver que na outra formatação, conseguimos entender melhor o timestamp. Na última coluna, temos a rotulação dos dispositivos falsos, feita com um CSV auxiliar que foi construído manualmente.

D. Desenvolvimento no Google Colab

Para facilitar na visualização do código e no retorno de cada execução, esta segunda parte da monografia em sua maioria foi desenvolvida no Google Colab. Inicialmente, importamos as bibliotecas necessárias, que são bastante conhecidas (*pandas, numpy, sklearn*), além da importação do nosso CSV do dataset final, utilizando a *gdown*, de forma a baixar esse arquivo direto do Google Drive toda vez que o código for executado, de forma a não precisar realizar o *upload* do arquivo manualmente toda vez que o notebook é aberto.

Na Figura 4, mostramos as primeiras linhas do nosso dataset total, de forma a explicitar as colunas e informações que possuimos.

No momento de extração dos PCAPs, as características que coletamos incluíam apenas o *timestamp*, endereço MAC de origem e destino e tamanho do pacote, além do rótulo de cada dispositivo. A utilização do endereço MAC nestas análises, independente de qual rede estes dados foram coletados, pode ser considerada uma invasão de privacidade, ferindo a LGPD (Lei Geral de Proteção de Dados). Porém, se removermos essas informações do nosso dataset, teremos poucos dados restantes para analisar e alimentar os algoritmos. Sendo assim, uma opção disponível é pegar a característica de tamanho do pacote e *timestamp*, que são muito úteis para análise de tráfego, calcular medidas estatísticas destas características e inseri-las no dataset. As medidas que escolhemos inicialmente, foram a média, mediana e desvio padrão. Além disso, a cada 3 linhas de um mesmo dispositivo no nosso dataset original, juntamos estas linhas em uma só, de forma a formar grupos de cada dispositivo.

A coluna que vamos classificar com os modelos é a coluna de *device_name*, que contém o nome dos dispositivos presentes na rede. Nossa tarefa é uma classificação supervisionada, visto que já temos conhecimento das classes. Sendo assim, nosso próximo passo é transformar os valores dessas colunas em inteiros, visto que nossas classes estão em formato de *string* e normalmente os algoritmos de classificação necessitam que as *labels* sejam números inteiros para conseguir interpreta-las. Fazemos a utilização da biblioteca *LabelEncoder* para esse procedimento.

Em seguida, a coluna *device_name* é separada em um

```
[ ] df = pd.read_csv('df_total.csv')
    df.head()
```

	device_name	mean_n_bytes	stdev_n_bytes	median_n_bytes	mean_timestamp	stdev_timestamp	median_timestamp
0	Assistant1	121.666667	57.185857	157.0	1.474553e+09	6.363523	1.474553e+09
1	Assistant1	126.666667	50.135372	157.0	1.474553e+09	4.934770	1.474553e+09
2	Assistant1	121.666667	57.185857	157.0	1.474553e+09	5.098356	1.474553e+09
3	Assistant1	51.000000	7.071068	56.0	1.474553e+09	4.278881	1.474553e+09
4	Assistant1	121.666667	57.185857	157.0	1.474553e+09	10.553324	1.474553e+09

Fig. 4. Primeiras linhas do arquivo .CSV com dados estatísticos

dataset contendo apenas esta coluna (comumente chamado de dataset Y), e separamos outro dataset contendo todas as características menos a *device_name* (comumente chamado de dataset X). Isso possibilita a separação dos dados em bases de treino e teste, essencial para treinamento de modelos de classificação. Basicamente, vamos selecionar uma parcela dos dados para treinar o nosso modelo, em que as características e rótulos de cada entrada será analisada, estudada e "aprendida" pelo algoritmo. Esse conhecimento adquirido será posteriormente colocado à prova nos dados de teste, verificando se o modelo será capaz de acertar a classificação dos dispositivos no teste, com as informações que foram analisadas no treino.

Teremos um dataset X de treino e teste, assim como um dataset Y de treino e teste. Dos parâmetros de criação destas bases, utilizamos o *random_state* e *text_size*. O primeiro é uma *seed* que determina a forma como os dados serão separados, fazendo com que toda vez que executamos eles serão divididos da mesma forma, evitando diferenças nesta seção do trabalho. O segundo parâmetro se refere a quanto dos dados será colocado em treino e quanto irá para teste. Com o valor de 0.3, determinamos que 30% do dataset original será usado para teste, e 70% usado para treino.

IV. Classificadores

A. Decision Tree

O primeiro classificador que utilizamos é o Decision Tree, um algoritmo de árvores de decisão. Ele separa os atributos em nós, e várias arestas que representam uma tomada de decisão. O caminho percorrido em toda árvore, ao final, determinará a classificação da entrada recebida. Criando um modelo com este classificador, e verificando a acurácia retornada pela sua função de predição, obtivemos um resultado de 97%. Isso significa que, no nosso dataset de teste, ele conseguiu classificar corretamente 97% das nossas entradas. Sem dúvida alguma, é uma métrica bastante satisfatória.

Outros índices que podemos observar sobre a predição de modelos está contido no *Classification Report*[1]. O que podemos observar neste relatório são as seguintes métricas:

- *Precision*: informa a porcentagem de entradas classificadas como X, que realmente eram X.
- *Recall*: de todos os dispositivos que são X, informa a porcentagem deles que o modelo foi capaz de identificar.
- *F1 Score*: informa a média harmônica do *precision* e *recall*. Mede o equilíbrio do nosso modelo entre as duas métricas anteriores.
- *Support*: a quantidade de entradas usadas por cada métrica para realizar os cálculos.

```
======================Classification Report=======================
              precision    recall  f1-score   support

           0       0.96      0.96      0.96      3104
           1       0.93      0.90      0.91       198
           2       0.00      0.00      0.00         1
           3       0.97      0.97      0.97      1807
           4       0.97      0.98      0.97       570
           5       0.99      0.99      0.99      5593
           6       1.00      1.00      1.00     19281
           7       0.99      0.99      0.99      1945
           8       0.98      0.98      0.98     39453
           9       0.81      0.81      0.81      2698
          10       0.95      0.95      0.95      1462
          11       0.67      0.66      0.66       440
          12       0.12      0.16      0.13        19
          13       0.88      0.91      0.90       284
          14       0.87      0.86      0.87       320
          15       0.91      0.62      0.74        16
          16       0.83      0.79      0.81        19
          17       0.97      0.97      0.97       236
          18       0.93      0.88      0.90       153
          19       0.75      0.72      0.74       626
          20       0.99      0.97      0.98       178
          21       0.98      0.98      0.98      3177
```

Fig. 5. Relatório de classificação

Além do relatório de classificação, normalmente verificamos também a matriz de confusão, muito útil para análise em classificação binária. Na sua versão mais simples, a matriz de confusão possui 4 valores:

- *Verdadeiro positivo*: número de entradas que o modelo classificou como 1, e realmente eram 1.
- *Falso negativo*: número de entradas que o modelo classificou como 0, mas eram 1.

[1]https://medium.com/@chanakapinfo/classification-report-explained-precision-recall-accuracy-macro-average-and-weighted-average-8cd358ee2f8a

- *Falso positivo*: número de entradas que o modelo classificou como 1, mas eram 0.
- *Verdadeiro negativo*: número de entradas que o modelo classificou como 0, e realmente eram 0.

Os valores de verdadeiro positivo e verdadeiro negativo ficam na diagonal principal da matriz, e os falsos na outra diagonal. Visto que a nossa tarefa é de multi-classificação, tendo muito mais do que 2 *labels* possíveis, a matriz de confusão ainda pode ser utilizada mas é de difícil visualização, tendo uma diagonal principal mais extensa.

B. Random Forest

O próximo classificador que testamos é o *Random Forest*, um algoritmo que basicamente usa um conjunto de árvores de decisão para treinar o modelo. Com esse classificador, também obtivemos uma eficiência de 97% nas predições. Estes resultados mostram a eficiência destes dois classificadores, amplamente usados em aprendizado de máquina.

C. K-Neighbors

Em seguida, foi utilizado o *K-Neighbors*, classificador que determinará as labels baseado em quão similar é um dispositivo em relação ao outro, realizando cálculos matemáticos para determinar a distância de um "ponto" a outro (interpretando cada entrada como um ponto). No nosso experimento, determinamos que a função usaria a distância euclidiana. Além disso, é definido o número de vizinhos, que seria a quantidade de menores distâncias que o algoritmo que vai analisar, antes de definir qual classe provavelmente aquela entrada pertence. Após realizar a predição com estes parâmetros, o *K-Neighbors* retornou uma acurácia de 80%.

Um teste válido é verificar como a eficiência do modelo se alteraria conforme alterássemos o valor de K recebido na declaração. Para uma visualização mais interessante, foi gerado um gráfico, em que o eixo X é o valor de K conforme ele vai aumentando, e o eixo Y representa a acurácia do modelo com cada valor de K. É facilmente perceptível que assim que K vai aumentando, a eficiência vai abaixando. Sendo assim, o valor ideal nesse caso seria 2 (que apresentou o valor de 80%).

D. Redes Neurais

Até agora, analisamos a acurácia de classificadores lineares. Vamos experimentar também tarefas de classificação utilizando aprendizado profundo, envolvendo redes neurais. Uma biblioteca amplamente difundida para esse fim em Python é o *Keras*[2]. Inicialmente, declaramos nossa primeira rede neural (que é inicializada com um *Sequential*) seguida de duas camadas densas. A camada densa possui uma matriz de pesos que é ajustada durante o período de treinamento. Também é necessária a função de ativação[3], que calcula a saída do nó baseada em cada entrada e nos pesos da matriz, atualizados a cada iteração. Escolhemos a função *ReLU*. O

[2]https://keras.io/

[3]https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

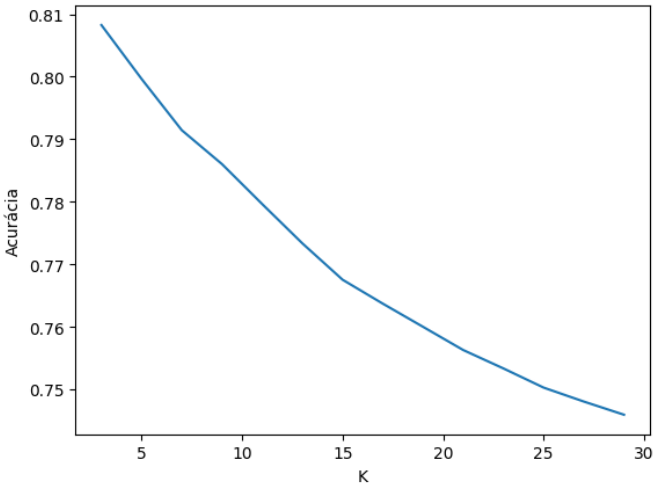

Fig. 6. Avaliação do valor de K no K-Neighbors

valor inteiro recebido na declaração determina a dimensionalidade do espaço de saída, em que definimos o número 100 para testes. Ademais, fornecemos à primeira camada qual a dimensão dos nossos dados de entrada (na nossa situação, tirando a coluna de *device_name*, temos 6 colunas). A segunda camada densa terá dimensionalidade 22. Considerando que nosso dataset possui 22 tipos de dispositivos diferentes, que foram transformados em inteiros e tendo um intervalo de 0 até 21 de *labels* possíveis, o número mínimo de dimensionalidade aceito na camada densa é 22. Para finalizar, é necessário utilizar mais uma função de ativação, que nesta última camada foi escolhida a função softmax, conhecida por ser usada em tarefas de classificação multi-classe. O próximo passo é a compilação deste modelo com redes neurais, em que foi escolhido o otimizador Adam e a entropia cruzada categórica, na sua versão esparsa, como função de perda. Depois, é realizada o treinamento do modelo com a função *fit*, que também é utilizada pelos classificadores lineares verificados anteriormente assim como a maioria. No caso deste modelo, alteramos também alguns parâmetros específicos, que são os seguintes:

- *Batch size*: número de amostras da nossa base de treino que será utilizado em cada iteração do treinamento do modelo.
- *Epochs*: define o número de iterações do treinamento. Após cada iteração, os pesos nas camadas são atualizados.
- *Validation split*: fração da base de treino que será usada ao final de cada iteração para avaliar as métricas de eficiência.

Finalmente, depois de todas essas declarações, usamos a função *evaluate* para executar o treinamento do modelo e receber os resultados ao final. Com esta nossa primeira rede neural, obtivemos uma acurácia de 23%, um valor pouco satisfatório.

Sendo assim, experimentamos com mais camadas e novos tipos de camadas. Nossa próxima rede neural inicia com

uma camada de *Dropout*, útil para reduzir overfitting. Depois, declaramos a mesma camada densa com dimensão 100, seguida da camada densa de dimensão 22, e após ela temos uma camada de *BatchNormalization*. Esse novo tipo normaliza as entradas, mantendo a média da saída perto de 0 e o desvio padrão perto de 1. Esse ajuste traz mais estabilidade ao modelo. Para finalizar, uma camada de *Activation*, que nada mais é do que uma camada contendo apenas uma função de ativação. Anteriormente, declaramos as funções de ativação junto com as camadas densas, mas também é possível declara-las separadamente. Para fechar a rede neural, usamos a mesma função softmax para ativação. Inicialmente, mantemos os mesmos parâmetros no momento de executar a função fit. Desta vez, o nosso modelo retornou um resultado de 48% de acurácia, uma melhora significativa em relação à primeira rede neural.

Foram feitas as seguintes tentativas de aprimorar este resultado:

- *Outras camadas*: utilização de outros tipos de camadas, como a *Conv2D* [6], que consiste de redes convolucionais normalmente usadas em reconhecimento de imagens, sendo assim não funcionando bem no nosso contexto apesar dos esforços;
- *Alteração de parâmetros*: vários testes foram realizados alterando os parâmetros na declaração das redes e nas funções de compilação e ativação. Mesmo subindo o número de *epochs*, a partir de uma certa iteração a acurácia permanecia na faixa dos 48% e não aprimorava além disso. Aumentando e diminuindo o *batch_size*, variava o tempo de execução do treinamento, podendo levar mais ou menos tempo para finalizar, mas o resultado permaneceu o mesmo. Mudança do otimizador e função de perda não surtiu efeito. Ao usar a função padrão de entropia cruzada categórica, é retornado um erro, funcionando apenas com a versão esparsa.
- *Classificadores similares*: verificamos nas bibliotecas do Python classificadores que também utilizam dos conceitos de aprendizado profundo, como o Perceptron, uma das primeiras implementações de redes neurais e uma das mais simples, além do MLP (*Multi-Layer Perceptron*), que é uma versão mais complexa do Perceptron. Estes dois classificadores tiveram a mesma eficácia.

Sendo assim, decidimos verificar a atuação dos ataques adversariais nesta rede neural que construímos, além de também testar nos modelos anteriores.

V. FGSM (Fast Gradient Sign Method)

O FGSM [7] é uma técnica popular nos estudos de aprendizado de máquina, que tem como objetivo causar perturbações na base de dados original, gerando amostras adversariais [8]. Resumidamente, o método realiza pequenas alterações nos nossos dados, baseadas nos gradientes da função de perda relacionados à entrada, de forma a parecer uma diferença mínima para a percepção humana, mas para os modelos de classificação essa diferença pode leva-los a rotular incorretamente os dados, com o conhecimento que já foi adquirido do dataset. Em classificações de imagens, essa perturbação é quase imperceptível, mas em alguns casos é possível notar uma diferença na colorização da foto. Como nesta monografia temos dados numéricos, não é tão clara essa ”representação”. Na Figura 7, é demonstrado as alterações que são inseridas em uma foto. Antes do ataque, o classificador identificava corretamente a gravura como sendo um panda, mas após o ataque, passou a identifica-la como um gibão (primata), mesmo que a foto ainda pareça idêntica.

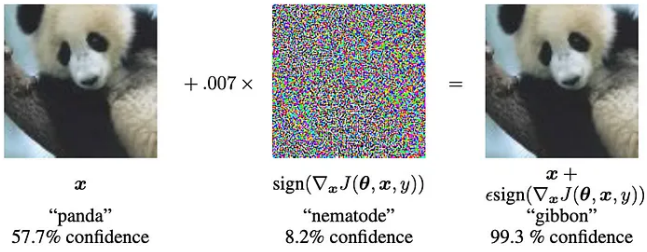

Fig. 7. Demonstração do FGSM em imagens

Para começarmos com o FGSM no código, primeiro precisamos instalar um pacote denominado *adversarial-robustness-toolbox*, que facilitará os nossos experimentos. Após isso, importamos as funções que precisamos das bibliotecas, que envolvem o FGSM em si e uma função de classificador do próprio Keras. Na declaração de um novo classificador, enviamos como parâmetro o nosso modelo de rede neural já treinado, criamos um ataque adversarial usando este classificador novo como estimador, e então geramos um novo dataset em cima do nosso dataset original de teste, contendo as alterações nos valores. Comparando os dois, podemos ver claramente que houve uma pequena perturbação em cada valor das entradas.

O próximo passo é utilizar a função *predict* do nosso modelo para tentar prever a classificação dos dispositivos, mas dessa vez fazendo essa previsão no dataset alterado. Ao final, verificamos que a acurácia do nosso modelo, após o ataque, permaneceu como 48%. Isso provavelmente aconteceu devido a essa métrica já possuir um valor abaixo inicial, mesmo sem os ataques aplicados.

Para agregar aos nossos estudos, aplicamos o FGSM nos modelos lineares, que foram os primeiros que definimos no começo do desenvolvimento. Nosso modelo do Decision Tree, que anteriormente apresentou uma eficácia de 97%, teve seu resultado reduzido para 73%, demonstrando que as amostras adversariais geradas afetaram o desempenho deste classificador.

No modelo de Random Forest, a acurácia original de 97% abaixou para 86% após a aplicação das amostras perturbadas, o que é algo curioso, visto que ele tinha a mesma acurácia do Decision Tree inicialmente e os dois algoritmos são similares, mas no caso do Random Forest a eficácia abaixou apenas 11%, em comparação ao primeiro que reduziu 24%. Talvez pela floresta aleatória conter várias árvores de decisão, ele foi capaz de se proteger dos adversários com mais precisão.

	mean_n_bytes		mean_n_bytes
16823	9.000000	0	8.200000
96248	90.000000	1	89.199997
137356	1410.000000	2	1409.199951
100914	90.000000	3	89.199997
180693	1448.000000	4	1447.199951
...	...	...	...
102569	1.000000	81575	0.200000
72377	90.000000	81576	89.199997
243226	1448.000000	81577	1447.199951
36575	340.666667	81578	339.866669
53414	1025.333333	81579	1024.533325
81580 rows × 6 columns		81580 rows × 6 columns	

Fig. 8. Perturbações causadas pelo FGSM

Em relação ao K-Neighbors, a acurácia original de 80% teve uma diminuição para 78%. Dentre os modelos lineares, este foi o menos afetado, tendo uma redução de apenas 2%.

Abaixo, temos um gráfico mostrando visualmente a atuação dos ataques em cada caso. O eixo Y representa a acurácia e o eixo X cada um dos classificadores testados. As barras azuis demonstram a acurácia antes do ataque adversarial, e as barras vermelhas a acurácia após o ataque.

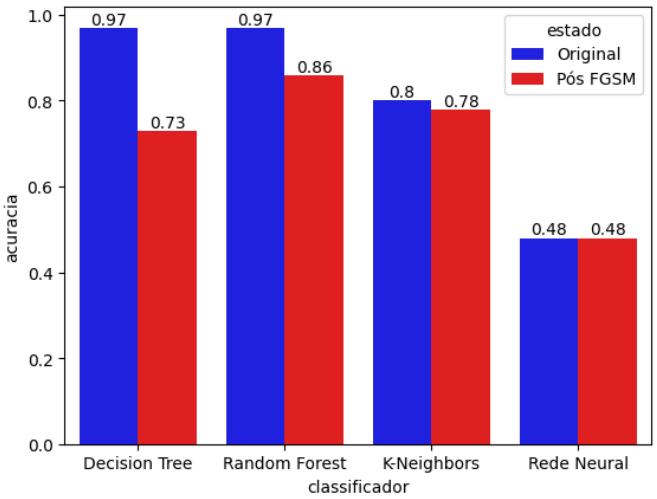

Fig. 9. Gráfico de barras contendo a atuação do FGSM

VI. Conclusão

Explorou-se neste trabalho os arquivos de captura PCAPs, que contém dados de tráfego de rede que podem ser gerados em qualquer ambiente, além da utilização de uma importante ferramenta chamada Wireshark/Tshark, criada para análise de protocolos de rede, com o objetivo de extrair destas capturas características fundamentais para o estudo de uma rede de dispositivos. Além disso, a junção dessas ferramentas com a linguagem de programação Python se mostrou muito poderosa, visto que é uma linguagem já amplamente conhecida por suas bibliotecas de ciência de dados, e as várias possibilidades que o Python nos traz auxiliou a extrair e converter os dados dos PCAPs em conjunto com o Tshark, realizar pré-processamento destas informações, salvando tudo em um CSV, um arquivo que contém nosso *dataset* final, que foram posteriormente alimentados para algoritmos de inteligência artificial e aprendizado de máquina.

Na segunda parte do trabalho, visualizamos mais uma vez o dataset final gerado, optando por remover as características de endereço MAC e tamanho do pacote, já que elas podem trazer um viés no treinamento do modelo, além de ser consideradas uma invasão de privacidade na rede analisada. Dessa forma, definimos um método capaz de extrair dados estatísticos do nosso *dataframe*, como por exemplo a mediana, média e desvio padrão, com o objetivo de popular a nossa base com mais dados, visto que a remoção das características dos MACs e pacotes deixaram a base original com poucas informações.

Em seguida, os nomes dos dispositivos são transformados em inteiros, para que fossem interpretados corretamente pelos algoritmos. Em um dado momento, todos os elementos do *dataset* haviam sido transformados em inteiros, um ajuste incorreto que foi corrigido durante o desenvolvimento. Logo depois, a base é separada em duas, uma contendo as características sem a coluna com os nomes dos dispositivos, e outra sendo um vetor coluna contendo apenas estes nomes. Esse procedimento é necessário definirmos as bases de treino e teste, etapa preliminar essencial para tarefas de aprendizado supervisionado.

Começamos pelos classificadores lineares, envolvendo algoritmos já difundidos presentes na literatura, como *Decision Tree*, *Random Forest* e *K-Neighbors*. Também foram testados outros classificadores, como o de Regressão Logística, mas que apresentaram erros de compilação devido ao tipo dos nossos dados não se aplicarem a como este é implementado. Para cada classificador, é possível visualizar o relatório de classificação e o sua matriz de confusão, métricas úteis para conferir a performance do modelo instanciado. A mais importante dentre elas é a acurácia, que demonstra em porcentagem quanto dos nomes dos dispositivos foi possível classificar corretamente.

Em seguida, testamos classificadores que envolvem aprendizado profundo, como o *Perceptron*, *Multi-Layer Perceptron* e a construção das nossas próprias redes neurais. Foram usadas diversos tipos de camadas para a implementação destas redes, como camadas densas, de ativação, normalização e *dropout*,

além da variação nos otimizadores, número de iterações e exemplos de treinamento. Apesar dos esforços, os modelos construídos utilizando estes últimos algoritmos de aprendizado profundo não apresentaram uma eficácia satisfatória.

Prosseguindo, na última subseção do trabalho verificamos como os nossos modelos se comportariam quando confrontados com ataques adversariais, que causam uma perturbação na nossa base de dados original justamente para tentar confundir modelos de classificação que foram treinados com os dados inalterados. O algoritmo de ataque escolhido, chamado FGSM, é relativamente novo na literatura, sendo apresentado num artigo de 2014. Nos modelos de *Decision Tree* e *Random Forest*, a eficácia na classificação de dispositivos foi reduzida de forma significativa. Com o *K-Neighbors*, não foi tão expressiva como as anteriores. Ao testar os ataques contra o modelo de redes neurais, a acurácia permaneceu a mesma, o que não deve ser visto como algo positivo nesse caso já que a acurácia inicial não foi um bom resultado. Em trabalhos futuros, seria um bom objeto de estudo o motivo pelo qual os algoritmos de aprendizado profundo apresentaram uma performance ruim neste *dataset*.

O desenvolvimento deste trabalho foi fundamental para estudar mais sobre conceitos de redes, se familiarizar com novas ferramentas que facilitam a gravação e leitura de tráfego de rede (a coleta de tráfego, no início do desenvolvimento da monografia, era uma das principais dúvidas), além de ver como existem diversas formas de tratar dataframes, arquivos e multiprocessamento com a linguagem Python. Outrossim, a aplicação de conhecimentos sobre aprendizado de máquina, inteligência artificial e aprendizado profundo adquiridas durante o curso ajudaram a fixar a prática desta área da computação, havendo também a descoberta de novos tópicos relacionados. Os modelos e ataques adversariais eram conceitos novos, que foram estudados e pesquisados conforme a monografia foi escrita e desenvolvida, indubitavelmente agregando à bagagem de conhecimentos. Não só este último tópico, mas tudo que foi aprendido é de suma importância para a formação acadêmica.

A. Repositório Git

Para aqueles que desejam explorar o código-fonte e obter mais detalhes sobre sua implementação, o repositório do projeto está disponível em: https://github.com/mathelho/msi

Nesse repositório, é possível encontrar os arquivos de código implementados.

Além disso, para uma melhor visualização dos retornos do código e gráficos, é possível também acessar os links dos notebooks.

Notebook criado para verificar o funcionamento da extração de PCAPs: https://colab.research.google.com/drive/1em5SOQUi2jaSvfMYUXSia5PPxTrVybeB?usp=sharing

Notebook criado para implementar os classificadores: https://colab.research.google.com/drive/1ZnlZQl6Y8NUBH9PsdNL9R6baVCfuSFw?usp=sharing

REFERÊNCIAS

[1] Sadeghzadeh, A. M., Shiravi, S., and Jalili, R. (2021). Adversarial network traffic: Towards evaluating the robustness of deep-learning-based network traffic classification. IEEE Transactions on Network and Service Management, 18(2):1962–1976.

[2] Dos Santos, B. V., Vergütz, A., Macedo, R. T., and Nogueira, M. (2023). A dynamic method to protect user privacy against traffic-based attacks on smart home. Ad Hoc Networks, page 103226.

[3] Fotiadou, K., Velivassaki, T.-H., Voulkidis, A., Skias, D., Tsekeridou, S., and Zahari- adis, T. (2021). Network traffic anomaly detection via deep learning. Information, 12(5):215.

[4] Tekerek, A. and Bay, O. (2019). Design and implementation of an artificial intelligence- based web application firewall model. Neural Network World, (4).

[5] Chakraborty, P., Rahman, M. Z., and Rahman, S. (2019). Building new generation firewall including artificial intelligence. International Journal of Computer Applications, 975:8887.

[6] IBM, "O que são redes neurais convolucionais?" ibm.com. https://www.ibm.com/br-pt/topics/convolutional-neural-networks (acesso em: 27 jul. 2024)

[7] Arun George Zachariah, "A Deep Dive into the Fast Gradient Sign Method" medium.com. https://medium.com/@zachariaharungeorge/a-deep-dive-into-the-fast-gradient-sign-method-611826e34865 (acesso em: 28 jun. 2024)

[8] Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.