Análise Exploratória de Ataques por meio de Instruções em Código de Máquina em Automóveis Inteligentes

Bruno H. Labres[1], **Ovídio J. Silva J.**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba - PR - Brasil

{bhl16,ojsj18}@inf.ufpr.br

Resumo. *O uso de IoT em veículos acarreta no aparecimento de novos alvos para exploradores de falhas que podem resultar em acidentes fatais. Esse trabalho realiza uma análise exploratória de ataques que usam instruções em código de máquina em automóveis inteligentes. O objetivo é estudar o desenvolvimento de um classificador para detectar se a instrução é maliciosa.*

1. Introdução

O uso de recursos inteligentes em carros tem aumentado com os avanços tecnológicos, este avanço tem como consequência o aumento de conectividade externas, como servidores para direção autônomas, uso de sensores geográficos e de orientação e etc. Tais funcionalidades abrem brechas para ataques de disponibilidade, que em casos como os veiculares, envolvem a segurança do usuário [Kang et al. 2021]. Esses fatores abrem espaço para uma nova área focada em segurança em IoT automobilístico. Este trabalho estuda o desenvolvimento de um classificador binário para identificação de ataques por instruções em veículos inteligentes.

2. Descrição do problema

Os veículos na sua maior parte se utilizam do protocolo CAN(*Controller Area Network*), um modelo de transmissão de informações através de nós entre microcontroladores e dispositivos sem a necessidade de um computador host, para que tudo ocorra de maneira segura e eficiente as mensagens são divididas no padrão da imagem a seguir:

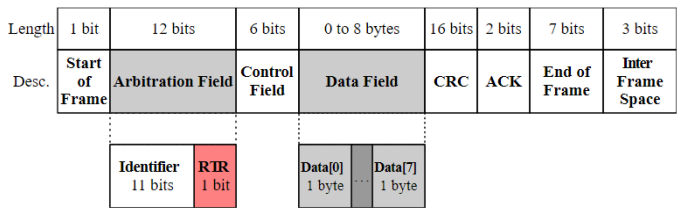

Figure 1. Encapsulamento da mensagem no padrão CAN

Tendo um sistema extremamente regrado, os ataques se utilizam de recursos do modelo de mensagens para quebrar a conexão entre os nós do sistema veicular. Com isso, seria útil identificar possíveis padrões que ocorrem em ataques a esses sistemas.

3. Motivação

Como é uma área relativamente recente, a escassez de pesquisa focada nesse campo se caracteriza como algo urgente, já que ataques nesse nível podem resultar diretamente em danos graves a indivíduos. Com o avanço na área de *IoT*, espera-se que essas pesquisas se tornem mais recorrentes na academia e no mercado.

4. Visão geral do processo

Nesta seção apresentaremos as etapas do estudo.

4.1. Coleta

Os conjuntos de dados foram coletados através de diferentes bancos disponibilizados pelo Hacking and Countermeasure Research Lab [Seo et al. 2018]. Esses dados incluem exemplos de ataques por negação de serviço, fuzzy attack, *spoofing* em RPM e *spoofing* na engrenagem motriz. Os conjuntos de dados foram obtidos através dos logs do tráfico de CAN na porta OBD-II de um veículo enquanto ataques por injeção de mensagem ocorriam.

Os dados foram coletados através de quatro arquivos CSV disponibilizados pelo laboratório, um para cada tipo de ataque.

4.2. Atributos

Os atributos de cada tabela estão organizados no seguinte esquema:

- **Timestamp**: horário em que a instrução foi registrada;
- **CAN ID**: identificador da mensagem CAN em hexadecimal;
- **DLC**: número de bytes de dados, de 0 a 8;
- **DATA[0-7]**: campos de dados (por byte);
- **Flag**: T ou R, T representa mensagem injetada enquanto R representa mensagem normal.

4.3. Pré-processamento

Foram obtidos quatro arquivos CSV, um para cada tipo de ataque e a partir destes obtivemos instruções não-maliciosas. Cada tabela foi transformada em um *dataframe* do Pandas com os campos equivalentes aos das tabelas. A partir disso, embaralhamos os ataques dos quatro tipos e selecionamos aleatoriamente 10.000 deles, além de 10.000 amostras não-maliciosas. Com isso, temos duas classes.

Além disso, os campos de dados foram convertidos de hexadecimal para decimal. Isso facilitará a extração de características na etapa seguinte.

4.4. Extração de características

Para o escopo do projeto, escolhemos fazer a extração de característica com base nos oito campos de dados (cada um representando um byte). Essa escolha foi feita pelo seguinte motivo: os campos de *timestamp* e *CAN ID* podem ser relevantes para identificar ataques, porém para isso seria necessário avaliar a série temporal de amostras, já que são ataques de injeção de mensagens com um certo período ou intervalo de tempo, o que não seria viável no escopo deste projeto. O campo DLC tem em sua maioria instruções de

timestamp	can_id	dlc	data0	data1	data2	data3	data4	data5	data6	data7	type
1.478196e+09	0545	8	d8	00	00	8a	00	00	00	00	0
1.478196e+09	0002	8	00	00	00	00	00	01	07	15	0
1.478196e+09	0153	8	00	21	10	ff	00	ff	00	00	0
1.478196e+09	0130	8	19	80	00	ff	fe	7f	07	60	0
1.478196e+09	0131	8	17	80	00	00	65	7f	07	9f	0
1.478196e+09	0140	8	00	00	00	00	02	20	27	a8	0
1.478196e+09	0350	8	05	20	14	68	78	00	00	21	0
1.478196e+09	02c0	8	15	00	00	00	00	00	00	00	0
1.478196e+09	0370	8	00	20	00	00	00	00	00	00	0
1.478196e+09	043f	8	10	40	60	ff	7d	8c	09	00	0

timestamp	can_id	dlc	data0	data1	data2	data3	data4	data5	data6	data7	type
1.478196e+09	01cd	8	d0	21	ec	f7	6a	d2	da	6e	1
1.478196e+09	0378	8	45	de	26	09	f8	48	11	51	1
1.478196e+09	01e2	8	48	20	7d	e8	62	34	61	7b	1
1.478196e+09	034e	8	73	9e	b9	77	13	e0	e5	23	1
1.478196e+09	0108	8	b9	48	4b	24	a0	35	8f	27	1
1.478196e+09	04e8	8	e7	23	3a	fa	6d	34	f8	8b	1
1.478196e+09	05e1	8	f0	51	41	f2	69	c2	ac	a5	1
1.478196e+09	0102	8	2b	ca	a4	da	e3	42	40	f0	1
1.478196e+09	007c	8	2e	19	7b	ea	ed	46	ae	cf	1
1.478196e+09	0051	8	6f	5e	40	04	e7	ae	d7	b3	1

Figure 2. Amostra de dados de instruções normais (*dataframe* superior) e ataques (*dataframe* inferior).

8 bytes. O campo de flag foi pré-processado na escolha de classes (já que ele indica se é ou não uma mensagem injetada). Com isso, o campo utilizado para a extração de características são os oito campos de dados. A transformação para dados numéricos foi feita no pré-processamento para assim facilitar a extração de características. Seguindo as recomendações em aula, os dados numéricos foram pré-processados de hexadecimais para decimais. O vetor de características possui oito elementos e é obtido a partir dos bytes de dados das instruções CAN.

4.5. Classes

Nós trabalhamos com duas classes: uma de ataques e uma de não-ataques. Ambas as classes são balanceadas e a classe de ataques contém amostras de ataques dos tipos:*DoS*, *fuzzy attack*, *spoofing* na RPM e *spoofing*.

5. Metodologia

5.1. Organização

Os dados foram divididos afim de treinar um modelo binário, para isso foi necessário agrupar toda classe de ataque em apenas uma classe, denominada *"attack"*, em contrapartida os dados que correspondiam a classe *normal* era presente em todas as outras classes com a *key* "R", enquanto as de ataques "T", portanto foi pego uma parcela desse dado de um dos dataset de ataque.

5.2. Balanceamento

Feito o agrupamento, para que não houvesse desbalanceamento das classes foi dividido números iguais de classes por parcela a ser usada, assim como aleatoriedade na escolha de

cada classe que faria parte do conjunto de treinamento e dados, com isso os dados teriam parcelas semelhantes, evitando problemas de *Overfitting* em determinado tipo de ataque. Na figura 2 é possível observar a divisão das classes que serão utilizadas no treinamento.

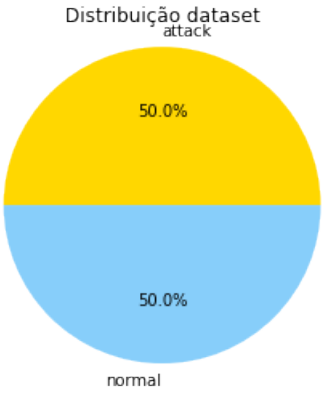

Figure 3. Proporção das classes

5.3. Escolha das características

O dataset é dividido em 12 colunas, *timestamp*, *CAN ID*, *DLC*, oito colunas de dados e uma correspondente ao tipo (ataque injetado ou instrução usual). Para o treinamento foi utilizamos as oito colunas de dados, com o objetivo de verificar se as informações injetadas nesse vetor são suficientes para identificar ataques. Após a extração dessas características foi realizado uma conversão de base no dataset que se encontrava em hexadecimal e foi convertido para decimal, para trabalharmos com os dados em decimais como características ao invés de hexadecimal.

5.4. Validação Cruzada

Para utilizar a validação cruzada, foi necessário o uso da função StratifiedKFold, e, também, como queríamos obter amostragens aleatórias de dados, utilizamos os parâmetros de randomização. Os dados foram separados em cinco pastas, e divididos de forma que 80% das amostras fosse para treino e 20% para testes.

5.5. Treinamento e testes

Foram utilizados três algoritmos de aprendizado de máquina distintos: Floresta aleatória, *k-nearest neighbors* (KNN) e um *multi-layer perceptron*. Com isso, foram obtidas métricas em cada iteração da validação cruzada, como as matrizes de confusão, gráfico de precisão/recall e curvas ROC.

5.5.1. Floreta aleatória

Partindo para o treinamento com o primeiro método, temos **floresta aleatória**, foram obtidas métricas em cada iteração da validação cruzada.

Os parâmetros foram selecionados utilizando GridSearch, e assim treinando de fato com os parâmetros que apresentaram melhor desempenho através desse método. O leque de parâmetros inserido no GridSearch corresponde a:

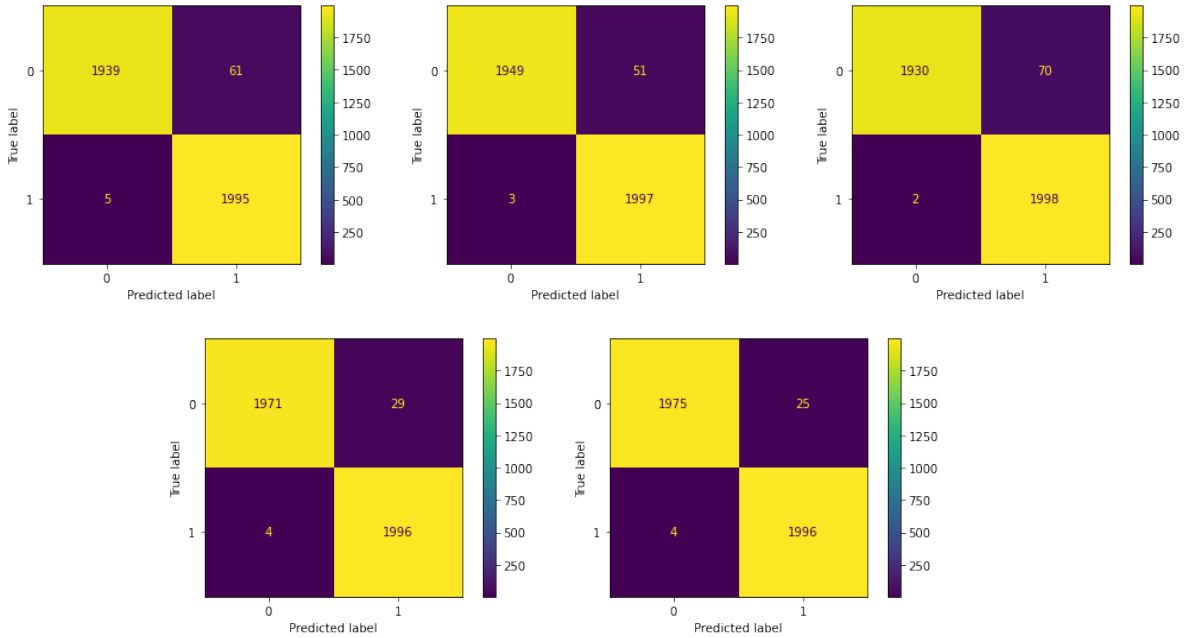

Figure 4. Matrizes de confusão das iterações de validação cruzada para a floresta aleatória.

```
param_grid_rf = {
    "max_depth": [1,2],
    "min_samples_split": [2,3,4],
    "min_samples_leaf": [2,3,4]
}
```

A lista completa de parâmetros utilizados (escolhidos manualmente ou com a ajuda do GridSearch) é:

- n_estimators: 100
- criterion: gini
- max_depth: 2
- min_samples_split: 2
- min_samples_leaf: 2
- min_weight_fraction_leaf: 0.0
- max_features: auto
- max_leaf_nodes: None
- min_impurity_decrease: 0.0
- bootstrap: True
- oob_score: True
- n_jobs: None
- random_state: None
- verbose: 0
- warm_start: None
- class_weight: None
- ccp_alpha: 0.0
- max_samples=None

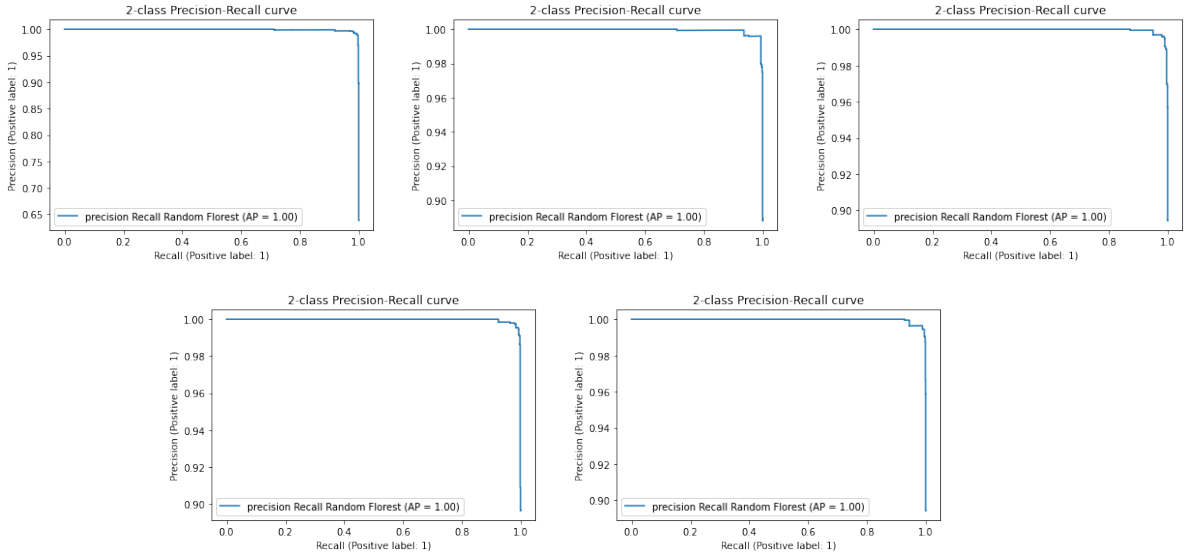

Figure 5. Comparação entre precisão e *recall* para a floresta aleatória em cada iteração de validação cruzada.

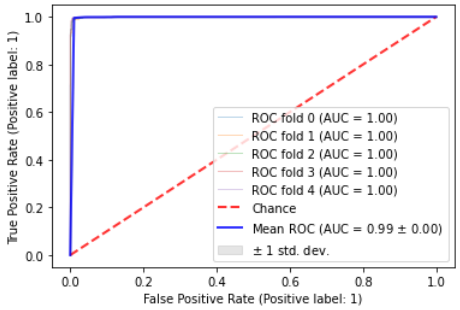

Figure 6. Curva ROC para a floresta aleatória.

5.5.2. *K-Nearest Neighbors*

Os parâmetros foram selecionados utilizando GridSearch, e assim treinando de fato com os parâmetros que apresentaram melhor desempenho através desse método. O leque de parâmetros inserido no GridSearch corresponde a:

```
param_grid_knn = {
    "n_neighbors": [2,3,5,7,9],
    "weights": ['uniform','distance'],
    "algorithm": ['ball_tree', 'kd_tree', 'brute']
}
```

A lista completa de parâmetros utilizados (escolhidos manualmente ou com a ajuda do GridSearch) é:

- n_neighbors: 9
- weights: uniform
- algorithm: ball_tree

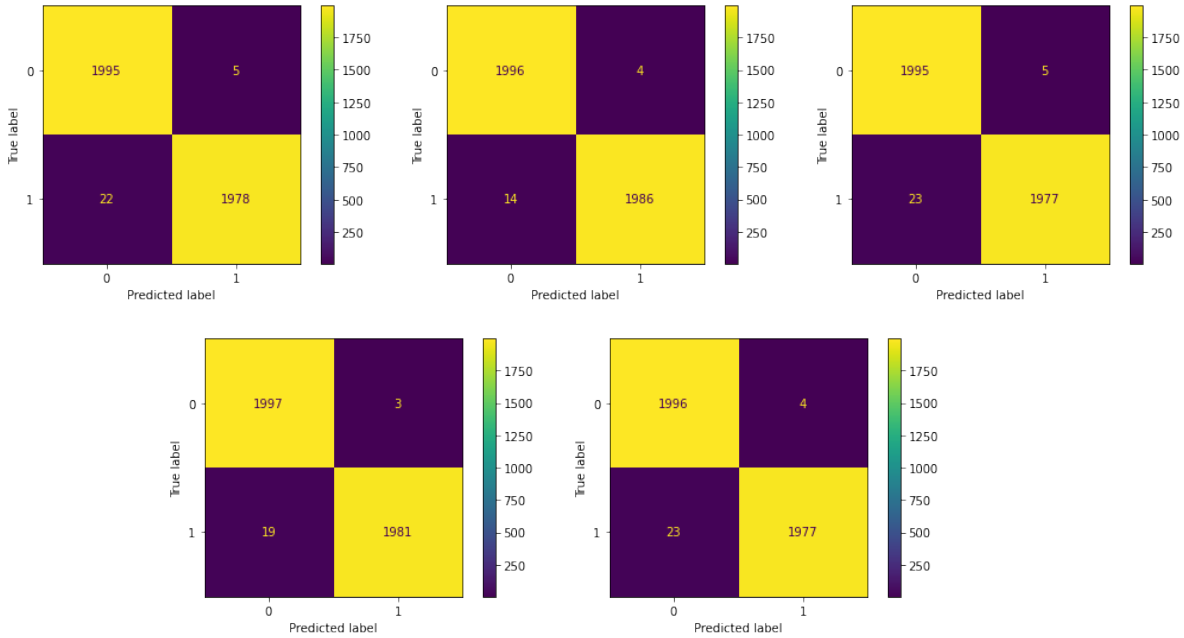

Figure 7. Matrizes de confusão das iterações de validação cruzada para KNN.

- leaf_size: 30
- p: 2
- metric: minkowski
- metric_params: None
- n_jobs: None

5.5.3. *Multilayer Perceptron*

Os parâmetros foram selecionados utilizando GridSearch, e assim treinando de fato com os parâmetros que apresentaram melhor desempenho através desse método. O leque de parâmetros inserido no GridSearch corresponde a:

```
param_grid_mlp = {
    'hidden_layer_sizes': [(10,10,10), (10,10), (10,)],
    'activation': ['logistic', 'tanh'],
}{'activation': 'logistic', 'hidden_layer_sizes': (10,)}
```

A lista completa de parâmetros utilizados (escolhidos manualmente ou com a ajuda do GridSearch) é:

- hidden_layer_sizes: (10,)
- activation: logistic
- solver: adam
- alpha: 0.0001
- batch_size: auto
- learning_rate: constant
- learning_rate_init: 0.001

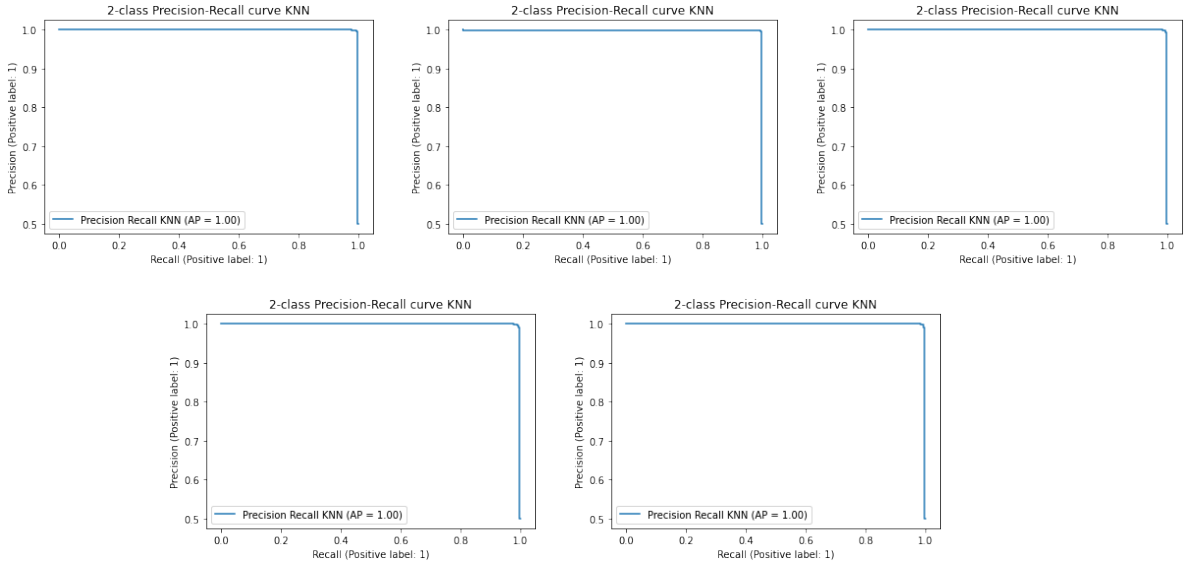

Figure 8. Comparação entre precisão e *recall* para KNN em cada iteração de validação cruzada.

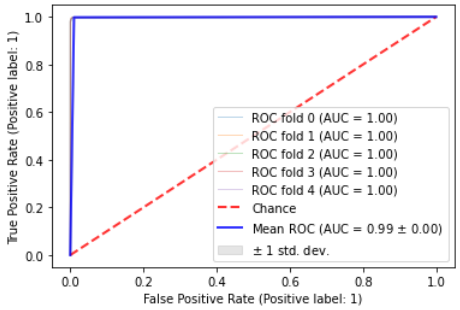

Figure 9. Curva ROC para KNN.

- power_t: 0.5
- max_it: 200
- shuffle: True
- random_state: None
- tol: 1e-4
- verbose: False
- warm_start: False
- momentum: 0.9
- nesterovs_momentum: True
- early_stopping: False
- validation_fraction: 0.1
- beta_1: 0.9
- beta_2: 0.999
- epsilon: 1e-8
- n_iter_no_change: 10
- max_fun: 15000

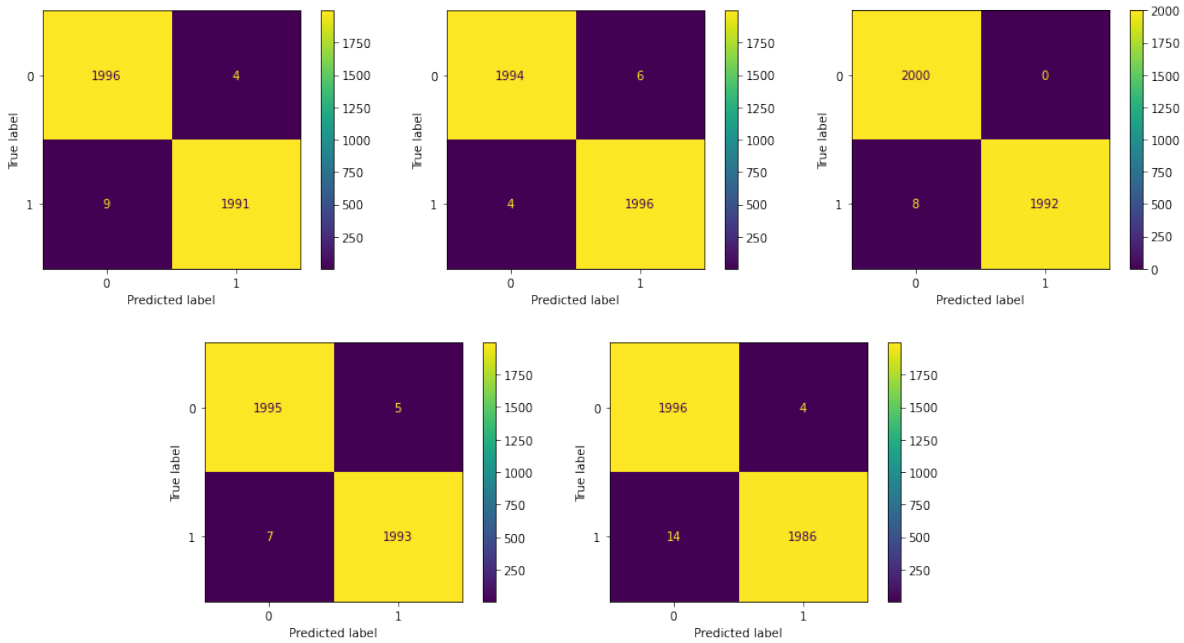

Figure 10. Matrizes de confusão das iterações de validação cruzada para MLP.

6. Discussão

Os resultados acabaram sendo excelentes, o que nos fez verificar se algoritmos de aprendizado de máquina seriam de fato a melhor opção para a resolução deste problema. Isso acontece pois, como podemos notar na Figura 2, as instruções normais possuem campos de dados muito mais definidos do que os de ataque. Enquanto os campos de ataque são preenchidos com valores aleatórios ou zerados, as instruções normais tem comportamento mais previsível e comportado em relação ao seu vetor de dados. Isso faz com que os algoritmos de aprendizado de máquina possuam ótimos resultados, porém que talvez seu uso seja desnecessário, já que um método mais simples ou computacionalmente mais baratos poderiam ser utilizados. Porém, caso seja desejado permanecer na área de aprendizado de máquina, entre os próximos passos podemos levar em conta o comportamento temporal de injeção de instruções maliciosas e utilizar outros algoritmos, como rede neurais recorrentes.

Levando isso em conta, a dificuldade encontrada nos três algoritmos foi de controlar seu **overfitting**, já que as taxas de acurácia apresentavam valores altos. Com a floresta aleatória, por exemplo, nós temos uma tendência natural de florestas e árvores resultarem em overfitting. Por isso, ajustamos seus parâmetros para manter sua profundidade baixa, com baixo número de *splits* e número de amostras por folha. Isso foi feito pra que as classificações generalizassem mais as amostras, ao invés de gerar overfitting. No *K-Nearest Neighbors*, exploramos as métricas envolvendo a criação da árvore de vizinhos. Isto é, os algoritmos de criação de árvores, o número de vizinhos usados na classificação e os pesos. A melhor escolha de parâmetros foi com nove vizinhos, o que dá indícios de que o problema em questão tem tendência a dar overfitting, já que com menos vizinhos a precisão já é excelente. No multi-layer perceptron, uma única camada com dez neurônios foi o suficiente para apresentar ótimos resultados. Além disso, entre as opções de função de ativação consideramos alguns tipos de função sigmoide, já que são indicadas

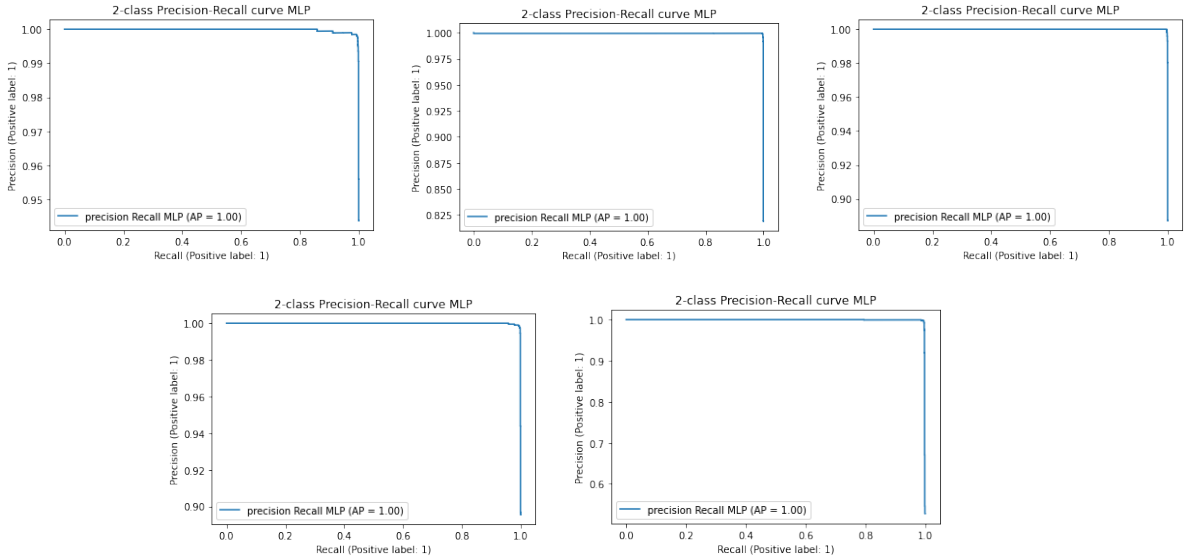

Figure 11. Comparação entre precisão e *recall* para MLP em cada iteração de validação cruzada.

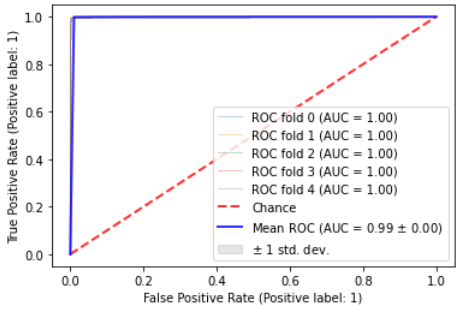

Figure 12. Curva ROC para KNN.

para classificação binária. O tipo que apresentou melhor resultado foi a logística.

7. Reproducibilidade

- Link para o repositório com os scripts: car-hack-study
- Link para o conjunto de dados: CAN-intrusion-dataset

8. Conclusão

A área de segurança em IoT em veículos inteligentes está em ascenção e deve apresentar uma evolução cada vez maior nas próximas décadas. Neste trabalho, realizamos experimentos com ataques injetados como instruções CAN em automóveis. Dessa forma, identificamos que as instruções normais seguem padrões mais definidos em seus vetores de dados do que os ataques, que acabam sendo valores aleatórios ou zerados nesses campos. Devido à isso, a classificação utilizando essas características foi fácil e os resultados foram excelentes, porém, isso mostra que esse problema poderia ser resolvido por algoritmos mais simples do que os de aprendizado de máquina.

References

[Kang et al. 2021] Kang, H., Kwak, B. I., Lee, Y. H., Lee, H., Lee, H., and Kim, H. K. (2021). Car hacking: Attack defense challenge 2020 dataset.

[Seo et al. 2018] Seo, E., Song, H. M., and Kim, H. K. (2018). Gids: Gan based intrusion detection system for in-vehicle network. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, pages 1–6.