Operating System Detection

Issa Almadani

Dec 10, 2023

CONTENTS:

CHAPTER

ONE

PROJECT INTRODUCTION

1.1 Introducton

In this final project, I have decided to recreate and expand on the results from the original nPrintML paper, specifically the OS Detection section. In the original paper, nPrint, a tool used to parse network traces (PCAP) files into bit representations, is used to train ML models to learn to utilize the nPrint feature input to predict the operating system from which the packet had originated. The input data is thus read in and transformed via the tool, and then passed onto the model in groups of packets to train and test against.

For this project, I draw on this primary aspects from the original paper. Although, I have decide to forgo using the nPrint / nPrintML tools, and instead use Scapy, a python module for reading PCAP files, in order to directly parse the PCAP files myself, which I then convert to dataframes, and manipulate directly. I have decide to adopt this approach to have better control over the input data, data processing, and to be able to test engineering features and dropping other features to assist in model training.

Various ML models will be used to train on the input data in order to test which models are best capable for learning to predict the OS. The models will be primarily trained a per packet basis, rather than groups of packets like the original paper and other tools. The best model will then be taken and trained on groups to test if this can enhance OS predictions. This will extrapolate well to real-life, as packet traces can include multiple packets from a specific machine when attempting to predict that machine's operating system.

CHAPTER

TWO

DATA PROCESSING AND INITIALIZATION

2.1 Operating System Labels

The dataset used will be from the "Intrusion Detection Evaluation Dataset (CIC-IDS2017)" dataset, also the dataset used in the orignal paper. This dataset includes network traces that can be divided into groups of packets based on IP address. The dataset also provides the list of operating systems that each IP address maps to, which is used to generate the labels in this project.

The following OS labels are used in training:

```
# each unique IP address is mapped to the operating system, which is taken
# from the Intrusion Detection Evaluation Dataset (CIC-IDS2017) mapping

# check how many unique IP addresses there are
np.unique(df["ip_src"])

# map the IP addresses to the OS from the original dataset
ip_to_os = {
    "192.168.10.12": "Ubuntu 16.4, 64B",
    "192.168.10.14": "Win 10, pro 32B",
    "192.168.10.15": "Win 10, 64B",
    "192.168.10.16": "Ubuntu 16.4, 32B",
    "192.168.10.17": "Ubuntu 14.4, 64B",
    "192.168.10.19": "Ubuntu 14.4, 32B",
    '192.168.10.25': "MAC",
    '192.168.10.5': "Win 8.1, 64B",
    '192.168.10.50': "Web server 16 Public",
    '192.168.10.51': "Ubuntu server 12 Public",
    '192.168.10.8': "Windows Vista",
    '192.168.10.9': "Win 7 Pro, 64B",
    '205.174.165.73': "Kali"
}

df["OS"] = df['ip_src'].map(ip_to_os)

df
```

Namely, the operating systems are Linux-based (Ubuntu), Windows, and MacOS. The operating systems are actually fine-tuned to specific versions, as recommended as a more challenging approach by the paper. Using such fine-tuned OS detection is important as it can show that ML models are able to learn fine-tuned OS detection to even the version from a given dataset.

These labels where subsequently encoded by the scikit-learn LabelEncoder() class, and then used to train our model:

```python
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(labels)

y_encoded
```

2.2 PCAP Data

As stated in the introduction, I decided to manually load and parse in the data from the PCAP file. This file contained ~250 MB of data, specifically around 1,200,000 packets. Loading is performed in a lazy manner using the PcapReader() class offered by the Scapy module. I specifically, decide to write a helper get_headers function that performs this processing. This part of the data loading was the bottleneck for the notebook as loading in the data takes about 10 minutes to perform. After performed, the conversion to a pandas dataframe, and parsing the data takes relatively little time.

```python
def get_headers(pcap_path, limit = None, print_on = 10000):
    '''
        extracts the IP / TCP headers from the given pcap file.

            Parameters:
                    pcap_path (str): A string file path to the PCAP input file.
                    limit (Optional) (int): Integer limit on number of packets to read␣
↪in.
                    print_on (Optional, default = 10000) (int): Print time on which load␣
↪intervals.

            Returns:
                    headers (list): list of ip / tcp header dictionaries
    '''
    # time logging
    start_time = time.time()
    headers = []

    # Lazy read PCAP file
    packets = PcapReader(pcap_path)
    for i, packet in enumerate(packets):

        # break on limit, log on interval
        if (limit and i >= limit): break
        if (i and not i % print_on): print(f"Loaded {i} headers in {time.time() - start_
↪time} seconds.")
        header = {}

        if IP in packet:
            # Read the following IP headers:
            header = {
                "ip_version": packet[IP].version,
                "ip_ihl": packet[IP].ihl,
                "ip_tos": packet[IP].tos,
                "ip_tot_len": packet[IP].len,
```

(continues on next page)

(continued from previous page)

```
                "ip_flags": packet[IP].flags,
                "ip_frag_ofs": packet[IP].frag,
                "ip_ttl": packet[IP].ttl,
                "ip_protocol": packet[IP].proto,
                "ip_checksum": packet[IP].chksum,
                "ip_src": packet[IP].src,
                "ip_dst": packet[IP].dst,
            }

        if TCP in packet:
            # Read the following TCP headers:
            header |= {
                "tcp_src": packet[TCP].sport,
                "tcp_dst": packet[TCP].dport,
                "tcp_seq_num": packet[TCP].seq,
                "tcp_ack_num": packet[TCP].ack,
                "tcp_ofs": packet[TCP].dataofs,
                "tcp_flags": packet[TCP].flags,
                "tcp_winsize": packet[TCP].window,
                "tcp_checksum": packet[TCP].chksum,
                "tcp_urgptr": packet[TCP].urgptr,
            }

        headers.append(header)
```

2.3 Further Processing

To further process the data and prep it for training, I decide to drop certain columns and perform type casting to ensure all data types are numeric for training:

```
clean_df = df.drop(["ip_version", "ip_ihl", "ip_src", "ip_frag_ofs", "ip_protocol", "ip_
↪dst",
                    "tcp_src", "tcp_dst", "tcp_seq_num", "tcp_ack_num", "tcp_urgptr", "OS
↪"], axis = 1)
labels = df["OS"]

clean_df["ip_flags"] = clean_df['ip_flags'].apply(lambda x: int(x))
clean_df["tcp_flags"] = clean_df['tcp_flags'].apply(lambda x: int(x))
```

First, it is important to note that the columns “ip_src”, “ip_dst”, “tcp_src”, “tcp_dst”, “tcp_seq_num”, and “tcp_ack_num” were all dropped from the dataset. This was suggested and performed by the original paper, which I also did to ensure that the models do not learn extraneous features that are not relevant and only represent the topography of the network.

Additionally, I decided to also drop “ip_version”, “ip_ihl”, “ip_frag_ofs”, and “ip_protocol” after studying the dataset and noticing that all these values were exactly the same regardless of OS in all packets. So, to facilitate efficiency in training, I removed this additional data as there are no features that will be learnable from them.

Finally, I also dropped the “OS” column after previously storing and encoding it in a label column from the previous section. This is to ensure that the model does not have access to the label column when training.

Secondly, I made sure to cast the flags columns to integers in order to pass in numeric values in all columns to the models.

CHAPTER

THREE

MODEL TRAINING

3.1 Test / Train Split

I have decided to use a simple test / train split on the dataset, with 20% of the dataset being used for testing.

```
X_train, X_test, y_train, y_test = train_test_split(clean_df, y_encoded, test_size=0.2,␣
↪random_state=42)
```

3.2 Model 1: Random Forest

The first model I chose was a random forest classifier, with the number of estimators being set to 50. This model takes around 3 minutes to train, and produces a very accurate prediction.

```
random_forest = RandomForestClassifier(n_estimators=50)
random_forest.fit(X_train, y_train)
```

3.3 Model 2: MLP Classifier

The second model was chosen to be an MLP classifier. The number of hidden layers was varied, but ultimately had no effect on training accuracy, and accuracy actually dropped with higher numbers of layers. So, the default hidden layer was used: (100,). Training takes a long time to finish, but the model automatically stops at around 50 iterations (at convergence).

```
mlp = MLPClassifier(verbose=True)
mlp.fit(X_train, y_train)
```

3.4 Model 3: Decision Tree Classifier

Finally, the last model was a decision tree classifier. This model prodcued the best results in a relatively short time.

```
clf = DecisionTreeClassifier()
clf = clf.fit(X_train, y_train)
```

CHAPTER

FOUR

MODEL EVALUATION AND TUNING

4.1 Evaluation Methods

The primary evaluation methods I used were accuracy scores and the built-in skikit-learn classification report function, which performs automatic calculations with various values such as precision, recall, f-score…etc.

For the random forest, and decision tree models, I was able to plot the feature importance plots, and visualize which features the model learned from and associated with operating system. This was not possible with MLP as that model does not calculate feature importance in a direct manner.

This was the general code structure used for evaluation:

```python
# Calculate Accuracy Directly
y_pred = MODEL.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(accuracy)

# Plot Learned Feature Importance (NOT FOR MLP)
feature_importance = MODEL.feature_importances_
feature_names = ["ip_tos", "ip_tot_len", "ip_flags", "ip_ttl", "ip_checksum",
                "tcp_ofs", "tcp_flags", "tcp_winsize", "tcp_checksum"]
plt.figure(figsize=(10, 6))
plt.barh(range(len(feature_importance)), feature_importance, align="center")
plt.yticks(range(len(feature_importance)), feature_names)
plt.xlabel("Feature Importance")
plt.show()

# Print Out Full Classification Report
report = classification_report(y_test, y_pred)
report
```

4.2 Model 1: Random Forest

For the random forest accuracy, it was calculated to be 80.3%, which indicates that this model was very successful in being able to determine the fine-tuned operating systems from the given packet.

The classification report also produced the given metrics:

```
          precision    recall  f1-score   support

```

(continues on next page)

(continued from previous page)

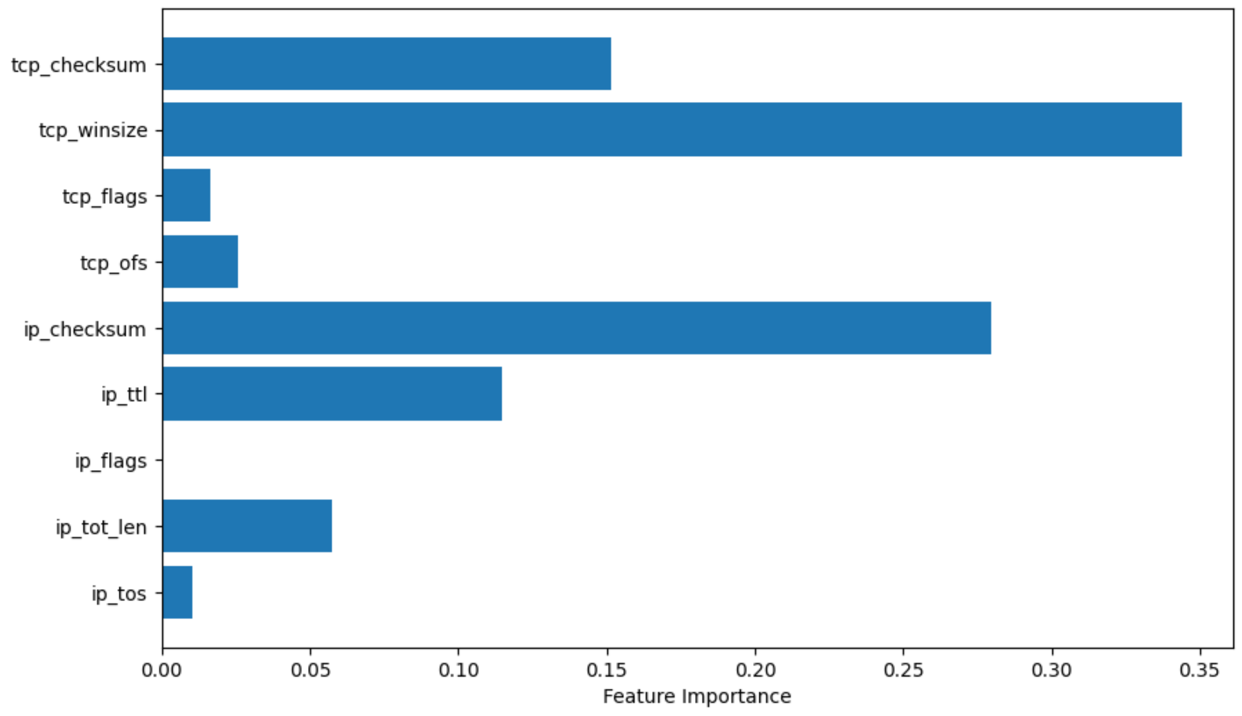
```
       0       1.00      1.00      1.00      8702
       1       0.98      0.98      0.98     20115
       2       0.61      0.63      0.62     19993
       3       0.61      0.63      0.62     19982
       4       0.66      0.65      0.66     20126
       5       0.65      0.63      0.64     19925
       6       0.61      0.60      0.61     19863
       7       0.96      0.97      0.96     19933
       8       0.96      0.96      0.96     20172
       9       0.94      0.92      0.93     20090
      10       0.81      0.80      0.80     19987
      11       0.81      0.85      0.83     19788
      12       0.94      0.94      0.94     20104

accuracy                           0.80    248780
macro avg      0.81      0.81      0.81    248780
weighted avg   0.80      0.80      0.80    248780
```

Here, indices 0 - 12 are the encode labels, which are the following labels:

```
Kali: 0
MAC: 1
Ubuntu 14.4, 32B: 2
Ubuntu 14.4, 64B: 3
Ubuntu 16.4, 32B: 4
Ubuntu 16.4, 64B: 5
Ubuntu server 12 Public: 6
Web server 16 Public: 7
Win 10, 64B: 8
Win 10, pro 32B: 9
Win 7 Pro, 64B: 10
Win 8.1, 64B: 11
Windows Vista: 12
```

This indicates that the model was having the hardest time making precise predictions with linux-based machines, but was a lot more successful in other types of operating systems.

Finally, for the feature importance, the following was calculated:



This indicates that the checksum for both ip / tcp headers and the winsize feature were the most important features in determining the operating system.

Overall, this model was very successful in finding trends in these features and using them to make accurate predictions.

4.3 Model 2: MLP Classifier

The second model was evaluated similarly based on accuracy and the classification report.

```
              precision    recall  f1-score   support

           0       0.44      0.70      0.54      8702
           1       0.63      0.96      0.76     20115
           2       0.00      0.00      0.00     19993
           3       0.11      0.00      0.00     19982
           4       0.18      0.73      0.29     20126
           5       0.37      0.05      0.09     19925
           6       1.00      0.00      0.00     19863
           7       0.50      0.47      0.48     19933
           8       0.45      0.92      0.60     20172
           9       0.31      0.01      0.02     20090
           10       0.55      0.04      0.08     19987
           11       0.48      0.71      0.57     19788
           12       0.73      0.98      0.83     20104

    accuracy                           0.42    248780
macro avg       0.44      0.43      0.33    248780
weighted avg       0.44      0.42      0.32    248780
```

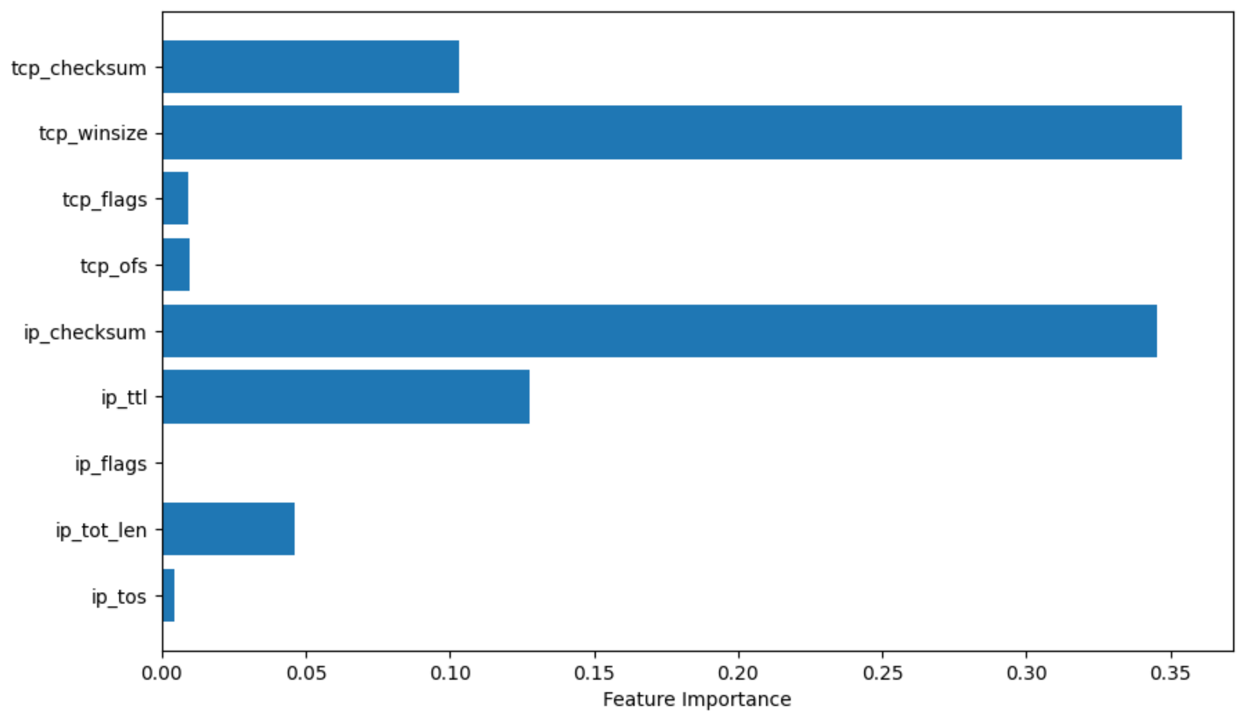
Overall, this was a very poor model as it had a very low accuracy of around 42%. The metrics above also shwocase that it is very poor in making predictions as well, which is why it was not considered as a plausible choice for the model to use in the final section.

4.4 Model 3: Decision Tree Classifier

This was the most successful model in making predictions about the operating system. The calculated metrics are as follows:

```
          precision    recall  f1-score   support

       0       1.00      1.00      1.00      8702
       1       0.99      0.97      0.98     20115
       2       0.71      0.71      0.71     19993
       3       0.71      0.71      0.71     19982
       4       0.72      0.72      0.72     20126
       5       0.70      0.70      0.70     19925
       6       0.70      0.69      0.69     19863
       7       0.95      0.97      0.96     19933
       8       0.96      0.96      0.96     20172
       9       0.93      0.93      0.93     20090
      10       0.81      0.79      0.80     19987
      11       0.82      0.83      0.83     19788
      12       0.94      0.94      0.94     20104

    accuracy                           0.83    248780
macro avg       0.84      0.84      0.84    248780
weighted avg    0.83      0.83      0.83    248780
```

The overall accuracy was around 83% and the ubuntu precision predictions are a lot higher than the first model. The feature importance plot is also as follows:



This feature importance plot also highlights the same trend of focusing on the checksum and winsize. More specifically, however, it further lowers the importance on certain features a lot more than the previous model, for instance the tcp flags and offset have the lowest importance now, and the ip_tos and ip_flags as well.

Based on this, I have decide to choose this model to try to further tune, by eliminating these features and retraining on the most important ones. I have also decided to perform the training on groups of 100 as well in order to see if that is able to further boost the accuracy as well.

4.5 Extension

I performed two additional tests to see what effect they will have on model accuracy based on the previous section. First of all, I took the third model and removed the extraneous features listed above from it to see what the accuracy scores would be:

```
          precision    recall  f1-score   support

         0       1.00      1.00      1.00      8702
         1       0.98      0.97      0.98     20115
         2       0.72      0.72      0.72     19993
         3       0.72      0.71      0.72     19982
         4       0.73      0.74      0.73     20126
         5       0.71      0.71      0.71     19925
         6       0.71      0.70      0.71     19863
         7       0.95      0.97      0.96     19933
         8       0.96      0.97      0.96     20172
         9       0.93      0.93      0.93     20090
        10       0.82      0.80      0.81     19987
        11       0.83      0.84      0.84     19788
        12       0.94      0.94      0.94     20104
```

(continues on next page)

(continued from previous page)

```
    accuracy                           0.84    248780
   macro avg       0.85      0.85      0.85    248780
weighted avg       0.84      0.84      0.84    248780
```

Although there was slight improvement actually, and this wasn't an important change to make.

The second change I made was trying to train on groups of 100. First, I grouped the dataset based on the source IP, and then divided each group into chunks of 100 entries:

```python
chunks_df = []
chunk_labels = []

for name, group in df.groupby("ip_src"):
    chunks_df.extend([group.iloc[i:i + 100] for i in range(0, len(group), 100)])
    chunk_labels.extend([np.unique(group.iloc[i:i + 100]["OS"]) for i in range(0,␣
↪len(group), 100)])
```

I then trained the decision tree classifiier on this new chunk dataset after flattening each set into a 2D array:

```python
# remove extraneous columns from input data
# flatten 3d dataframe to a 2d array in order to pass in to training
clean_chunks = []
for chunk in chunks_df:
    clean_chunks.append(chunk.drop(["ip_src", "ip_dst", "tcp_src", "tcp_dst", "tcp_ack_
↪num", "tcp_seq_num", "OS", "ip_flags", "tcp_flags"], axis = 1).to_numpy().reshape(-1))

clean_chunks = np.array(clean_chunks)

# encode labels for training
encoded_chunk_labels = label_encoder.fit_transform(chunk_labels)

X_train, X_test, y_train, y_test = train_test_split(clean_chunks, np.array(encoded_chunk_
↪labels), test_size=0.2, random_state=42)

# I chose to train a third Decision Tree Classifier as it was the best peforming model␣
↪from the previous tests.
chunks_clf = DecisionTreeClassifier()
chunks_clf.fit(X_train, y_train)

# accuracy calculation
y_pred = chunks_clf.predict(X_test)
accuracy = accuracy_score(np.array(y_test), y_pred)
print(accuracy)
```

However, the accuracy ended up being only 52%, a regression from when packets are treated seperately, which could indicate that global trends from the same operating system do not exist to an appreciable extent, or maybe that the work should be furthered with other more complex models to learn from this data.

CHAPTER FIVE

CONCLUDING REMARKS

Overall, this project shows that it is possible to train a model to make accurate predictions on operating system from just the IP/TCP headers alone. Specifically, models such as random forest or tree based classifying models are very effective in performing this prediction, much more than other model types or even neural networks.

These models can be trained relatively quickly and are able to quickly produce a prediction when used later. Moreover, it is revealed that the most important features to consider in this prediction are the checksum and winsize features.

In my initial proposal, I hoped to try to engineer new features or even include some payload bytes in this prediction. However I later decided against doing so after careful analysis of the data, especially when I noticed that the top payload bytes can sometimes include topographical information such as restating the source / destination address or the ack number…etc. So, uncontrolled addition of those bytes can introduce extraneous information that can corrupt the training output. So, I decided to refocus feature generation on simply having more control to choose and remove features from the dataset, which turned out to be a more powerful tool in training situations such as this.

I also hoped to test whether or not further removing extraneous columns would have an impact on training, although it did slightly, the improvement was not too substantial.

Finally, even when considering chunks instead of just one packet at a time, the models I used did not learn well or better than just at one packet at time.

Ultimately, the tree decision classifier was the best model to use in predictions like this.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER SEVEN

CITATION

- Feamster, Nick. Holland, Jordan. Schmitt, Paul, and Mittal, Prateek. "New Directions in Automated Traffic Analysis." (2021): 8 - 10. (https://arxiv.org/pdf/2008.02695.pdf)
- "Intrusion Detection Evaluation Dataset (CIC-IDS2017)." *Canadian Institute for Cybersecurity*, 2017. https://www.unb.ca/cic/datasets/ids-2017.html (accessed Nov 5, 2023).