IoT Malware Traffic Analysis - Visualizations

Dataset used: https://www.kaggle.com/datasets/agungpambudi/network-malware-detection-connection-analysis

This dataset describes network traffic which has been flagged by IoT malware detection systems. Important attributes for the data include the timestamp, uid (unique identifier), source IP address and port, destination IP address and port, and the label of the connection.

In this exercise, I perform the following:

- Download a dataset from Kaggle about network traffic and upload it to Colab.
- Prepare the data by organizing the data into columns utilizing the file's unique delimitter (|), performing cleaning tasks such as filling null values and irrelevant columns, and concatenating multiple dataframes into one.
- Perform EDA on the dataset.
- Create 5 different visualizions, answering different relevant questions about the data.
- Suggest next steps for the project.

Loading the Data

The first step is to load the data and the necessary libraries into the project.

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Importing the files
df_1 = pd.read_csv('/content/CTU-IoT-Malware-Capture-1-1conn.log.labeled.csv')
df_2 = pd.read_csv('/content/CTU-IoT-Malware-Capture-3-1conn.log.labeled.csv')
df_3 = pd.read_csv('/content/CTU-IoT-Malware-Capture-8-1conn.log.labeled.csv')
df_4 = pd.read_csv('/content/CTU-IoT-Malware-Capture-9-1conn.log.labeled.csv')
```

```
df_1.sample(100)
```

	ts|uid|id.orig_h|id.orig_p|id.resp_h|id.resp_p|proto|service|duration|orig_bytes|resp_bytes|conn_state|local_orig|local_resp|mi
553809	
746485	
958013	
426052	
978010	
...	
942816	
725710	
766054	
885467	
761172	

100 rows × 1 columns

Preparing/Cleaning the Data

The next step is to prepare the data to be used for analysis. I begin by splitting the original messy column into the rest of the columns in the df.

```
# Rename the existing column to 'column_name'
df_1 = df_1.rename(columns={df_1.columns[0]: 'all_data'})
# Organize the data into correct columns using the delimitter to separate values
df_1[['ts', 'uid', 'id.orig_h', 'id.orig_p', 'id.resp_h', 'id.resp_p', 'proto', 'service', 'duration', 'orig_bytes', 'resp_bytes', 'conn_sta
```

```
df_2 = df_2.rename(columns={df_2.columns[0]: 'all_data'})
df_2[['ts', 'uid', 'id.orig_h', 'id.orig_p', 'id.resp_h', 'id.resp_p', 'proto', 'service', 'duration', 'orig_bytes', 'resp_bytes', 'conn_sta

df_3 = df_3.rename(columns={df_3.columns[0]: 'all_data'})
df_3[['ts', 'uid', 'id.orig_h', 'id.orig_p', 'id.resp_h', 'id.resp_p', 'proto', 'service', 'duration', 'orig_bytes', 'resp_bytes', 'conn_sta

df_4 = df_4.rename(columns={df_4.columns[0]: 'all_data'})
df_4[['ts', 'uid', 'id.orig_h', 'id.orig_p', 'id.resp_h', 'id.resp_p', 'proto', 'service', 'duration', 'orig_bytes', 'resp_bytes', 'conn_sta


# Dropping the original non-organized column
df_1.drop('all_data', axis=1, inplace=True)

df_2.drop('all_data', axis=1, inplace=True)

df_3.drop('all_data', axis=1, inplace=True)

df_4.drop('all_data', axis=1, inplace=True)
```

```
df_2.sample(100)
```

	ts	uid	id.orig_h	id.orig_p	id.resp_h	id.resp_p	proto	service	duration	orig_bytes	.
29259	1526780671.525698	CpDe1C4WgNyXmVZik9	192.168.2.5	55232	32.187.83.123	22	tcp	-	-	-	
56211	1526802797.932574	CIfkFB1HhrpGwtWUE8	192.168.2.5	44299	144.22.77.58	22	tcp	ssh	3.504774	589	
88540	1526830412.313011	CUQkDzilNTNtwiyR3	192.168.2.5	41760	144.47.16.90	22	tcp	-	2.997004	0	
41007	1526789383.821168	CnZjtx2KT6P5RYpWV7	192.168.2.5	50153	141.237.26.3	22	tcp	-	2.997910	0	
37903	1526787949.658866	C8pSWT1xo0KFdP3V6d	192.168.2.5	43935	141.129.91.222	22	tcp	-	-	-	
...	...	...	...	...	...	...	...	...	...	...	
8741	1526764543.950679	C4t77V1HfEbcvQ6lD2	192.168.2.5	49498	69.170.1.97	22	tcp	-	-	-	
45295	1526794621.99879	CIXKs83RPhRZiHsSO1	192.168.2.5	34844	200.168.87.203	59353	tcp	-	-	-	
103789	1526843790.756593	CRamrh05rcKmKSCkf	192.168.2.5	58991	191.108.158.85	22	tcp	-	2.996870	0	
146509	1526876054.180572	C0c5QW1wHumFClW0ni	192.168.2.5	33454.0	203.169.196.80	22.0	tcp	-	-	-	
37466	1526787749.350807	CK2rNd1V8UetR24JWl	192.168.2.5	36711	141.30.53.83	22	tcp	-	-	-	

100 rows × 23 columns

```
# Concatenating all of the dfs into one document
concat_df = pd.concat([df_1, df_2, df_3, df_4])
```

```
concat_df.sample(100)
```

	ts	uid	id.orig_h	id.orig_p	id.resp_h	id.resp_p	proto	service	duration	orig_t
1054519	1532535498.998834	CcoI3o4ezmA3lJFTR2	192.168.100.111	18310	147.32.209.199	23	tcp	-	-	
30278	1525890966.012766	C9OyJxXzBG4B78CFb	192.168.100.103	39107	180.249.27.252	8080	tcp	-	0.386047	
827517	1532532751.006439	CNLGzw1w6W6W1BUDJe	192.168.100.111	11895	60.226.215.112	81	tcp	-	-	
129118	1525928285.002026	CCipLCKQiEiVFNAV8	192.168.100.103	38880	171.69.236.117	23	tcp	-	2.998548	
573224	1532529890.00103	CM9y1i47sHQOLGTYW	192.168.100.111	2147	175.230.255.18	23	tcp	-	-	
...	...	...	...	...	...	...	...	...	...	
507998	1532529298.003931	CB7B5x2WKclqBzftdf	192.168.100.111	17004	155.67.67.137	81	tcp	-	-	
1181324	1532536817.99893	C0IGd91HnOUsLuC056	192.168.100.111	22791	147.32.88.95	23	tcp	-	-	
889786	1532533398.001656	CVnlJI1QZ5bTwthlZf	192.168.100.111	53622	147.32.165.121	23	tcp	-	-	
146740	1526876388.642317	Ct72VS2rK43vVUY7Q5	192.168.2.5	59958.0	200.168.87.203	59353.0	tcp	-	2.998402	
743022	1526168153.017708	CfghpN3mtPYR3S6iK5	192.168.100.103	43763	134.209.179.234	20033	udp	-	-	

100 rows × 23 columns

```
# changing the columns to the correct dtypes

# Timestamp of connection
concat_df['ts'] = pd.to_datetime(concat_df['ts'], unit = 's')
# Unique ID
concat_df['uid'] = concat_df['uid'].astype(str)

# Source IP address
concat_df['id.orig_h'] = concat_df['id.orig_h'].astype(str)
# Source Port Used
concat_df['id.orig_p'] = pd.to_numeric(concat_df['id.orig_p'], errors='coerce').astype('Int64')

# Destination IP address
concat_df['id.resp_h'] = concat_df['id.resp_h'].astype(str)
# Destination port
concat_df['id.resp_p'] = pd.to_numeric(concat_df['id.resp_p'], errors='coerce').astype('Int64')

# Protocol
concat_df['proto'] = concat_df['proto'].astype(str)
# Service used
concat_df['service'] = concat_df['service'].astype(str)
# Duration of connection
concat_df['duration'] = concat_df['duration'].replace('-', np.nan).astype(float)

# Bytes sent from source
concat_df['orig_bytes'] = pd.to_numeric(concat_df['orig_bytes'], errors='coerce').astype('Int64')
# Bytes sent back from dest to source
concat_df['resp_bytes'] = pd.to_numeric(concat_df['resp_bytes'], errors='coerce').astype('Int64')

# Connection state
concat_df['conn_state'] = concat_df['conn_state'].astype(str)
# Whether origin is local
concat_df['local_orig'] = concat_df['local_orig'].astype(str)
# Whether destination is local
concat_df['local_resp'] = concat_df['local_resp'].astype(str)
# Number of missed bytes
concat_df['missed_bytes'] = pd.to_numeric(concat_df['missed_bytes'], errors='coerce').astype('Int64')
# History of connection states
concat_df['history'] = concat_df['history'].astype(str)

# Packets sent from source
concat_df['orig_pkts'] = pd.to_numeric(concat_df['orig_pkts'], errors='coerce').astype('Int64')
# IP bytes sent from source
concat_df['orig_ip_bytes'] = pd.to_numeric(concat_df['orig_ip_bytes'], errors='coerce').astype('Int64')

# Packets from dest back to source
concat_df['resp_pkts'] = pd.to_numeric(concat_df['resp_pkts'], errors='coerce').astype('Int64')
# IP bytes from dest back to source
concat_df['resp_ip_bytes'] = pd.to_numeric(concat_df['resp_ip_bytes'], errors='coerce').astype('Int64')

# Tunnel label
concat_df['tunnel_parents'] = concat_df['tunnel_parents'].astype(str)
# Label (Malicious/Benign)
concat_df['label'] = concat_df['label'].astype(str)
# Detailed label
concat_df['detailed_label'] = concat_df['detailed_label'].astype(str)
```

```
<ipython-input-11-c3f2b5f86328>:4: FutureWarning: The behavior of 'to_datetime' with 'unit' when parsing strings is deprecated. In a fut
  concat_df['ts'] = pd.to_datetime(concat_df['ts'], unit = 's')
```

```
# Replacing '-' with NaN
concat_df = concat_df.replace('-', np.nan)
```

```
# Checking dtypes in the df
concat_df.info()
concat_df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2010439 entries, 0 to 835184
Data columns (total 23 columns):
 #   Column          Dtype
---  ------          -----
 0   ts              datetime64[ns]
 1   uid             object
 2   id.orig_h       object
 3   id.orig_p       Int64
 4   id.resp_h       object
 5   id.resp_p       Int64
 6   proto           object
 7   service         object
 8   duration        float64
 9   orig_bytes      Int64
 10  resp_bytes      Int64
 11  conn_state      object
 12  local_orig      object
 13  local_resp      object
 14  missed_bytes    Int64
 15  history         object
 16  orig_pkts       Int64
 17  orig_ip_bytes   Int64
 18  resp_pkts       Int64
 19  resp_ip_bytes   Int64
 20  tunnel_parents  object
 21  label           object
 22  detailed_label  object
dtypes: Int64(9), datetime64[ns](1), float64(1), object(12)
memory usage: 385.4+ MB
```

	ts	id.orig_p	id.resp_p	duration	orig_bytes	resp_bytes	missed_bytes	orig_pkts	orig_ip_bytes	resp_
count	2010439	2010438.0	2010438.0	301696.000000	301696.0	301696.0	2010438.0	2010438.0	2010438.0	20104
mean	2018-06-13 00:37:44.394941696	39657.693967	8831.400376	3.899354	18.432833	47.128308	0.0	1.369656	71.293466	0.12
min	2018-05-09 15:30:31.015073061	0.0	0.0	0.000001	0.0	0.0	0.0	0.0	0.0	
25%	2018-05-11 20:28:33.019131904	34095.0	23.0	2.997202	0.0	0.0	0.0	1.0	40.0	
50%	2018-05-14 06:59:19.040211968	43763.0	81.0	2.998789	0.0	0.0	0.0	1.0	40.0	
75%	2018-07-25 14:13:14.002879488	49554.0	8080.0	2.999040	0.0	0.0	0.0	1.0	60.0	
max	2018-08-01 13:15:06.734905005	65535.0	65535.0	93280.030966	6303.0	55565.0	0.0	3031.0	164117.0	5[illegible]
std	NaN	15458.446268	16936.432746	175.278726	93.995077	305.201275	0.0	2.663687	157.650473	4.61

The next step is to fill the NaN values to properly analyze the data, per best practices. In this step, I replace the numerical NaNs with the mean of their respective columns and replace the categorical NaNs with 'unknown'.

```
# Renaming for ease of use
df = concat_df

# Replacing the NaN values
# Separating between numeric and categorical features
numeric_features = df.select_dtypes(include=np.number).columns.tolist()
categorical_features = df.select_dtypes(include=['object']).columns.tolist()

# Converting NaNs in numeric cols
for feature in numeric_features:
  # Check if datatype is an int64
  if df[feature].dtype == 'Int64':
    # If so, fill NaN with the mean converted to an integer
    df[feature] = df[feature].fillna(int(df[feature].mean()))
  else:
    # Otherwise, fill NaN with the mean as usual
    df[feature] = df[feature].fillna(df[feature].mean())

# Converting NaNs in categorical cols
for feature in categorical_features:
  df[feature] = df[feature].fillna('unknown')
```

```
df.sample(10)
```

	ts	uid	id.orig_h	id.orig_p	id.resp_h	id.resp_p	proto	service	duration	orig_byt
96594	2018-05-20 17:27:33.996419907	CKZ4rx2180wk13WTUl	192.168.2.5	33786	200.168.87.203	59353	tcp	unknown	3.899354	
828244	2018-07-25 15:32:39.004375935	CnKDy31xU363FiHnIf	192.168.100.111	61237	147.32.107.173	23	tcp	unknown	3.899354	
48294	2018-05-20 06:00:06.983911037	CW1SrQ3cgeTUMlhWk9	192.168.2.5	42064	161.183.132.204	22	tcp	unknown	2.994967	
308444	2018-05-10 23:56:44.024627924	CJzc494f0BiZRXW0Oi	192.168.100.103	40316	186.112.229.60	23	tcp	unknown	3.899354	
27484	2018-07-25 13:32:18.009426117	CXCgeq4wkce0UYdq95	192.168.100.111	3487	116.31.148.31	23	tcp	unknown	3.899354	
177881	2018-05-10 10:06:52.011528969	CqjNVW20KZlxlIOFai	192.168.100.103	43763	117.143.66.53	10998	udp	unknown	3.899354	
557494	2018-07-25 14:41:51.001157045	CZPEATokyamqcgtze	192.168.100.111	53424	120.196.211.35	81	tcp	unknown	3.899354	
46636	2018-05-09 20:15:37.004281998	CFv5wJ36soVs9eTBs7	192.168.100.103	43763	91.19.83.160	6361	udp	unknown	3.899354	
733339	2018-05-12 22:27:20.024682045	CkpvOxGqwdnsE1PD1	192.168.100.103	52828	25.120.68.30	8080	tcp	unknown	3.899354	
626481	2018-07-25 14:55:16.002269030	C7YsS83ZzeGtyi1qpc	192.168.100.111	30952	190.223.87.209	81	tcp	unknown	3.899354	

10 rows × 23 columns

```
Warning: Total number of columns (23) exceeds max_columns (20) limiting to first (20) columns.
```

Exploratory Data Analysis

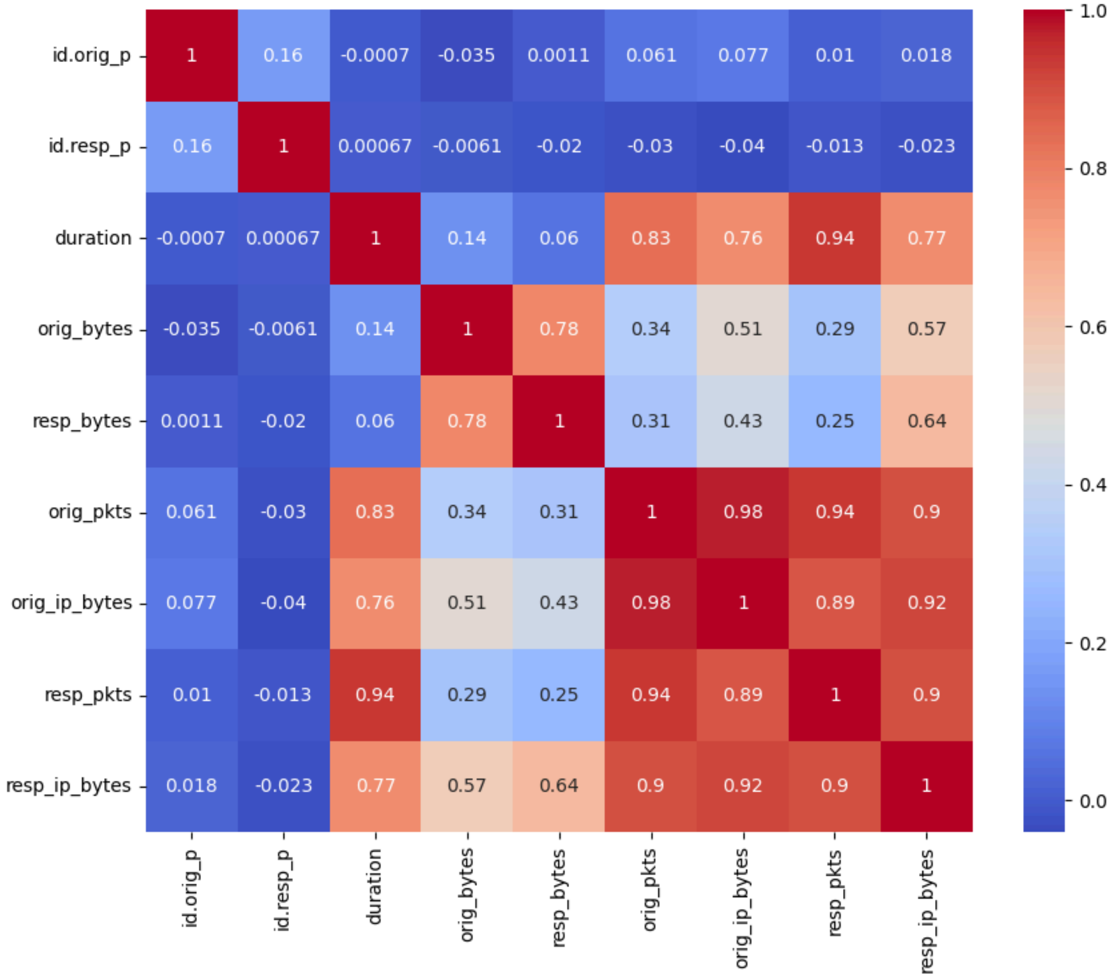
```
# Beginning EDA
df.shape
df.info()
df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2010439 entries, 0 to 835184
Data columns (total 23 columns):
 #   Column          Dtype
---  ------          -----
 0   ts              datetime64[ns]
 1   uid             object
 2   id.orig_h       object
 3   id.orig_p       Int64
 4   id.resp_h       object
 5   id.resp_p       Int64
 6   proto           object
 7   service         object
 8   duration        float64
 9   orig_bytes      Int64
 10  resp_bytes      Int64
 11  conn_state      object
 12  local_orig      object
 13  local_resp      object
 14  missed_bytes    Int64
 15  history         object
 16  orig_pkts       Int64
 17  orig_ip_bytes   Int64
 18  resp_pkts       Int64
 19  resp_ip_bytes   Int64
 20  tunnel_parents  object
 21  label           object
 22  detailed_label  object
dtypes: Int64(9), datetime64[ns](1), float64(1), object(12)
memory usage: 385.4+ MB
```

	ts	id.orig_p	id.resp_p	duration	orig_bytes	resp_bytes	missed_bytes	orig_pkts	orig_ip_bytes	resp_p
count	2010439	2010439.0	2010439.0	2.010439e+06	2010439.0	2010439.0	2010439.0	2010439.0	2010439.0	201043
mean	2018-06-13 00:37:44.394941696	39657.693966	8831.400376	3.899354e+00	18.064953	47.019255	0.0	1.369656	71.293466	0.124
min	2018-05-09 15:30:31.015073061	0.0	0.0	1.000000e-06	0.0	0.0	0.0	0.0	0.0	
25%	2018-05-11 20:28:33.019131904	34095.0	23.0	3.899354e+00	18.0	47.0	0.0	1.0	40.0	
50%	2018-05-14 06:59:19.040211968	43763.0	81.0	3.899354e+00	18.0	47.0	0.0	1.0	40.0	
75%	2018-07-25 14:13:14.002879488	49554.0	8080.0	3.899354e+00	18.0	47.0	0.0	1.0	60.0	
max	2018-08-01 13:15:06.734905005	65535.0	65535.0	9.328003e+04	6303.0	55565.0	0.0	3031.0	164117.0	59
std	NaN	15458.442424	16936.428534	6.789971e+01	36.412268	118.229292	0.0	2.663687	157.650434	4.616

```
# Removing the empty missed_bytes col as it is essentially empty
df.drop('missed_bytes', axis=1, inplace=True)
```

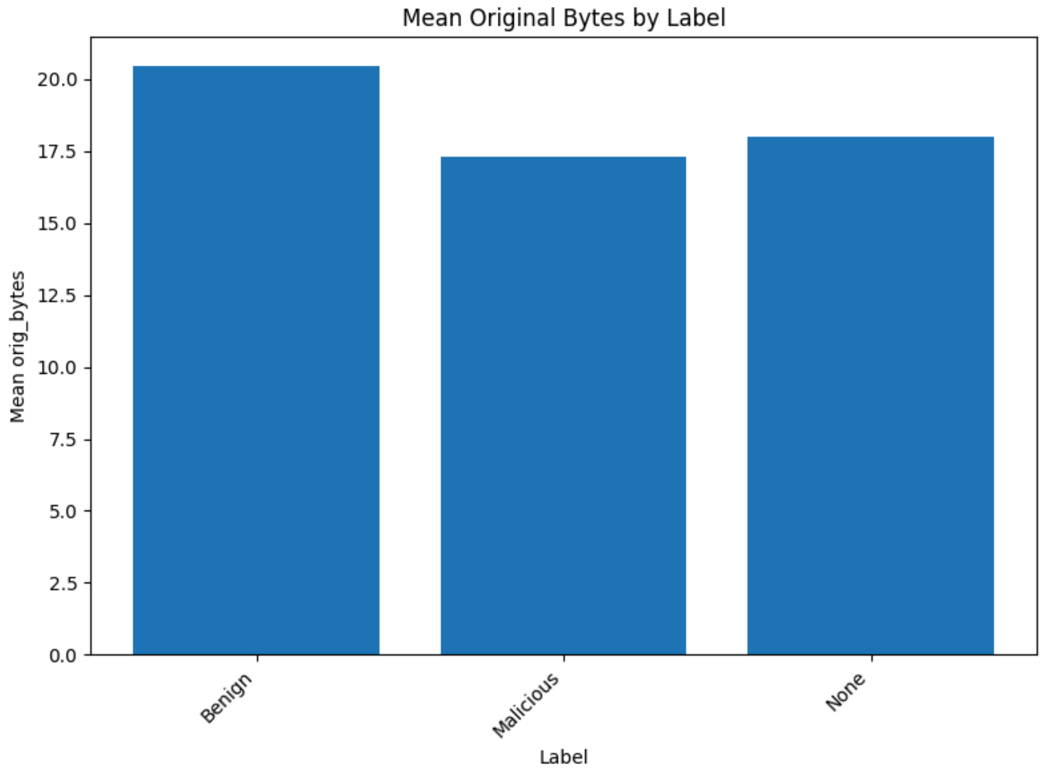
```
# creating a heatmap to begin looking at correlation
plt.figure(figsize=(10, 8))
numeric_df = df.select_dtypes(include=np.number)
sns.heatmap(numeric_df.corr(), annot=True, cmap='coolwarm')
plt.show()
```



Visualizing the Data

1. Visualize the number of connections based on their label (either malicious or benign). Is there a possible correlation between the number of original bytes sent and whether the connection was malicious or benign?

```
# Grouping data by label and finding mean number of origin bytes per group
grouped_data = df.groupby('label')['orig_bytes'].mean().reset_index()

# Visualizing the results
plt.figure(figsize=(8, 6))  # Adjust figure size if needed
plt.bar(grouped_data['label'], grouped_data['orig_bytes'])
plt.xlabel("Label")
plt.ylabel("Mean orig_bytes")
plt.title("Mean Original Bytes by Label")
plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels if needed
plt.tight_layout()  # Adjust layout for better spacing
plt.show()
```



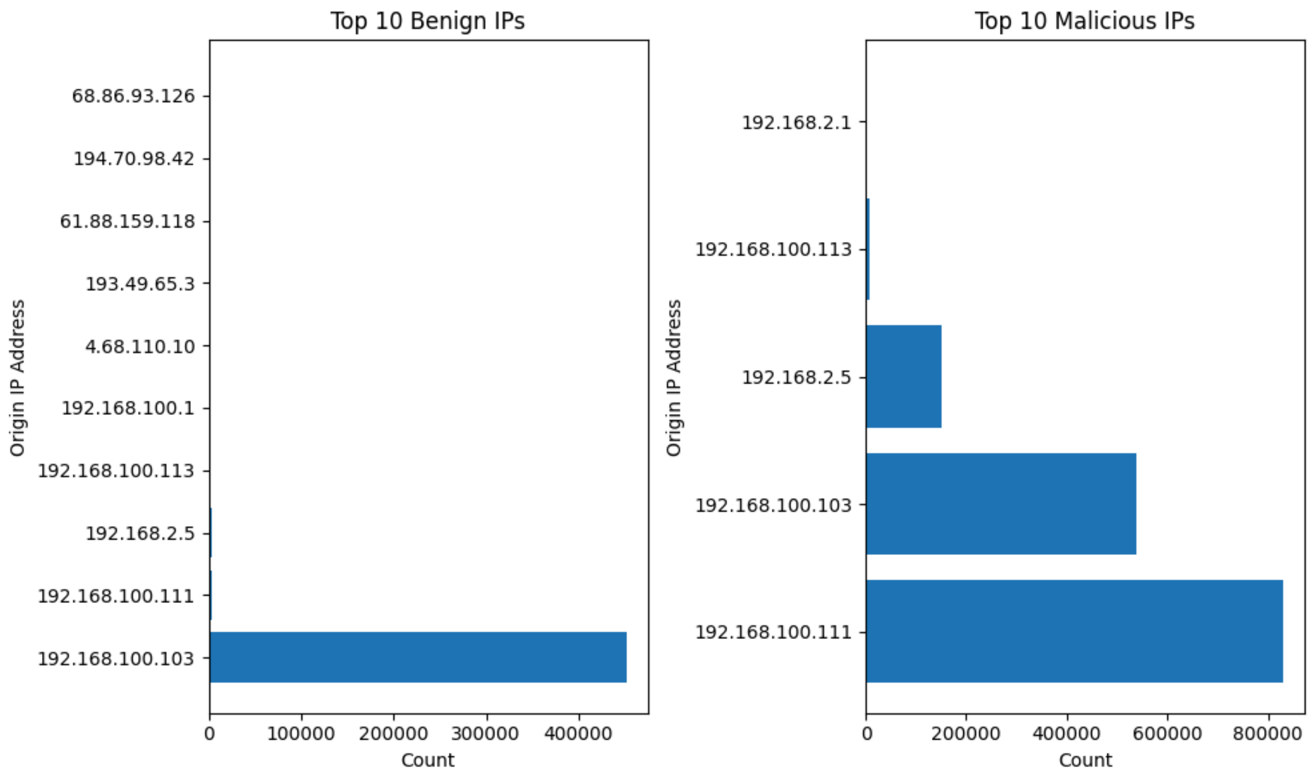
The data illustrates that there is not much of a difference between the average number of original bytes sent from the source of a benign connection versus that of a malicious connection.

The mean number of original bytes sent from the source of a benign connection was just slightly higher than that of a malicious connection.

2. Visualize the common origin IP addresses used in malicious and benign connections. Are some origin IP addresses more likely to be attempting a malicious connection than others?

```
# Grouping the data according to ip address and label
ip_counts = df.groupby(['id.orig_h', 'label'])['label'].count().reset_index(name='count')

# Filtering and storing the most common benign and malicious origin IP addresses
top_benign_ips = ip_counts[ip_counts['label'] == 'Benign'].sort_values('count', ascending=False).head(10)  # Top 10 benign IPs
top_malicious_ips = ip_counts[ip_counts['label'] == 'Malicious'].sort_values('count', ascending=False).head(10)  # Top 10 malicious IPs

# Printing results
print("Top 10 Benign IPs:")
print(top_benign_ips)

print("\nTop 10 Malicious IPs:")
print(top_malicious_ips)

# Visualizing results
plt.figure(figsize=(10, 6))

# Creating a subplot for benign IPs
plt.subplot(1, 2, 1)
plt.barh(top_benign_ips['id.orig_h'], top_benign_ips['count'])
plt.xlabel("Count")
plt.ylabel("Origin IP Address")
plt.title("Top 10 Benign IPs")

# Creating a subplot for malicious IPs
plt.subplot(1, 2, 2)
plt.barh(top_malicious_ips['id.orig_h'], top_malicious_ips['count'])
plt.xlabel("Count")
plt.ylabel("Origin IP Address")
plt.title("Top 10 Malicious IPs")

# Adjusting layout for better spacing
plt.tight_layout()
plt.show()
```

```
Top 10 Benign IPs:
            id.orig_h   label   count
6908   192.168.100.103  Benign  451588
6910   192.168.100.111  Benign    3603
6916       192.168.2.5  Benign    3320
6912   192.168.100.113  Benign    2179
6906     192.168.100.1  Benign    1651
11011      4.68.110.10  Benign      61
7059       193.49.65.3  Benign      26
12263    61.88.159.118  Benign      23
7139      194.70.98.42  Benign      23
12961     68.86.93.126  Benign      20

Top 10 Malicious IPs:
            id.orig_h      label   count
6911  192.168.100.111  Malicious  831256
6909  192.168.100.103  Malicious  539473
6917      192.168.2.5  Malicious  151566
6913  192.168.100.113  Malicious    8222
6915      192.168.2.1  Malicious       1
```

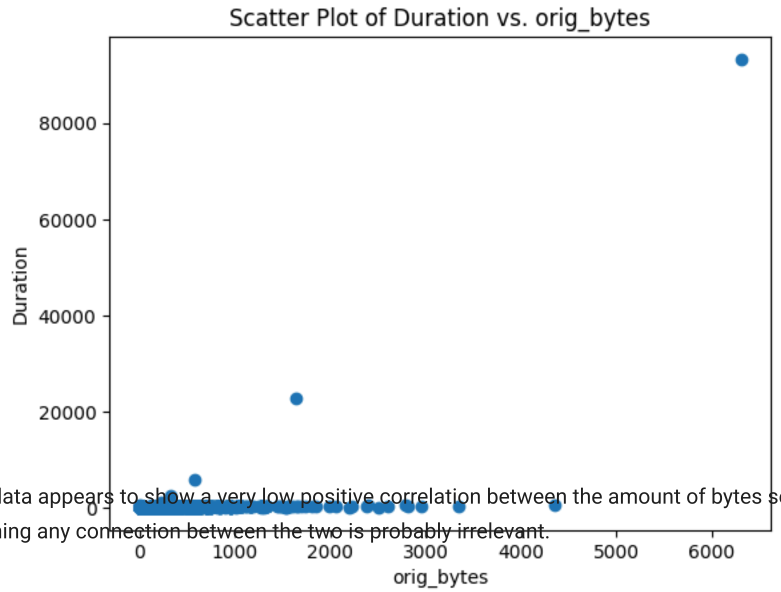


The data demonstrates that there are several IP addresses with many logged instances of malicious connections and several others which are commonly benign.

The origin IP address 192.168.100.111, in particular, had well over 800,000 logged instances of malicious connections.

3. Visualize the connection between the duration of the connection and the amount of bytes sent from the source to the destination. Does the amount of original bytes appear to influence the length of the connection duration?

```
# Splitting data into x and y
x = df['orig_bytes']
y = df['duration']

plt.scatter(x, y)
plt.xlabel('orig_bytes')  # Replace with the actual name of your x-axis feature
plt.ylabel('Duration')
plt.title('Scatter Plot of Duration vs. orig_bytes')  # Replace with a descriptive title
plt.show()
```



The data appears to show a very low positive correlation between the amount of bytes sent originally and the duration of the connection, meaning any connection between the two is probably irrelevant.

4. Visualize the distribution of the amount of original packets sent from the source to the destination. Does there appear to be a