What is Colaboratory?

Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with

- Zero configuration required
- Free access to GPUs
- Easy sharing

Whether you're a **student**, a **data scientist** or an **AI researcher**, Colab can make your work easier. Watch Introduction to Colab to learn more, or just get started below!

Introduction

Exploratory Data Analysis (EDA) Walkthrough

The goal of this walkthrough is to help show you the process done by analysts and researchers. EDA is one of the most important first steps when dealing with new data.

EDA allows us to better understand the data and the tools we can use in future steps.

Business Context

You are a new cybersecurity analyst, and have been tasked with finding out some more information on recent events that have happened over our network.

1) Most targeted Destination IP Address
2) Most Ports attacked
3) Most Frequently/common type of Attack
4) Different time of the day (odd , hours, day or night)
5) Find the patterns

Importing Libraries

The following lines of code are installing certain libraries or dependencies for the notebook to run properly.

```
import pandas as pd
```

```
import seaborn as sns
import glob
import matplotlib.pyplot as plt
import ipaddress
import numpy as np
from scipy import stats
from scipy.stats import chi2_contingency
from datetime import datetime, timedelta
import math
import missingno as msno
plt.style.use('ggplot')
import warnings
warnings.filterwarnings('ignore')
```

Load Data

```
df1 = pd.read_csv('https://raw.githubusercontent.com/BradleyConlin/DTI6160_EDA_Tutorial/main/D

df1.shape
```

```
(50000, 11)
```

```
df2 = pd.read_csv('https://raw.githubusercontent.com/BradleyConlin/DTI6160_EDA_Tutorial/main/D

df2.shape
```

```
(46615, 11)
```

```
df3 = pd.read_csv('https://raw.githubusercontent.com/BradleyConlin/DTI6160_EDA_Tutorial/main/D

df3.shape
```

```
(30000, 11)
```

```
df4 = pd.read_csv('https://raw.githubusercontent.com/BradleyConlin/DTI6160_EDA_Tutorial/main/D

df4.shape
```

```
(20000, 11)
```

```
combined = [df1, df2, df3, df4]
df = pd.concat(combined)
df.shape
```

```
(146615, 11)
```

Step 1: Data Exploration

Initial Discovery

We will now begin to gain insight on the data with the following commands:

- **df.shape** (row, column)
- **df.columns** (column titles)
- **df.head** (top 5 results in table format)
- **df.nunique** (count of unique values for each variable/dimension)

```
df.shape
```

```
(146615, 11)
```

This shows us that there are 11 columns of data, and there are 96,613 rows of data.

```
df.columns
```

```
Index(['Attack category', 'Attack subcategory', 'Protocol', 'Source IP',
       'Source Port', 'Destination IP', 'Destination Port', 'Attack Name',
       'Attack Reference', '.', 'Time'],
      dtype='object')
```

```
df.head()
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	Dest
0	Reconnaissance	HTTP	tcp	175.45.176.0	13284	149.171.126.16	
1	Exploits	Unix 'r' Service	udp	175.45.176.3	21223	149.171.126.18	
2	Exploits	Browser	tcp	175.45.176.2	23357	149.171.126.16	
3	Exploits	Miscellaneous Batch	tcp	175.45.176.2	13792	149.171.126.16	
4	Exploits	Cisco IOS	tcp	175.45.176.2	26939	149.171.126.10	

df.head() returns the top 5 results of the data in table format. By changing the value in "()", you can change

```
df.head(2)
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	Destination Port	A
0	Reconnaissance	HTTP	tcp	175.45.176.0	13284	149.171.126.16	80	Domir Da /dolad Solaris

```
df.nunique(axis=0)
```

```
Attack category          18
Attack subcategory      163
Protocol                131
Source IP                11
Source Port           41318
Destination IP           10
Destination Port      17330
Attack Name            7149
Attack Reference       3069
.                         1
Time                  46410
dtype: int64
```

df.nunique will show help us understand the type and spread of the data. Most notable would be the "." variable, with only has a single unique value.

Step 2: Data Cleaning

Initial Cleaning

Data cleaning is used to not only increase the data integrity, but to also make it more usable and understandable to humans.

We will begin with the time variable. It is currently in a 10-digit serial format, we need to convert it to normal data time.

Example of Feature Engineering

```
df[['Start time','Last time']] = df['Time'].str.split('-',expand=True)
df.head(2)
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	Dest
0	Reconnaissance	HTTP	tcp	175.45.176.0	13284	149.171.126.16	
1	Exploits	Unix 'r' Service	udp	175.45.176.3	21223	149.171.126.18	

```
df['Start time']
```

```
0        1421927414
1        1421927415
2        1421927416
3        1421927417
4        1421927418
            ...
19995    1424256873
19996    1424256873
19997    1424256874
19998    1424256874
19999    1424256875
Name: Start time, Length: 146615, dtype: object
```

Now that we know the format, and the beginning and end times in the data, we will transform the data, and add to it a duration column in seconds.

```
df['Start time'] = pd.to_datetime(df['Start time'], unit='s')
df['Last time'] = pd.to_datetime(df['Last time'], unit='s')
df['Duration'] = ((df['Last time'] - df['Start time']).dt.seconds).astype(int)
```

```
df.head()
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	Destination Port	A
0	Reconnaissance	HTTP	tcp	175.45.176.0	13284	149.171.126.16	80	Domi Dat
1	Exploits	Unix 'r' Service	udp	175.45.176.3	21223	149.171.126.18	32780	Vu

Our df looks much better, but we do not need the original time format anymore. So we will use df.drop to remove this df, and then check to make sure we did it correctly.

```
df = df.drop(['Time'],axis=1)
```

```
df.head(1)
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	Dest
0	Reconnaissance	HTTP	tcp	175.45.176.0	13284	149.171.126.16	

```
df['Start time'].astype(str).str.split(' ').str[0].unique()
```

```
array(['2015-01-22', '2015-02-18'], dtype=object)
```

As you can see from the above table, the start and Last time are now given in YYYY-MM-DD Time of Day.

For ease and time, we will be looking at two days in partcular in the following cells:

January 22, 2015 February 18, 2015

Next we will check to see if the "." holds any data worth saving.

```
df.dtypes
```

```
Attack category              object
Attack subcategory           object
Protocol                     object
Source IP                    object
Source Port                   int64
Destination IP               object
Destination Port              int64
Attack Name                  object
Attack Reference             object
.                            object
Start time           datetime64[ns]
```

```
Last time              datetime64[ns]
Duration                        int64
dtype: object
```

```
df['.'].unique()
```

```
array(['.'], dtype=object)
```

Given that the "." column is empty, we will next delete this column and the old *Time* column to clean up our data

```
df.shape
```

```
(146615, 13)
```

```
df = df.drop(['.'] ,axis=1)
df.head()
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	Destination Port	A
0	Reconnaissance	HTTP	tcp	175.45.176.0	13284	149.171.126.16	80	Domi Dat
1	Exploits	Unix 'r' Service	udp	175.45.176.3	21223	149.171.126.18	32780	Vu
2	Exploits	Browser	tcp	175.45.176.2	23357	149.171.126.16	80	Wir SetAb
3	Exploits	Miscellaneous Batch	tcp	175.45.176.2	13792	149.171.126.16	5555	HP (https:
4	Exploits	Cisco IOS	tcp	175.45.176.2	26939	149.171.126.10	80	C Byp

```
df.shape
```

```
(146615, 12)
```

As we can see, from the **df.shape** commands (before/after) that the dataframe has been transformed correctly.

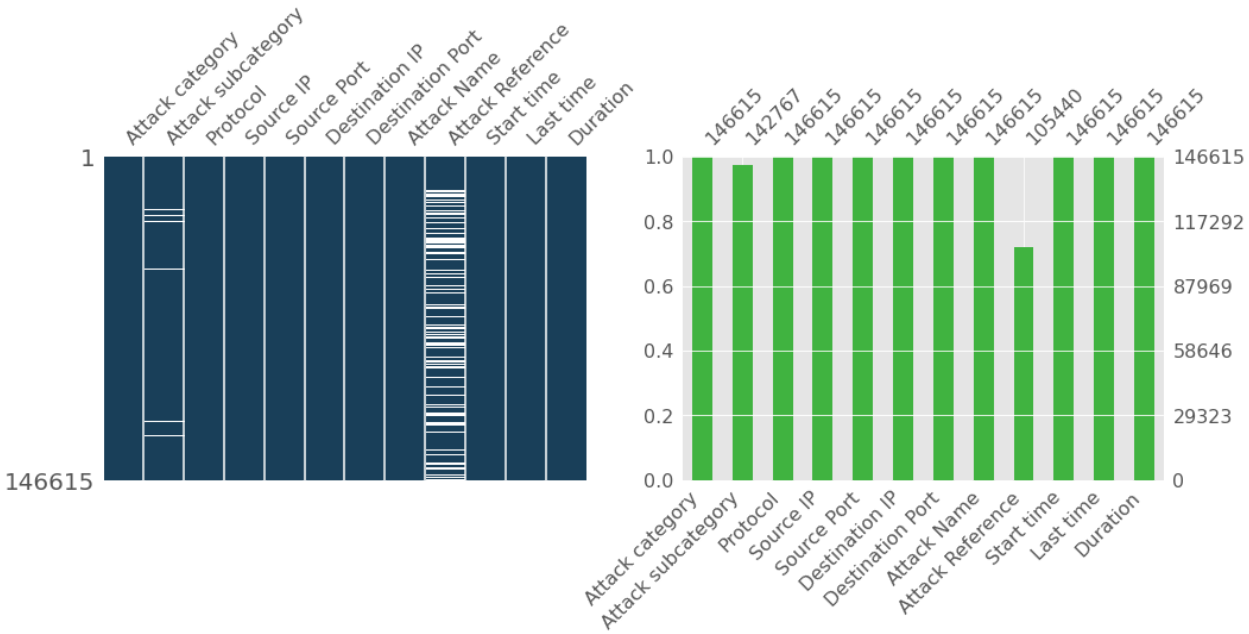
Graphical Analysis

Sometimes we will use graphical analysis to get a better understanding of the data. In the following subsection we will use the following two graphs to display ***missing values***:

Left: Matrix shows the distribution of missing data

Right: Bar chart showing the total number of values *(anything less than 178031 means missing data)*

```
figure, (ax1, ax2) = plt.subplots(1, 2, figsize=(16,5))
msno.matrix(df, ax=ax1, sparkline=False, color=(0.1, 0.25, 0.35))
msno.bar(df, ax=ax2, color=(0.25, 0.7, 0.25))
plt.show()
```



Left Graph

The missing data from "Attack Category" is rather sparse, and generally not a concern.

"Attack Reference" appears to have a lot of missing data that is somewhat evenly spread out.

Right Graph

This graph will show us the overall sum of the variables and inversely, the missing values.

This shows us that we have ~2000 and ~28,0000 missing values in "*Attack Category*" and "*Attack Reference*" respectfully.

Secondary Cleaning

To see more clearly how many data points are missing, the following code will help us see these values in a more aggregated view

```
df.isnull().sum()
```

```
Attack category           0
Attack subcategory     3848
Protocol                  0
Source IP                 0
Source Port               0
Destination IP            0
Destination Port          0
Attack Name               0
Attack Reference      41175
Start time                0
Last time                 0
Duration                  0
dtype: int64
```

```
df['Attack subcategory']
```

```
0                        HTTP
1           Unix 'r' Service
2                    Browser
3         Miscellaneous Batch
4                   Cisco IOS
                   ...
19995                    RIP
19996         Web Application
19997                    RIP
19998                    RIP
19999                    RIP
Name: Attack subcategory, Length: 146615, dtype: object
```

```
df['Attack Reference']
```

```
0                                                       -
1         CVE 2002-0573 (http://cve.mitre.org/cgi-bin/cv...
2         CVE 2005-4560 (http://cve.mitre.org/cgi-bin/cv...
3         CVE 2011-1729 (http://cve.mitre.org/cgi-bin/cv...
4         CVE 2001-0537 (http://cve.mitre.org/cgi-bin/cv...
                              ...
19995                                                   NaN
```

```
19996    CVE 2014-1649 (http://cve.mitre.org/cgi-bin/cv...
19997                                                  NaN
19998                                                  NaN
19999                                                  NaN
Name: Attack Reference, Length: 146615, dtype: object
```

We can see from exploring these two variables that while "Attack subcategory" has valuable data, due to the small amount of data and lack of tacit knowledge, it would be best to remove these entries for a more clean data set.

On the otherhand, we can see that "Attack Reference" appears to add little to no value for our current purposes. So it would most likely be best to remove this dimension (variable/column) completely.

```
# Moving unknown Attack subcategory to "Not Registered" or you could use "other"

df["Attack subcategory"] = df["Attack subcategory"].fillna("Not Registered")
```

```
df.isnull().sum()
```

```
Attack category           0
Attack subcategory        0
Protocol                  0
Source IP                 0
Source Port               0
Destination IP            0
Destination Port          0
Attack Name               0
Attack Reference      41175
Start time                0
Last time                 0
Duration                  0
dtype: int64
```

```
df[pd.isnull(df).any(axis=1)].shape
```

```
(41175, 12)
```

After checking to see if we have properly removed the *Attack subcategory* and checking the shape again, we will see if we have any duplicate line items.

```
df[df.duplicated()].shape
```

```
(4154, 12)
```

```
print('Dimensions before dropping duplicated rows: ' + str(df.shape))
```

```
df = df.drop(df[df.duplicated()].index)
print('Dimensions after dropping duplicated rows: ' + str(df.shape))
```

```
Dimensions before dropping duplicated rows: (146615, 12)
Dimensions after dropping duplicated rows: (134452, 12)
```

```
# This chart should be empty, it is showing us the duplicates that remain

df[df.duplicated()]
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	Destination Port	Attack Name	At Refer

---

Now that we have reviewed the obvious dimensions with issues, we now need to look at our other dimensions for validity.

The only valid port ranges are between **0** and **65535**.

As we saw from our original investigation, there are values beyond these limits, which means we need to clean up the data

---

Garbage In, Garbage Out

The following code will take any entries with invalid port values, and store them into their own dataframe (df)

```
invalid_SP = (df['Source Port'] < 0) | (df['Source Port'] > 65535)
invalid_DP = (df['Destination Port'] < 0) | (df['Destination Port'] > 65535)
df[invalid_SP | invalid_DP]
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	Destination Port	
24230	Generic	IXIA	udp	187.48.276.1	1043	149.171.126.14	-64	M
24576	Generic	IXIA	udp	175.45.171.3	1043	149.171.126.18	-230	M
24795	Generic	IXIA	udp	175.45.176.0	1043	149.171.126.12	-458	M
24859	Generic	IXIA	udp	175.45.171.3	1043	149.171.126.18	-336	M
26124	Fuzzers	RIP	udp	175.45.176.2	1508	149.171.126.13	-997	
...	...	...	...	...	...	...	...	
46598	Generic	IXIA	udp	175.45.176.0	79909	149.171.126.10	53	M

We will now delete the invalid entries and check the shape...again.

```
df = df[~(invalid_SP | invalid_DP)].reset_index(drop=True)
```

```
df.shape
```

```
(133567, 12)
```

(UDP) UDP

We will now check some of the data inside the df to make sure it is clean and accurate.

```
print('Total number of different protocols:', len(df['Protocol'].unique()))
print('Total number of different Attack categories:', len(df['Attack category'].unique()))
df['Protocol'].unique()
```

```
Total number of different protocols: 131
Total number of different Attack categories: 18
array(['tcp', 'udp', 'Tcp', 'UDP', 'ospf', 'sctp', 'sep', 'mobile',
       'sun-nd', 'swipe', 'pim', 'ggp', 'ip', 'ipnip', 'st2', 'cbt',
       'egp', 'argus', 'bbn-rcc', 'chaos', 'emcon', 'igp', 'nvp', 'pup',
       'xnet', 'mux', 'dcn', 'hmp', 'prm', 'trunk-1', 'xns-idp',
       'trunk-2', 'leaf-1', 'leaf-2', 'irtp', 'rdp', 'iso-tp4', 'netblt',
       'merit-inp', 'mfe-nsp', '3pc', 'idpr', 'xtp', 'ddp', 'idpr-cmtp',
       'tp++', 'il', 'ipv6', 'idrp', 'ipv6-frag', 'ipv6-route', 'sdrp',
       'gre', 'mhrp', 'rsvp', 'bna', 'i-nlsp', 'rvd', 'narp', 'ipv6-no',
       'skip', 'tlsp', 'ipv6-opts', 'any', 'cftp', 'kryptolan',
       'sat-expak', 'ippc', 'sat-mon', 'cpnx', 'ipcv', 'visa', 'cphb',
```

```
       'wsn', 'br-sat-mon', 'pvp', 'wb-expak', 'wb-mon', 'iso-ip',
       'secure-vmtp', 'vmtp', 'ttp', 'vines', 'nsfnet-igp', 'dgp',
       'eigrp', 'tcf', 'sprite-rpc', 'larp', 'mtp', 'ax.25', 'ipip',
       'micp', 'aes-sp3-d', 'encap', 'etherip', 'gmtp', 'pri-enc', 'ifmp',
       'pnni', 'aris', 'a/n', 'qnx', 'scps', 'compaq-peer', 'ipcomp',
       'snp', 'ipx-n-ip', 'vrrp', 'pgm', 'zero', 'ddx', 'iatp', 'l2tp',
       'srp', 'stp', 'smp', 'uti', 'sm', 'ptp', 'crtp', 'fire', 'isis',
       'crudp', 'sccopmce', 'iplt', 'pipe', 'sps', 'fc', 'unas', 'ib'],
      dtype=object)
```

As we can see, there are multiple duplications in the list.

TCP, UDP, etc.

The following code will show us the same for the attack categories.

```
df['Attack category'].unique()
```

```
array(['Reconnaissance', 'Exploits', 'DoS', 'Generic', 'Shellcode',
       ' Fuzzers', 'Worms', 'Backdoors', 'Analysis', ' Fuzzers ',
       ' Reconnaissance ', 'Backdoor', ' Shellcode ', ' Analysis',
       'Fuzzers', 'Analysis ', 'Reconnaissance ', 'Shellcode '],
      dtype=object)
```

As we can see, there are duplicates due to poor spelling. We will remove white space, uppercase the letters and align "*BACKDOOR*" and "*BACKDOORS*".

```
df['Protocol'] = df['Protocol'].str.upper().str.strip()
df['Attack category'] = df['Attack category'].str.upper().str.strip()
df['Attack category'] = df['Attack category'].str.strip().replace('BACKDOORS','BACKDOOR')
df['Attack category'] = df['Attack category'].str.strip().replace(' Analytics','Analytics')
```

Check to make sure the operation executed properly.

```
print('Total number of different protocols:', len(df['Protocol'].unique()))
print('Total number of different Attack categories:', len(df['Attack category'].unique()))
```

```
Total number of different protocols: 129
Total number of different Attack categories: 9
```

Now to look at *Attack Reference* for missing values.

```
df[pd.isnull(df['Attack Reference'])]
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	Destinatio Por
240	FUZZERS	OSPF	OSPF	175.45.176.3	0	149.171.126.14	
463	FUZZERS	BGP	TCP	175.45.176.2	63685	149.171.126.13	17
464	FUZZERS	BGP	TCP	175.45.176.2	48413	149.171.126.13	17
465	FUZZERS	BGP	TCP	175.45.176.2	30451	149.171.126.13	17
466	FUZZERS	BGP	TCP	175.45.176.2	58077	149.171.126.13	17
...	...	...	...	...	...	...	.
133561	RECONNAISSANCE	SunRPC Portmapper (TCP) TCP Service	TCP	175.45.176.2	2380	149.171.126.12	11
133563	FUZZERS	RIP	UDP	175.45.176.3	2722	149.171.126.11	52
133564	FUZZERS	RIP	UDP	175.45.176.0	64228	149.171.126.13	52
133565	FUZZERS	RIP	UDP	175.45.176.3	13825	149.171.126.11	52

```
df[pd.isnull(df['Attack Reference'])].shape
```

```
(38585, 12)
```

38585 rows × 12 columns

Now to segment the missing *Attack Reference* by *Attack Category*

```
#Missing Values for each category

print(df[pd.isnull(df['Attack Reference'])]['Attack category'].value_counts())
```

```
FUZZERS           22806
RECONNAISSANCE    13579
ANALYSIS           1248
SHELLCODE           610
GENERIC             249
```

```
BACKDOOR             47
DOS                  35
WORMS                 8
EXPLOITS              3
Name: Attack category, dtype: int64
```

```
#Total Values for each category

print(df['Attack category'].value_counts())
```

```
EXPLOITS          49767
FUZZERS           27565
DOS               18735
RECONNAISSANCE    15856
GENERIC           14781
BACKDOOR           3875
ANALYSIS           1556
SHELLCODE          1303
WORMS               129
Name: Attack category, dtype: int64
```

```
#Calculates the percentage of values missing
((df[pd.isnull(df['Attack Reference'])]['Attack category'].value_counts()/df['Attack category'
```

```
RECONNAISSANCE    85.639506
FUZZERS           82.735353
ANALYSIS          80.205656
SHELLCODE         46.815042
WORMS              6.201550
GENERIC            1.684595
BACKDOOR           1.212903
DOS                0.186816
EXPLOITS           0.006028
Name: Attack category, dtype: float64
```

Feature Engineering

We will now add in the TCP-ports.csv values into our working set.

As you will see below, this will give us a better idea of what we are looking at.

```
tcp_ports = pd.read_csv('https://raw.githubusercontent.com/BradleyConlin/DTI6160_EDA_Tutorial/
tcp_ports['Service'] = tcp_ports['Service'].str.upper()
tcp_ports.head()
```

	Port	Service	Description
0	0	NaN	Reserved

```
print('Dimensions before merging dataframes: ' ,(df.shape))

newdf = pd.merge(df, tcp_ports[['Port','Service']], left_on='Destination Port', right_on='Port
newdf = newdf.rename(columns={'Service':'Destination Port Service'})

print('Dimensions after merging dataframes: ' + str(newdf.shape))
```

```
Dimensions before merging dataframes:  (133567, 12)
Dimensions after merging dataframes: (133567, 14)
```

Looking at the shape of the newdf or new data frame, we can see that we have the two newly added columns (namely the Destination Port Service.

We will now remove "Port" from the data as this will just be a duplication of data.

```
newdf = newdf.drop(columns=['Port'])
newdf.head()
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	D
0	RECONNAISSANCE	HTTP	TCP	175.45.176.0	13284	149.171.126.16	
1	EXPLOITS	Unix 'r' Service	UDP	175.45.176.3	21223	149.171.126.18	
2	EXPLOITS	Browser	TCP	175.45.176.2	23357	149.171.126.16	
3	EXPLOITS	Miscellaneous Batch	TCP	175.45.176.2	13792	149.171.126.16	
4	EXPLOITS	Cisco IOS	TCP	175.45.176.2	26939	149.171.126.10	

Data Cleaning is Complete

Now to move forward into looking into the data more deeply. We will begin by looking at the shape and distribution of the data.

Data Shape and Distribution

```
newdf['Attack category'].unique()
```

```
array(['RECONNAISSANCE', 'EXPLOITS', 'DOS', 'GENERIC', 'SHELLCODE',
       'FUZZERS', 'WORMS', 'BACKDOOR', 'ANALYSIS'], dtype=object)
```

```
# Checking to see the total counts in the Attack Category

newdf['Attack category'].value_counts()
```

```
EXPLOITS          49767
FUZZERS           27565
DOS               18735
RECONNAISSANCE    15856
GENERIC           14781
BACKDOOR           3875
ANALYSIS           1556
SHELLCODE          1303
WORMS               129
Name: Attack category, dtype: int64
```

```
# Checking the pertentage of overall attacks by Attack Category

newdf['Attack category'].value_counts()*100/newdf['Attack category'].value_counts().sum()
```

```
EXPLOITS          37.259952
FUZZERS           20.637583
DOS               14.026668
RECONNAISSANCE    11.871196
GENERIC           11.066356
BACKDOOR           2.901166
ANALYSIS           1.164958
SHELLCODE          0.975540
WORMS              0.096581
Name: Attack category, dtype: float64
```

What does the data tell us?

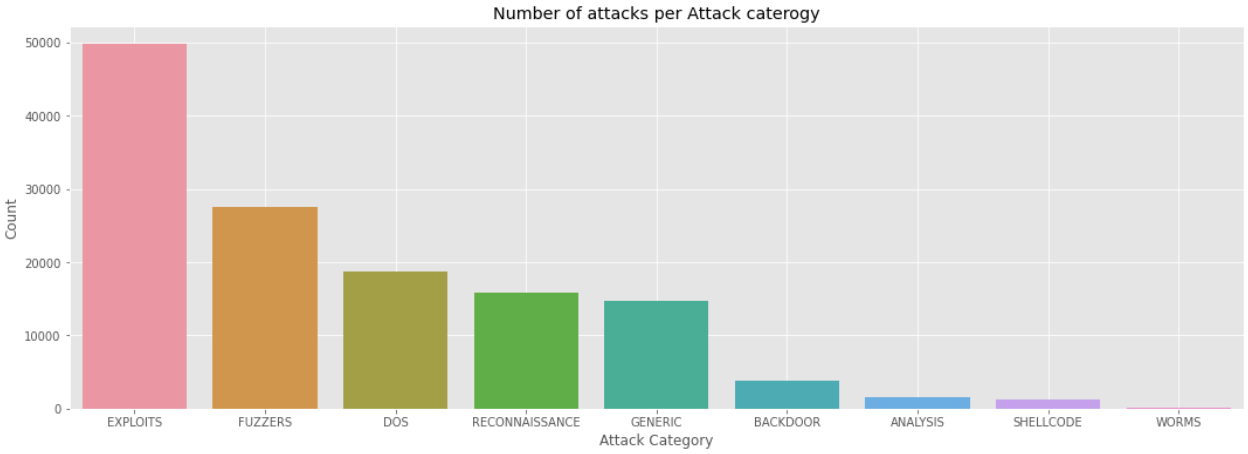
As we can see from the above dataframes, we have unbalanced data.

- The top 5 categories hold ~95% of the data.

Graphical Analysis

The following three cells are different ways for us to visualize the data.

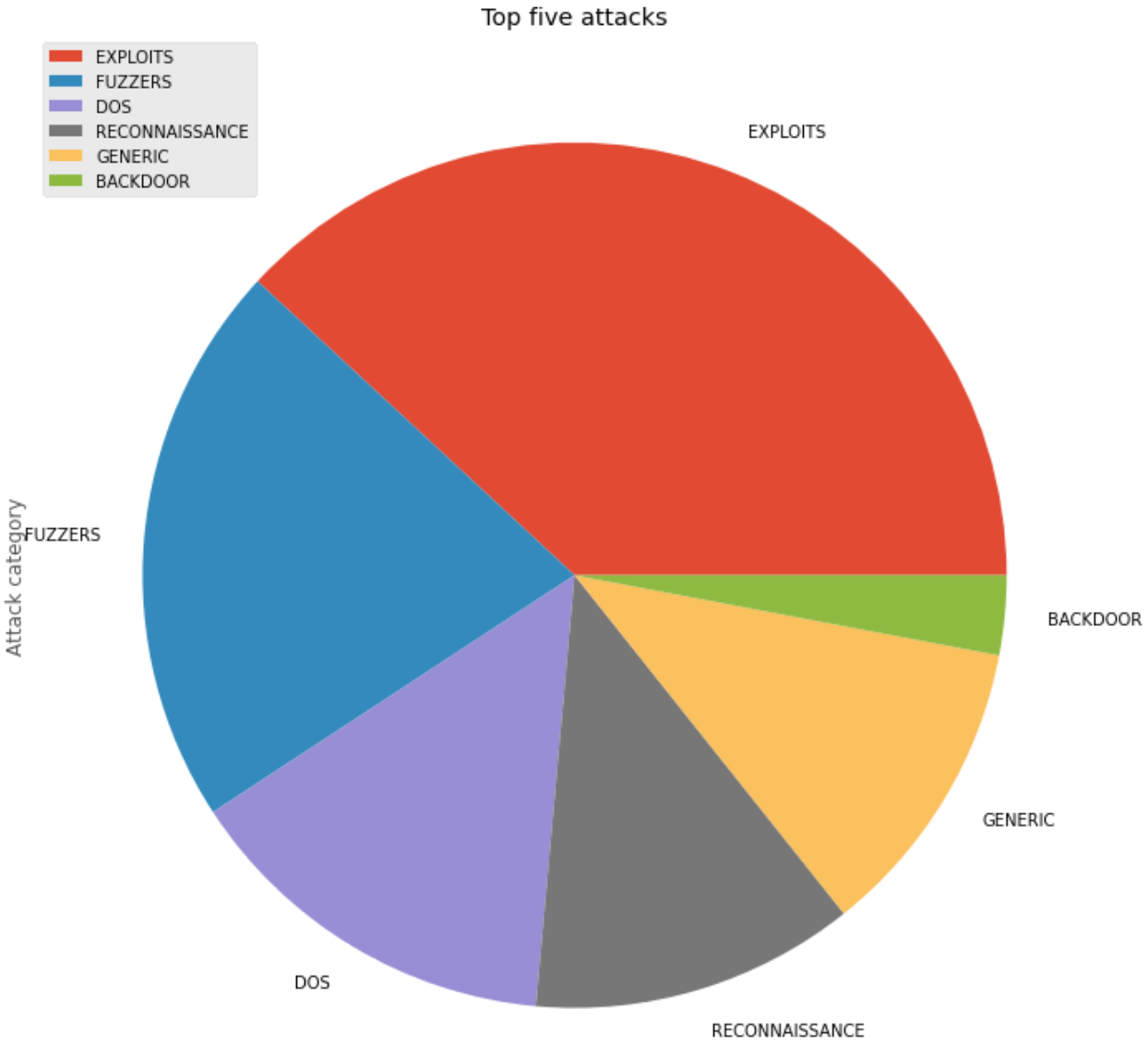
```
plt.figure(figsize=(18,6))
sns.barplot(x=newdf['Attack category'].value_counts().index,y=newdf['Attack category'].value_c
plt.xlabel('Attack Category')
plt.ylabel('Count')
plt.title('Number of attacks per Attack caterogy')
plt.grid(True)
```




```
pd.DataFrame(newdf['Attack category'].value_counts())[:]
```

	Attack category
EXPLOITS	49767
FUZZERS	27565
DOS	18735
RECONNAISSANCE	15856
GENERIC	14781
BACKDOOR	3875
ANALYSIS	1556
SHELLCODE	1303
WORMS	129

```
a=pd.DataFrame(newdf['Attack category'].value_counts())[:6]


a.plot(kind='pie', subplots=True, figsize=(12, 12))
plt.title('Top five attacks')
plt.legend(loc='left')
plt.show()
```




```
newdf.describe()
```

	Source Port	Destination Port	Duration
count	133567.000000	133567.000000	133567.000000
mean	17951.522869	3414.397853	2.362133
std	21042.782275	9018.154391	9.335327
min	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000

Interesting Findings

- Min & Max are identical
- Mean and 75% do not match

Hypothesis Testing

$$H_0 : \mu_1 = \mu_2$$
$$H_a : \mu_1 \neq \mu_2$$

We can obtain one of two results from the test:

1. If the p**-value** is less than our significance level ($p < \alpha$) we reject the null hypothesis H_0 and affirm that the observed difference is **statistically significant**.
2. If the p**-value** is greater than our significance level ($p > \alpha$) we will have to retain H_0 and conclude that the observed difference **is not statistically significant**.

The hypothesis test is conducted using a statistical $T - test$ which specifies the two Series `df_interest['Source Port']` and `df_interest['Destination Port']`.

By specifying these two Series, we are automatically referring to a comparative test of the means of both Series:

```
statistic, pvalue = stats.ttest_ind( newdf['Source Port'], newdf['Destination Port'], equal_va
print('p-value in T-test: ' + str(pvalue))
```

```
    p-value in T-test: 0.0
```

Because the p-value is very close to zero, Python approximates this measurement to 0.0.

Therefore, we can reject the null hypothesis H_0 regarding the equality of the means of the source and destination ports.

Meaning that the difference between the two *means* are significantly different.

Correlation Coefficiant

We will be using two methods for correlation calculation:

- ***Pearson's correlation***: evaluates the linear relationships between two variables. If the value is close to 0, there is a weak or nonexistent linear relationship between the variables.
- ***Spearman's correlation***: evaluates the monotonic relationships between two variables. If the value is close to 0, there is a weak or nonexistent monotonic relationship between the variables.

Definitions

Monotonic Relationship*: The variables tend to move in the same relative direction, but not necessarily at a constant rate.*

Linear Relationship*: The variables move in the same direction at a constant rate.*

Source

```
newdf.corr(method='pearson')
```

	Source Port	Destination Port	Duration
Source Port	1.000000	0.041885	-0.074127
Destination Port	0.041885	1.000000	-0.001844
Duration	-0.074127	-0.001844	1.000000

Pearson shows us that the correlation between Source Port and Desitination Port are close to 0, showing signs that there is a weak or nonexistent linear relationship.

Now to check **Monotonic**

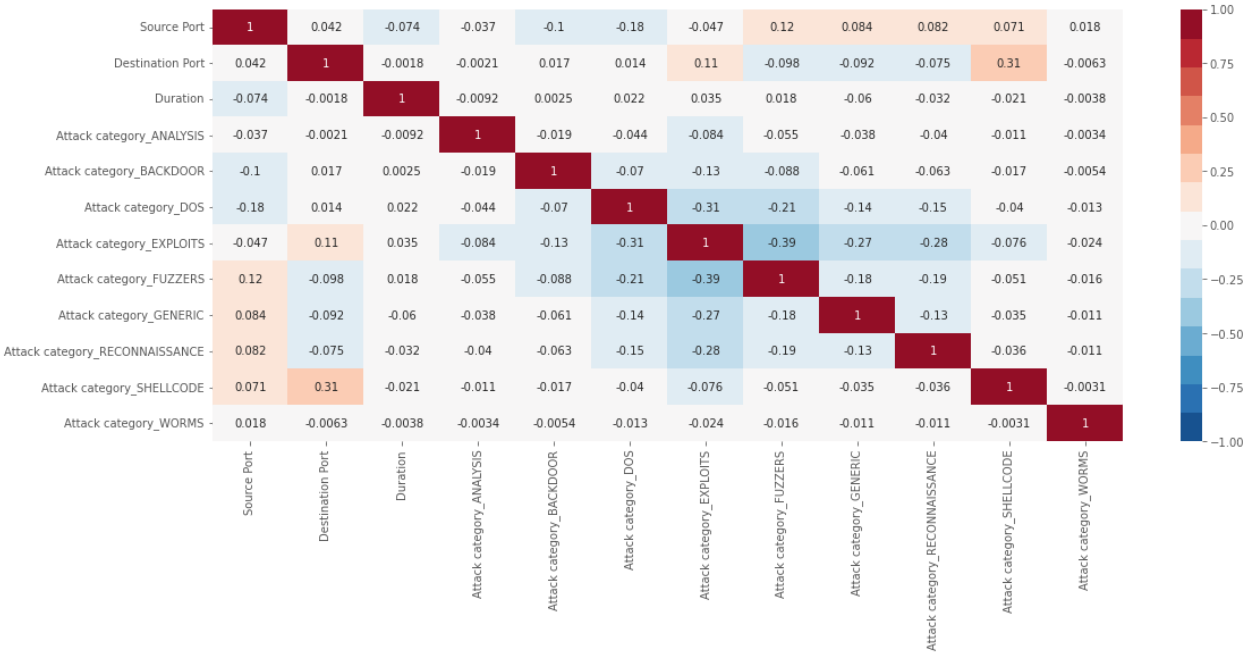
```
newdf.corr(method='spearman')
```

	Source Port	Destination Port	Duration
Source Port	1.000000	0.647630	0.291103
Destination Port	0.647630	1.000000	0.120116
Duration	0.291103	0.120116	1.000000

Using Spearman, we can see a value of 0.89, showing a strong likelihood that there is some form monotonic relationship.

The following are different ways to visually represent this data.

```
# df_dummies converts category values into a dummy or indicator value

df_dummies = pd.get_dummies(newdf, columns=['Attack category'])


plt.figure(figsize=(18,7))
sns.heatmap(df_dummies.corr(method='pearson'),
            annot=True, vmin=-1.0, vmax=1.0, cmap=sns.color_palette("RdBu_r", 15))
plt.show()
```



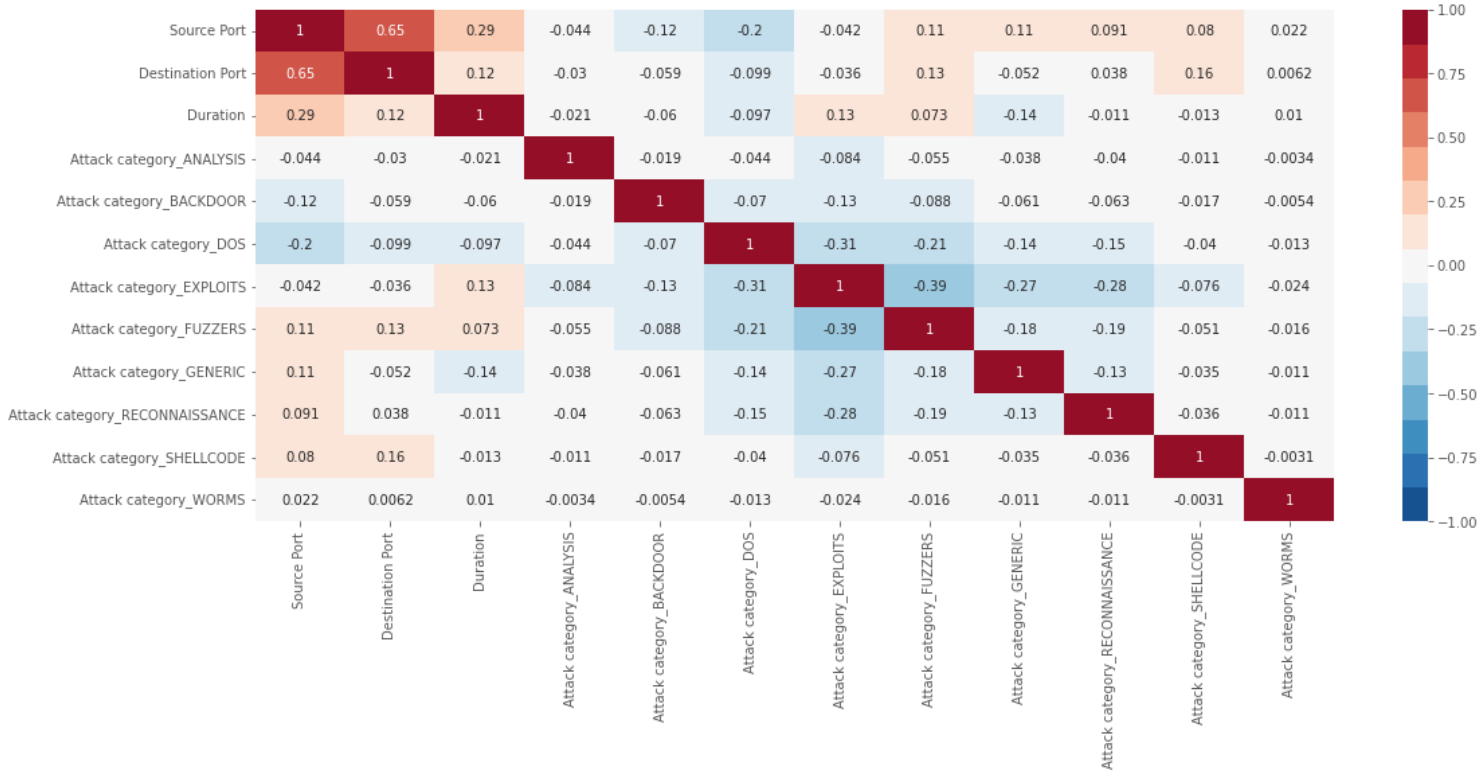
Pearson's Correlation

As seen with the overall calculation there is not much to see here, except 1 item.

- SHELLCODE & Destination Port

```
plt.figure(figsize=(18,7))
sns.heatmap(df_dummies.corr(method='spearman'),
            annot=True, vmin=-1.0, vmax=1.0, cmap=sns.color_palette("RdBu_r", 15))
```

```
plt.show()
```

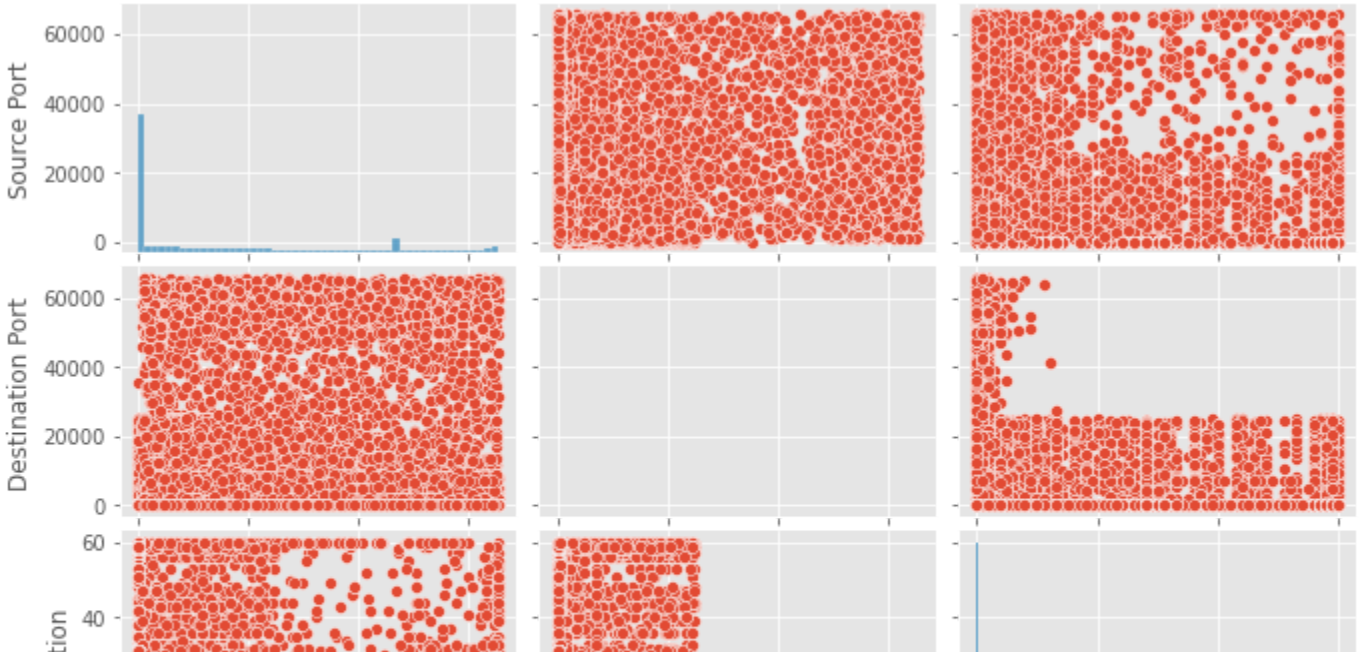


Spearman's Correlation

As we saw in the previous calculation, there is a high correlation between Destination and Source Port.

However, there seems to be something interesting with Duration as well, as you can see around the top left corner with .36 and .35.

```
g = sns.pairplot(newdf)
g.fig.set_size_inches(10,6)
plt.show()
```



The above pairplot shows us that:

- Source Port x Destination Port: Nothing in particular
- Source Port x Duration: Under 20s, high and low ports
- Destination x Duration: Low ports and upper ports short time/lower ports varied with low time concentration

Source & Destination Port Analysis

To get a better understanding of what is happening with *Source Port* and *Destination Port*, we will look to see where the attacks are happening.

We will begin by looking at the IP Addresses being attacked, and see how often they show up in the data.

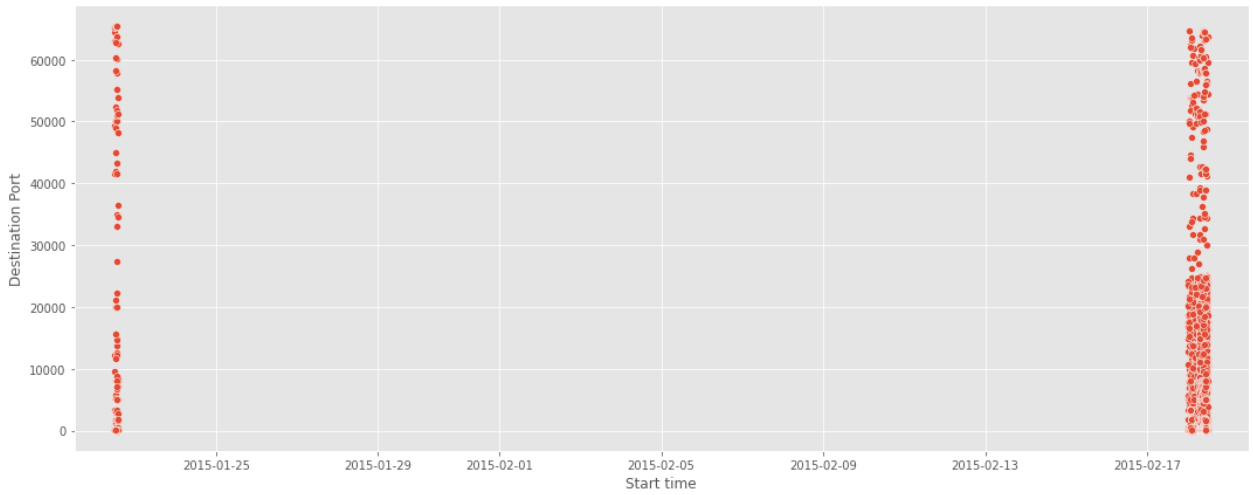
```
newdf['Destination IP'].value_counts()
```

```
149.171.126.17    35730
149.171.126.13    20570
149.171.126.19    13925
149.171.126.10    11292
149.171.126.14    10308
149.171.126.12    10288
149.171.126.15     9094
149.171.126.18     8988
149.171.126.11     7480
149.171.126.16     5892
Name: Destination IP, dtype: int64
```

The top IP Address has **43,199** attacks on it!

We will investigate this IP Address further, and start with time.

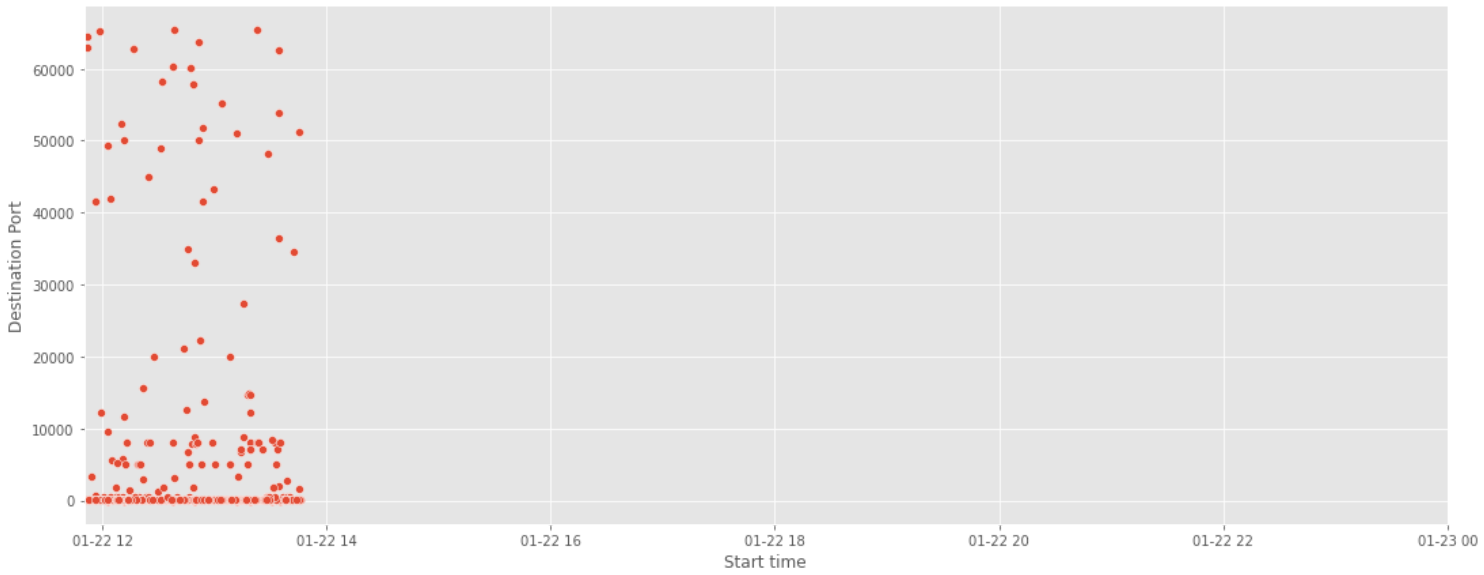
```
plt.figure(figsize=(18,7))
sns.scatterplot(x=newdf[newdf['Destination IP']=='149.171.126.17']['Start time'], y=newdf[newd
plt.xlim(left=newdf['Start time'].min()-timedelta(days=1),right=newdf['Start time'].max()+time
plt.grid(True)
plt.show()
```



We are seeing some trending here, but we still need to investigate further.

We will start with the left side of the scatterplot from above.

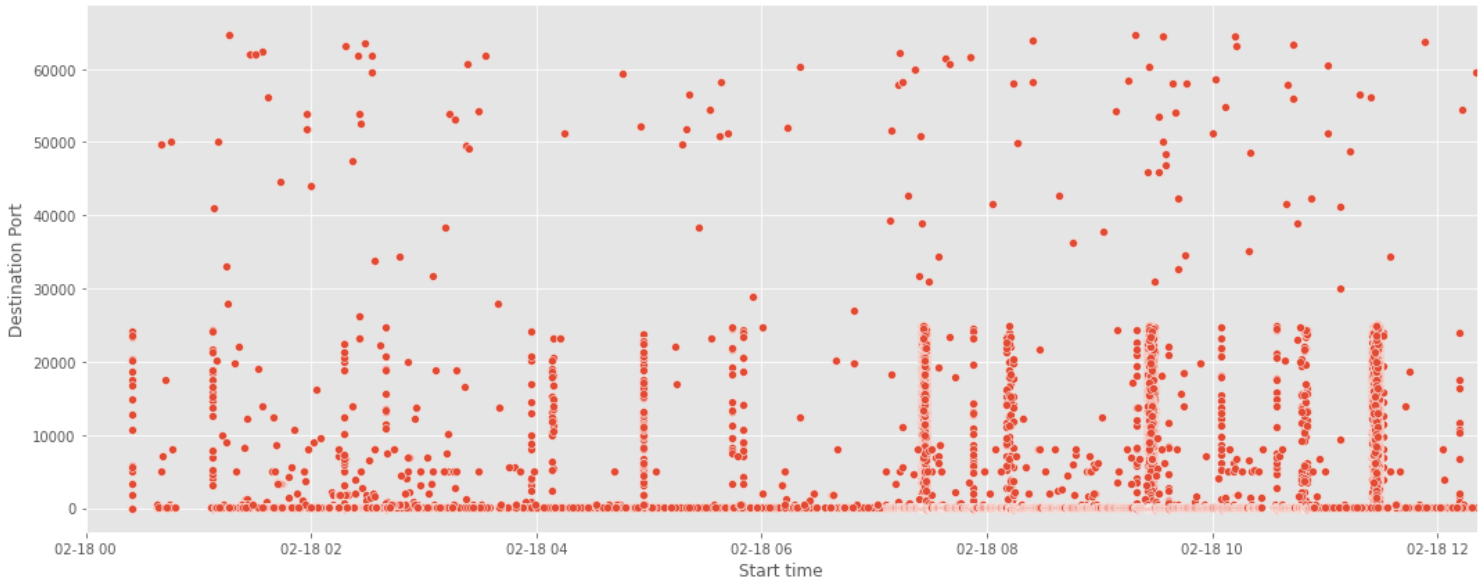
```
plt.figure(figsize=(18,7))
sns.scatterplot(x=newdf[newdf['Destination IP']=='149.171.126.17']['Start time'], y=newdf[newd
plt.xlim(left=newdf['Start time'].min(),right=datetime.strptime('15-01-23', '%y-%m-%d'))
plt.grid(True)
plt.show()
```



We can now see that there is a strong concentration of attacks on the lower ports.

Right side Analysis

```
plt.figure(figsize=(18,7))
sns.scatterplot(x=newdf[newdf['Destination IP']=='149.171.126.17']['Start time'], y=newdf[newd
plt.xlim(left=datetime.strptime('15-02-18', '%y-%m-%d'),right=newdf['Start time'].max())
plt.grid(True)
plt.show()
```



Very similar results to the left side

Ports <150

Since we have a lot of overlap on the x-axis, I would like to get a better idea of what is happening down where the concentration appears to by the highest.

```
plt.figure(figsize=(20,7))
sns.scatterplot(x='Start time', y='Destination Port', hue='Attack category',
                data=newdf[(newdf['Destination IP']=='149.171.126.17')&(newdf['Destination Por
                s=65)
plt.xlim(left=datetime.strptime('15-02-18 00:00:00', '%y-%m-%d %H:%M:%S'),
         right=datetime.strptime('15-02-18 13:00:00', '%y-%m-%d %H:%M:%S'))
plt.grid(True)
plt.show()
```

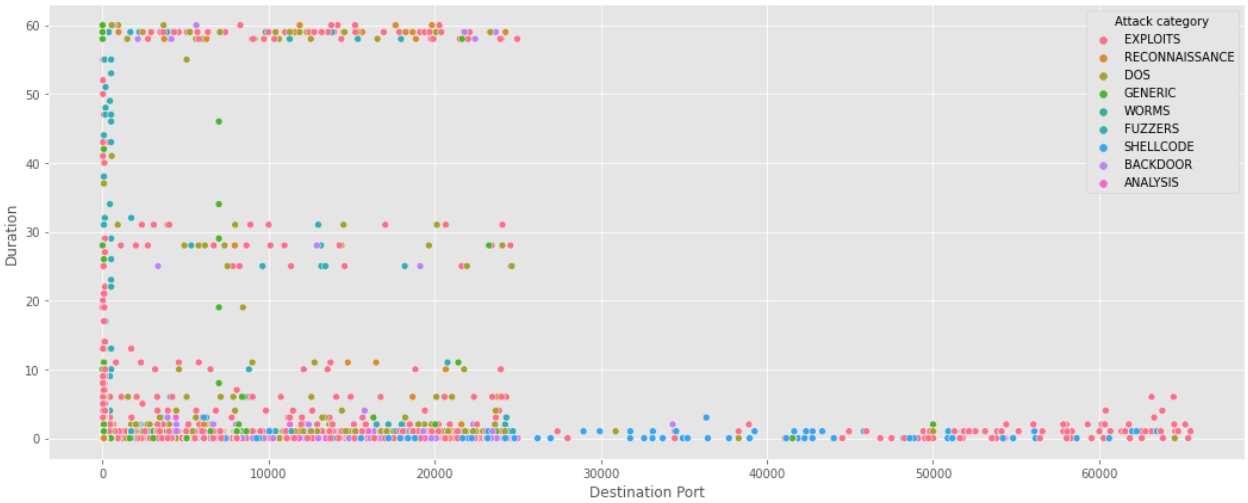
We can now see a lot of

- ***Backdoor*** attacks happening at the 21 and 25 port, those being the ftp and smtp (mail ports).
- ***Port 80***- *HTTP* has alot of everything, esspecially *Exploits*
- ***Port 110***- *POP* has alot of everything, esspecially *Reconnaissance*

Duration & Destination Port Analysis

To begin we will use a scatterplot to see if there are any insights we can gleen from the data

```
plt.figure(figsize=(18,7))
sns.scatterplot(x='Destination Port', y='Duration', hue='Attack category', data=newdf[newdf['D
plt.grid(True)
plt.show()
```

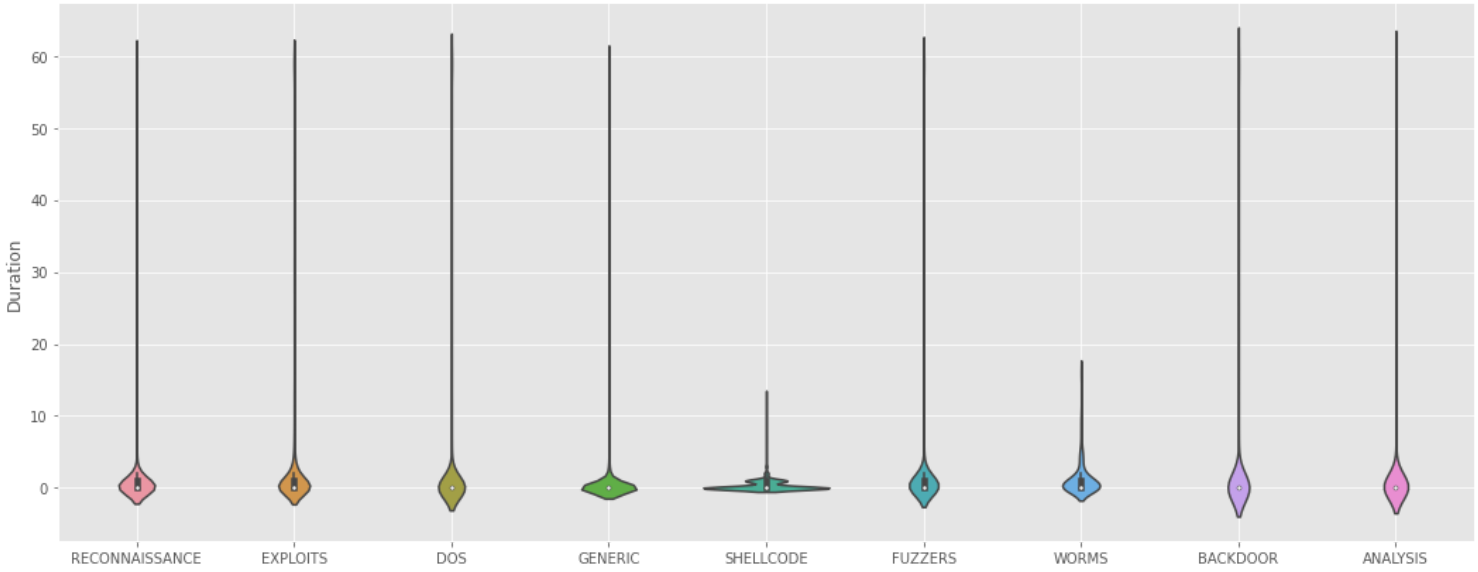


Most of the ports being attacked for long durations appear to be in the very low end of the values.

Below is a violinplot, similar to a boxplot.

- **Boxplot** shows summary statistics, e.g. mean, median, etc.
- **Violinplot** shows summary statistics & full data distribution

```
plt.figure(figsize=(18,7))
sns.violinplot(x='Attack category', y='Duration', data=newdf)
plt.grid(True)
plt.show()
```



Most of the Attack Categories appear to be normally distributed, except for SHELLCODE.

- This violinplot is showing us that SHELLCODE his bimodal

The following code blocks will be used to set up a heatmap graph and pivot table.

```
def heatmap_graph(df, xlabel, ylabel, title):
    plt.figure(figsize=(18,8))
    ax = sns.heatmap(df)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.xticks(rotation=90)
    plt.yticks(rotation=0)
    plt.show()
```

```
newdf["Start time"][1].hour
```

```
    11
```

```
df_pivot = newdf.copy()
df_pivot['hour'] = df_pivot.apply(lambda row: '0'*(2-len(str(row['Start time'].hour)))+str(row
```

```
# Creating a new df for the pivot table

df_pivot[:5]
```

	Attack category	Attack subcategory	Protocol	Source IP	Source Port	Destination IP	D
0	RECONNAISSANCE	HTTP	TCP	175.45.176.0	13284	149.171.126.16	
1	EXPLOITS	Unix 'r' Service	UDP	175.45.176.3	21223	149.171.126.18	
2	EXPLOITS	Browser	TCP	175.45.176.2	23357	149.171.126.16	
		Miscellaneous					

Pivot Table

4	EXPLOITS	Cisco IOS	TCP	175.45.176.2	26939	149.171.126.10	

```
df_p1 = pd.pivot_table(df_pivot,values='Attack Name', index=['hour'], columns=['Attack categor
df_p1
```

Attack category	ANALYSIS	BACKDOOR	DOS	EXPLOITS	FUZZERS	GENERIC	RECONNAISSANCE	SHELLCODE
hour								
00:00:00	4	18	148	572	435	63	190	25
01:00:00	219	1334	3001	7428	5017	1813	2083	118
02:00:00	77	109	667	2970	2314	1075	1176	137
03:00:00	130	31	2203	5756	1502	557	1678	93
04:00:00	13	23	159	668	553	361	261	30
05:00:00	53	161	740	2366	620	484	389	34
06:00:00	25	25	168	744	554	237	282	28
07:00:00	231	687	4006	6943	2830	2285	2203	134
08:00:00	87	129	794	3368	2496	1203	1266	137
09:00:00	248	756	4253	8605	3185	2346	2522	138
10:00:00	88	136	693	2884	2184	1094	1046	118
11:00:00	64	139	995	2361	1347	511	747	63
12:00:00	256	296	680	3406	3021	1104	1191	147
13:00:00	61	31	228	1696	1507	1648	822	101

The pivot table shows us some rather random looking numbers but with some form of pattern as well.

Let us try the heat map

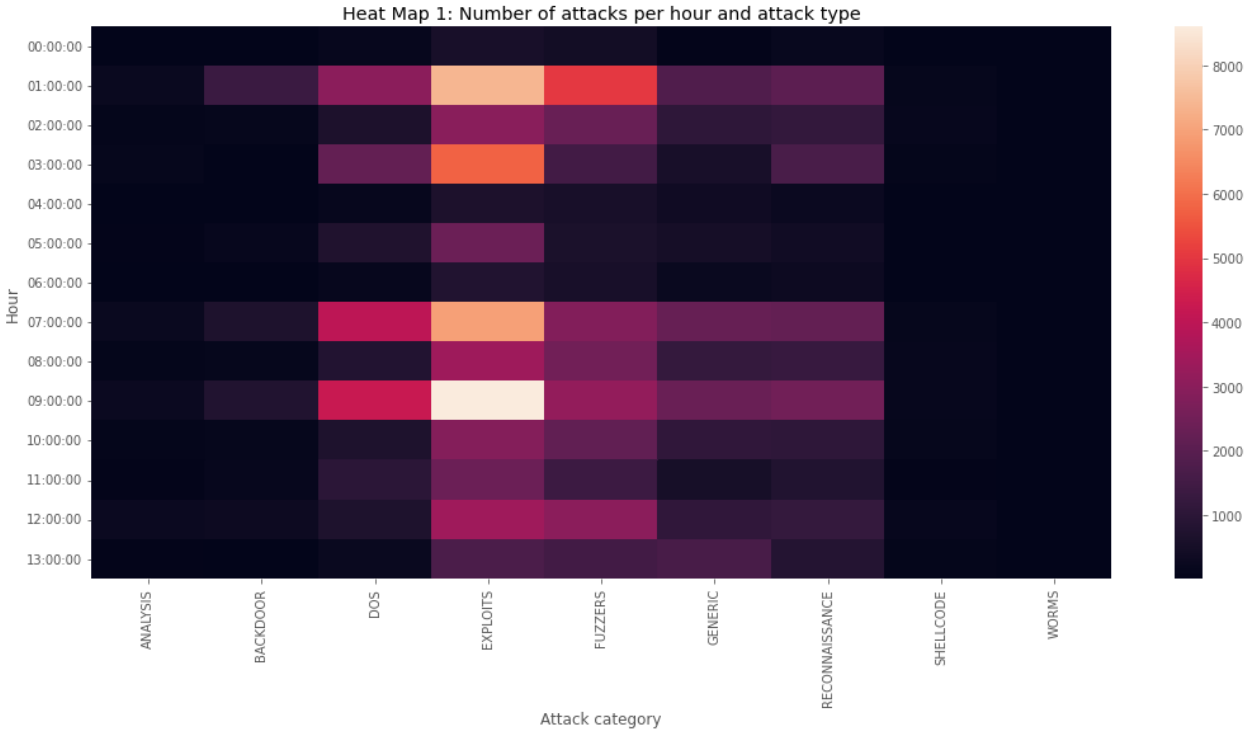
Heat Maps

The following heat map shows the number of attacks by attack type.

- Black will be <1000 where white is ~10,000 attacks

Heat Map 1

```
heatmap_graph(df = df_p1, xlabel = 'Attack category', ylabel = 'Hour', title = 'Heat Map 1: Nu
```



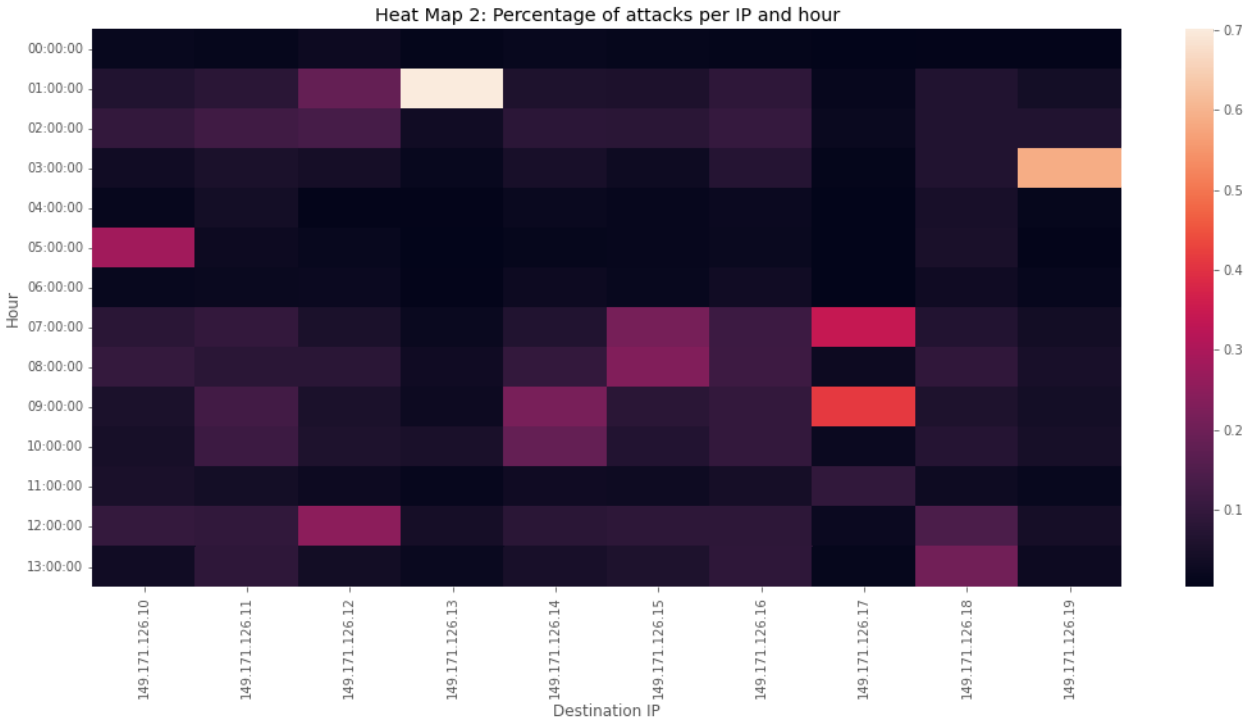
Insights from Heat Map 1:

- Analysis, Backdoor, Shellcode & Worms appear to be consistent throughout the day

- The remaining categoriess appear to have higher attacks on the odd hours of the day
- Exploits and DOS have shown to have the strongest patterning

Heat Map 2

```
df_p2 = pd.pivot_table(df_pivot, values='Attack Name', index=['hour'], columns=['Destination I
heatmap_graph(df = df_p2/df_p2.sum(), xlabel = 'Destination IP', ylabel = 'Hour', title = 'Hea
```



Insights from Heat Map 2:

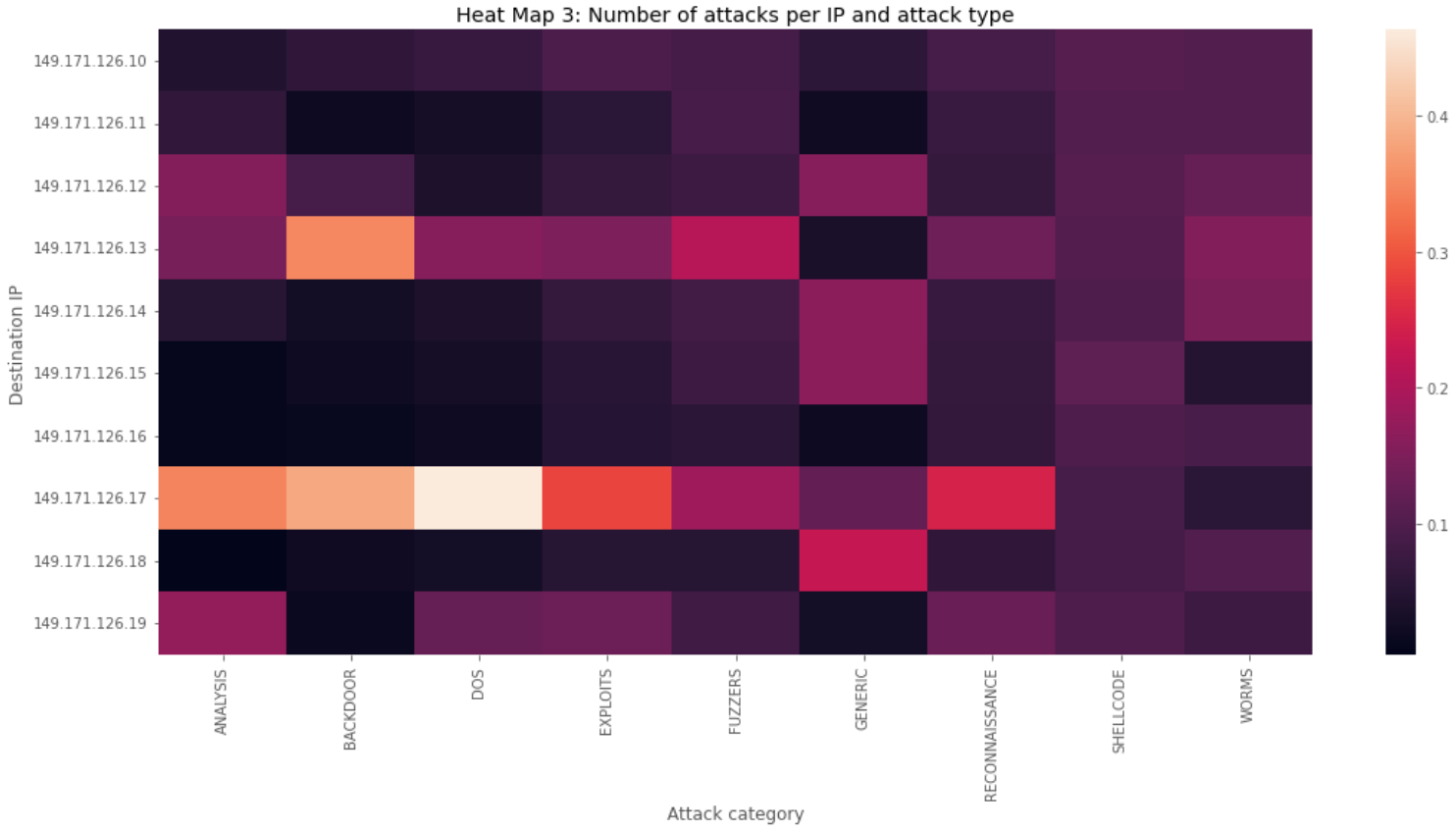
- IPv4 149.171.126.10 had a strong pattern showing of attacks at **5:00AM**
- IPv4 149.171.126.13 had a strong pattern showing of attacks at **1:00AM**
- IPv4 149.171.126.17 showed increased attacks at **7:00AM**, **9:00AM** & **11:00AM**
- IPv4 149.171.126.19 had a strong pattern showing of attacks at **3:00AM**

Attack Category Analysis

Heat Map 3

Since it is now clear that specific IPv4 addresses are being targetted, we will look more specifically at this data.

```
df_p3 = pd.pivot_table(df_pivot, values='Attack Name', index=['Destination IP'], columns=['Att
heatmap_graph(df = df_p3/df_p3.sum(), xlabel = 'Attack category', ylabel = 'Destination IP', t
```



Insights from Heat Map 3:

- **IPv4 149.171.126.17** had a strongest looking pattern across the 10 main IP addresses targetted
- **DOS**, **Backdoor** & **Exploits** were the most concentrated attacks
- **Shellcode** has the most consistent attacks over the time period

Pair-wise T-test

We will now look at the relationship between variables for Source Port and Destination Port, but by each Attack Category.

- *Similar to before, except we are breaking it down by Attack Category*

```
for attack in list(newdf['Attack category'].unique()):
    df_attack = newdf[newdf['Attack category'] == attack].copy()
    statistic, pvalue = stats.ttest_ind(df_attack['Source Port'], df_attack['Destination Port'
    print('p-value in T-test for ' + attack + ' | ' + str(pvalue))
```

```
p-value in T-test for RECONNAISSANCE | 0.0
p-value in T-test for EXPLOITS | 0.0
p-value in T-test for DOS | 0.0
p-value in T-test for GENERIC | 0.0
p-value in T-test for SHELLCODE | 0.11566083026535003
p-value in T-test for FUZZERS | 0.0
p-value in T-test for WORMS | 7.769085694955551e-33
p-value in T-test for BACKDOOR | 1.6545297907029784e-09
p-value in T-test for ANALYSIS | 1.0475274074721863e-53
```

The p-*values* of all but one attack category are very close to 0.0.

- This means that the attacks have been directed to the specific ports *(except Shellcode)*

Shellcode: We cannot reject null hypothesiss, therefore there is a defined randomness, which means the source and destination ports have similar averages.

To verify this statement, we will make use of a contingency table which allows to relate the count of a certain pair of variables, similar to how we saw the .pivot_table()

```
df_crosstab = pd.crosstab(newdf['Attack category'], newdf['Destination Port'])
df_crosstab
```

Destination Port	0	3	5	6	8	9	10	11	13	15	16	17	18	19	20	21	22	23	25
Attack category																			
ANALYSIS	850	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
BACKDOOR	2587	0	0	0	0	0	0	0	0	0	0	0	0	1	0	5	0	0	0
DOS	10721	0	0	0	1	1	1	0	0	1	0	0	2	0	0	40	0	12	200

From the contingency table using Attack type and Target port, we can see that the individual counts are not uniform.

- This helps to affirm our inference that there is potentially some interaction between these two variables

---

We will need to test to see if these variations are actual differences, or outcomes of randomness in the data.

SHELLCODE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Chi-square Test

9 rows × 10000 columns

While there are many ways to test the relationship between our variables, we will focus on the **Chi-square test**, as it is one of the more widely used tests, and easy to perform.

The null hypothesis for the Chi-square test is as follows:

$$H_0 : \text{The attack category is independent of the destination port}$$

```
chi2, p_value, dof, expected = chi2_contingency(df_crosstab)
print("p-value of Chi-square test for Attack category vs. Destination Port =", p_value)
```

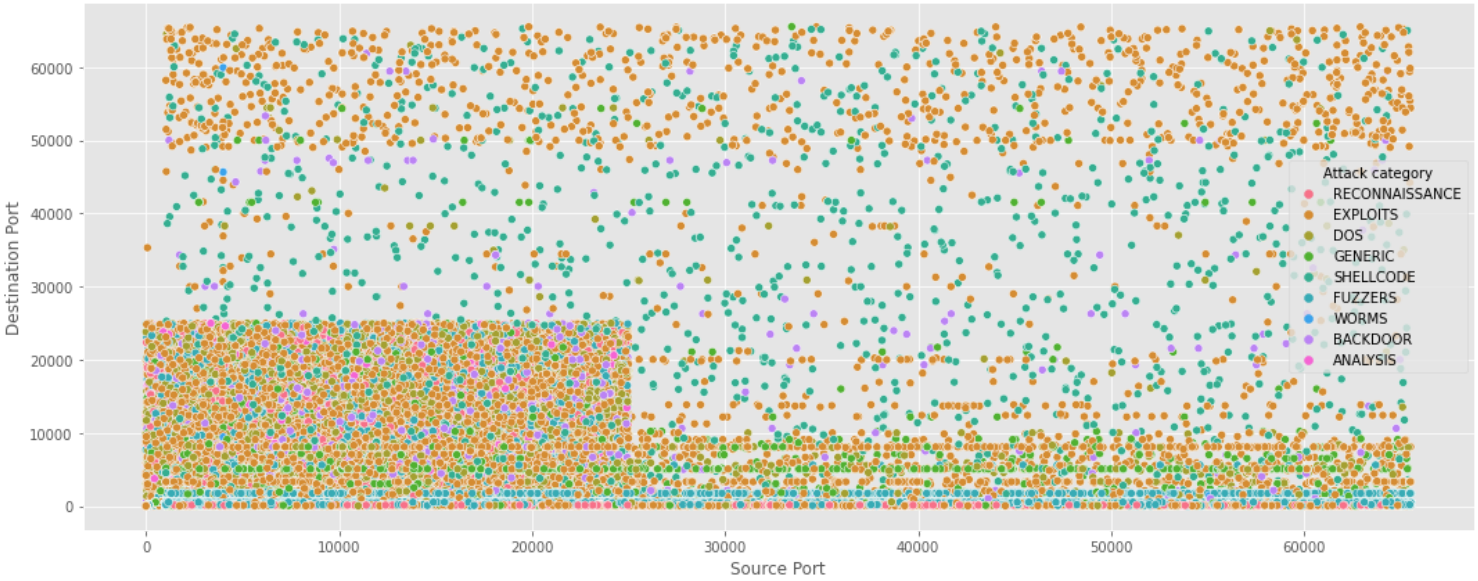
```
p-value of Chi-square test for Attack category vs. Destination Port = 0.0
```

As previously seen in the Hypothesis testing, our value for the Chi-square test is extremely small, and will be shown to us in python as 0.0

- Therefore, the destination port appearss to be somewhat dependant on the Attack category used.

To better visualize this relationship, we will use a scatterplot function to show us the Source Port along the x axis, the Destination Port on the y axis, and use the different attack categories for the colour (or hue).

```
plt.figure(figsize=(18,7))
sns.scatterplot(x='Source Port',y='Destination Port', hue='Attack category',data=newdf)
plt.show()
```



Emerging Patterns

- Strong concentration of attacks on Destination Ports <10,000 and >50,000
- Strongest concentration at the lowest ports <150
- Shellcode appears to be equally distributed

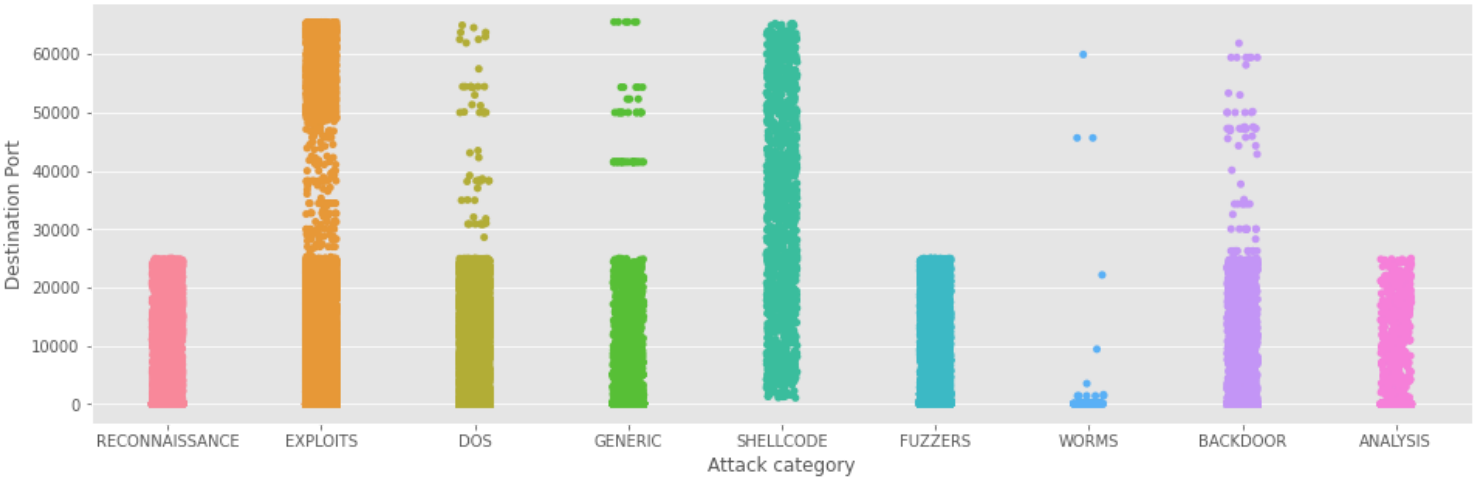
Strip Plot

To see this relationship more in depth, we can visualize the the distribution of the Attack Categories and Destination Ports with a strip diagram using the `.stripplot()` function:

```
# Source ports
plt.figure(figsize=(16,5))
sns.stripplot(x='Attack category',y='Source Port',data=newdf)
plt.show()
```

As we saw with the data previously, there is a pretty even spread of the use of Source Port for all attack types.


```
# Destination ports
plt.figure(figsize=(16,5))
sns.stripplot(x='Attack category',y='Destination Port',data=newdf)
plt.show()
```



Similar to Soure Ports, Destination Ports are showing us the same results ass the data.

- Recon, DOS, Gen, Fuz, Worms, Ana all have a strong trend to target low number ports
- Exploits are relatively spread out, but with concentrations at the upper and lower bounds
- Shellcode is relatively uniformly distributed

IPv4 Analysis

We will now look at the Attack Categories by

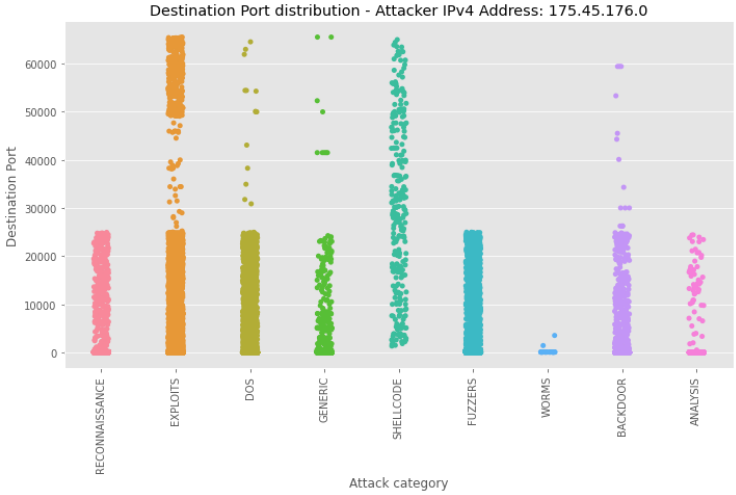
```
# Find unique Attacker IPv4

list(newdf['Source IP'].unique())
```

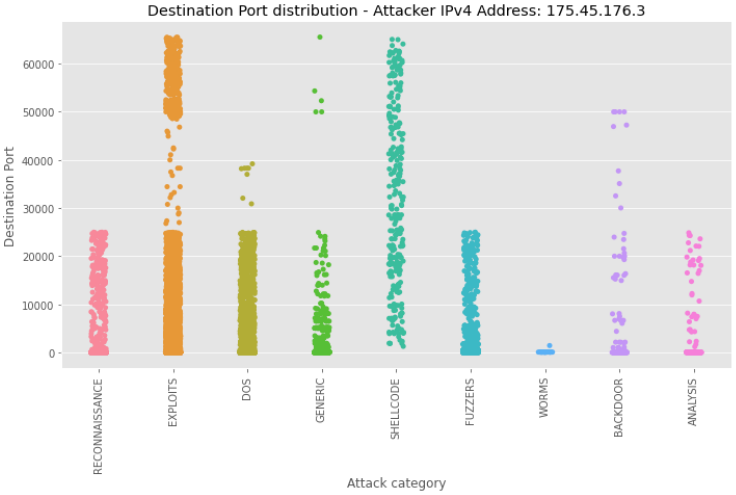
```
['175.45.176.0',
 '175.45.176.3',
 '175.45.176.2',
 '175.45.176.1',
 '187.48.276.1',
 '175.45.171.3',
 '175.45.186.6',
 '175.45.195.1',
 '187.48.276.0',
 '175.45.171.2',
 '175.45.186.5']
```

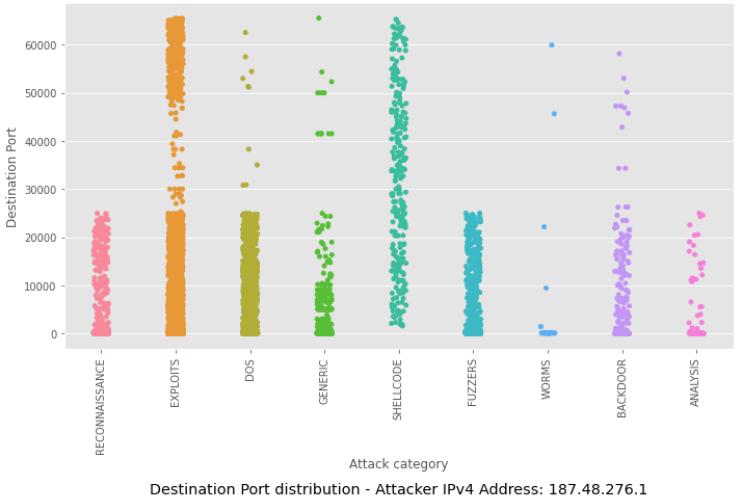
Attacker IPv4

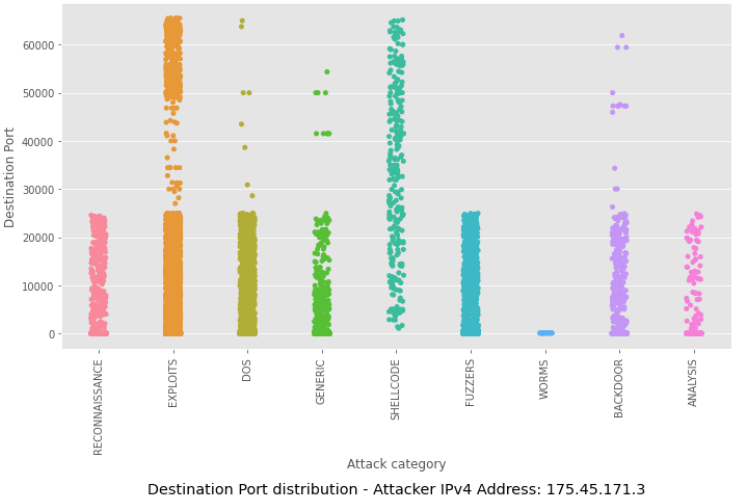
Below will split out the four unique IP Addresses we have for attackers, and compare the distribution of Attack Categories to the Destination Ports.

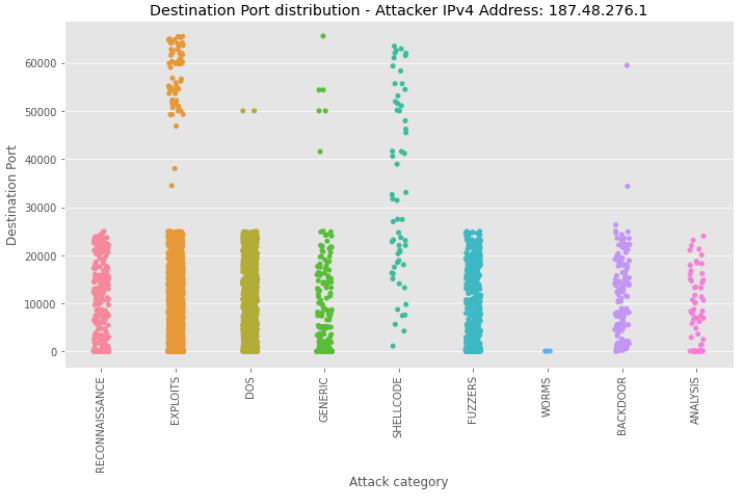
```
ips = list(newdf['Source IP'].unique())
f, axes = plt.subplots(6,2)
f.set_figheight(40)
f.set_figwidth(20)

labels = list(newdf['Attack category'].unique())
for i, ip in enumerate(ips):
    sns.stripplot(x='Attack category',y='Destination Port',data=newdf[newdf['Source IP'] == ip
    axes[int(i/2)][i%2].set_xlabel('Attack category')
    axes[int(i/2)][i%2].set_ylabel('Destination Port')
    axes[int(i/2)][i%2].set_title('Destination Port distribution - Attacker IPv4 Address: ' +
    axes[int(i/2)][i%2].set_xticklabels(labels,rotation=90)
plt.tight_layout()
plt.show()
```

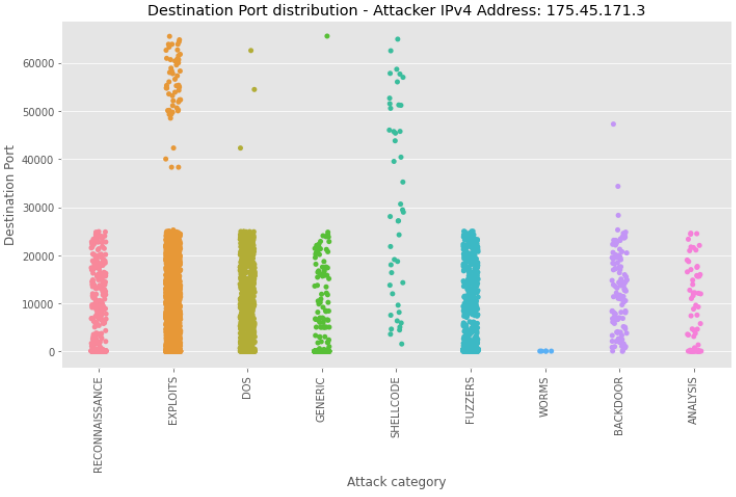

Destination Port distribution - Attacker IPv4 Address: 175.45.176.0
Destination Port
Attack category
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS

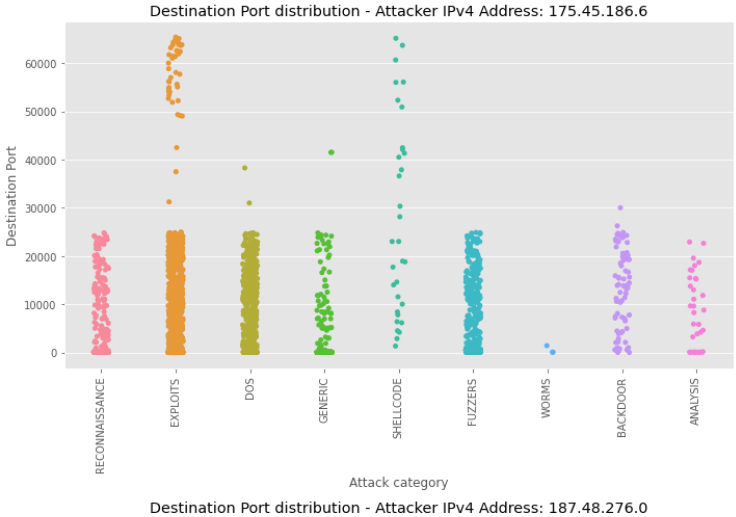

Destination Port distribution - Attacker IPv4 Address: 175.45.176.3
Destination Port
Attack category
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS

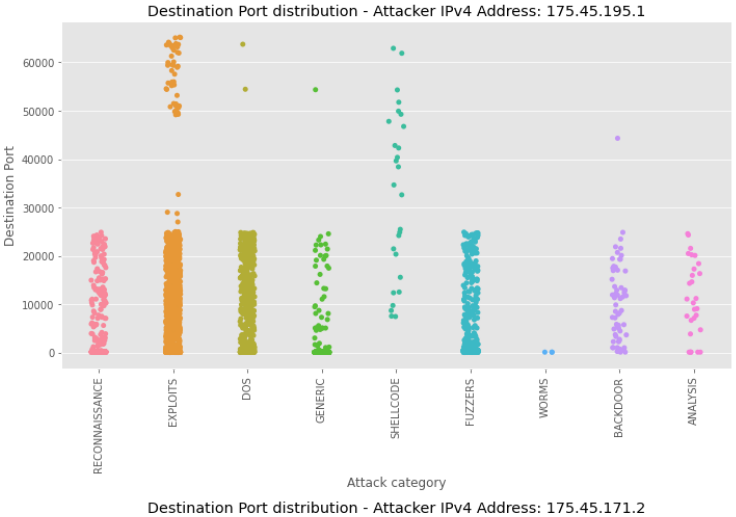

Destination Port distribution - Attacker IPv4 Address: 175.45.176.2
Destination Port
Attack category
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS


Destination Port distribution - Attacker IPv4 Address: 175.45.176.1
Destination Port
Attack category
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS


Destination Port distribution - Attacker IPv4 Address: 187.48.276.1
Destination Port
Attack category
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS

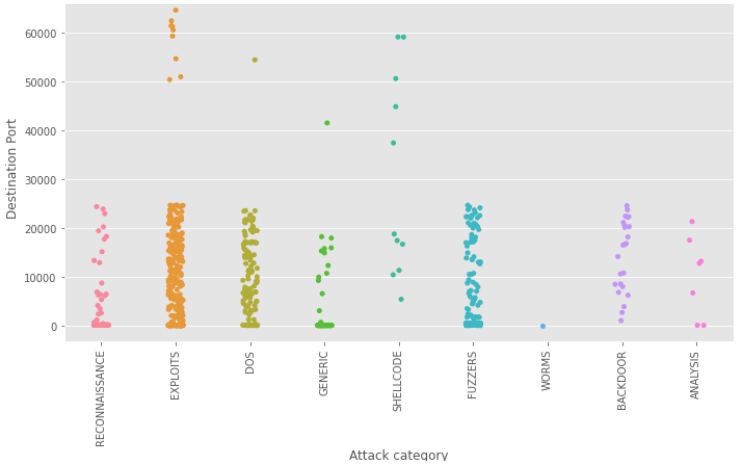

Destination Port distribution - Attacker IPv4 Address: 175.45.171.3
Destination Port
Attack category
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS

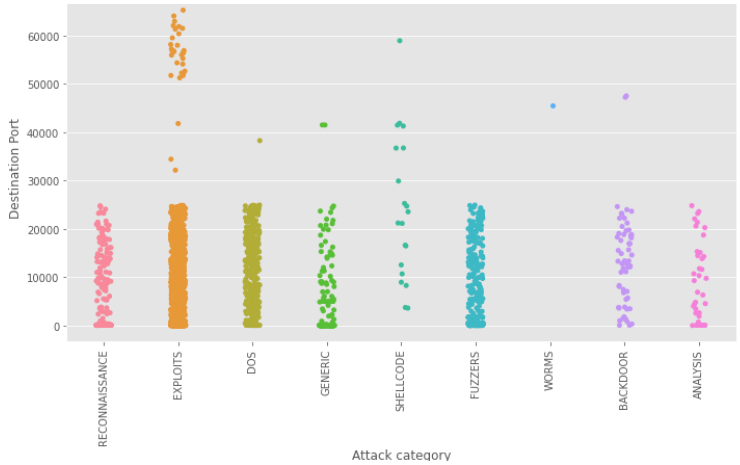

Destination Port distribution - Attacker IPv4 Address: 175.45.186.6
Destination Port
Attack category
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS


Destination Port distribution - Attacker IPv4 Address: 175.45.195.1
Destination Port
Attack category
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS

Destination Port distribution - Attacker IPv4 Address: 187.48.276.0

Destination Port distribution - Attacker IPv4 Address: 175.45.171.2


Destination Port
60000
50000
40000
30000
20000
10000
0
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category


Destination Port
60000
50000
40000
30000
20000
10000
0
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category

While there are obviously going to be some variance between the four graphs, we can clearly see similarity among the four different IPv4 values explored.

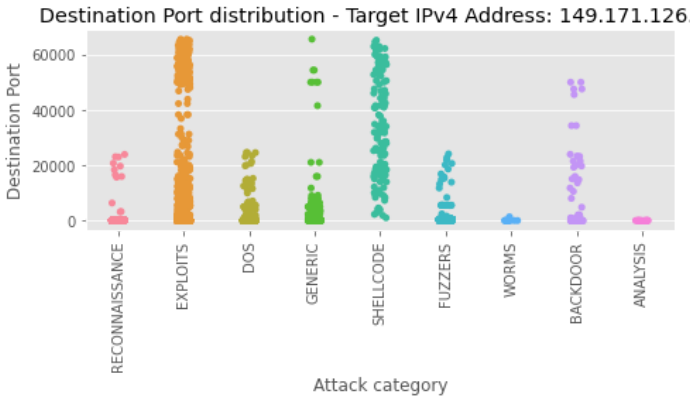
Victim IPv4

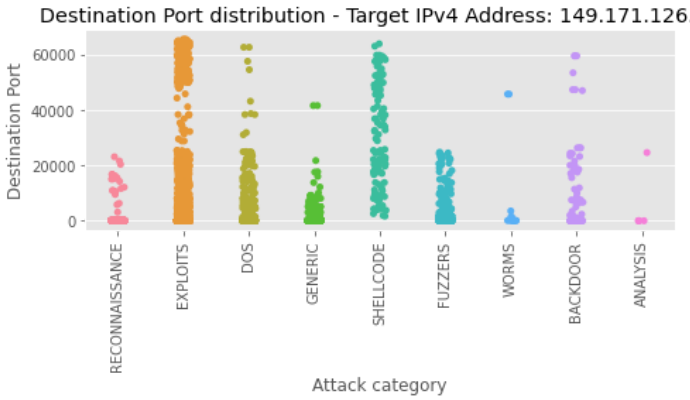
We will complete the same exercise as above, but for the victim IP's.

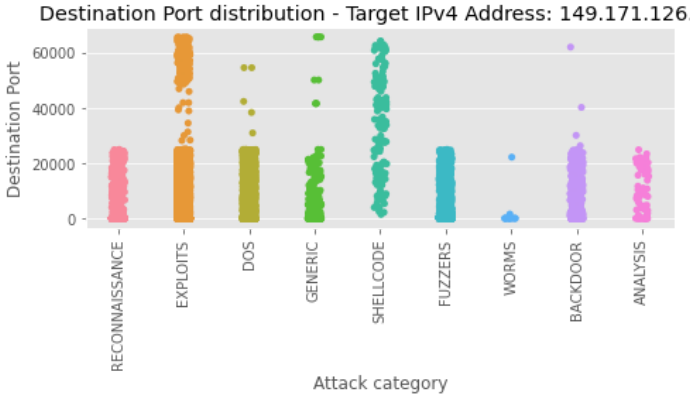
```
list(newdf['Destination IP'].unique())
```

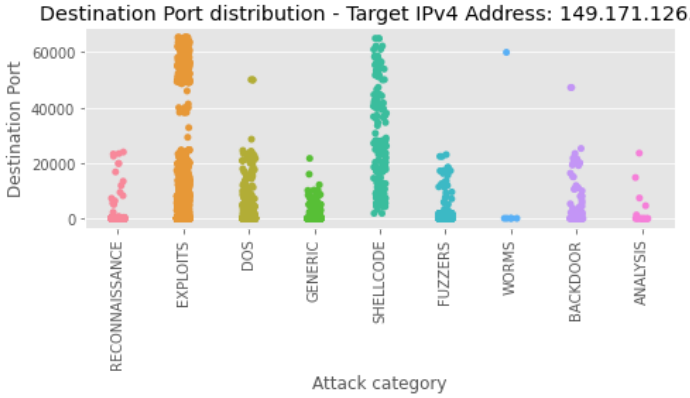
```
['149.171.126.16',
 '149.171.126.18',
 '149.171.126.10',
 '149.171.126.15',
 '149.171.126.14',
 '149.171.126.12',
 '149.171.126.13',
 '149.171.126.11',
 '149.171.126.17',
 '149.171.126.19']
```

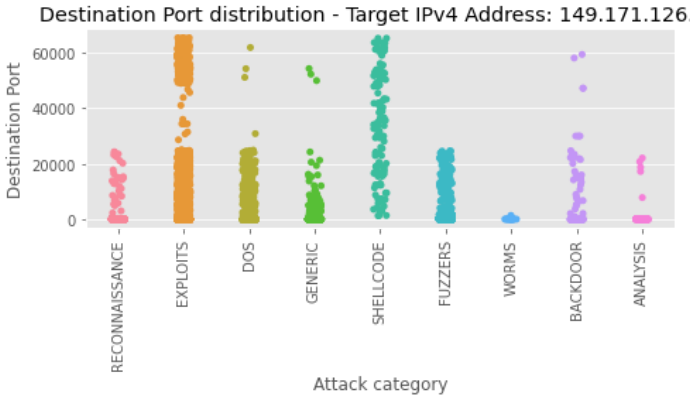
```
ips = list(newdf['Destination IP'].unique())
f, axes = plt.subplots(5, 2)
f.set_figheight(20)
f.set_figwidth(15)

labels = list(newdf['Attack category'].unique())

for i, ip in enumerate(ips):
    sns.stripplot(x='Attack category',y='Destination Port',data=newdf[newdf['Destination IP']
    axes[int(i/2)][i%2].set_xlabel('Attack category')
    axes[int(i/2)][i%2].set_ylabel('Destination Port')
    axes[int(i/2)][i%2].set_title('Destination Port distribution - Target IPv4 Address: ' + ip
    axes[int(i/2)][i%2].set_xticklabels(labels,rotation=90)
plt.tight_layout()
plt.show()
```

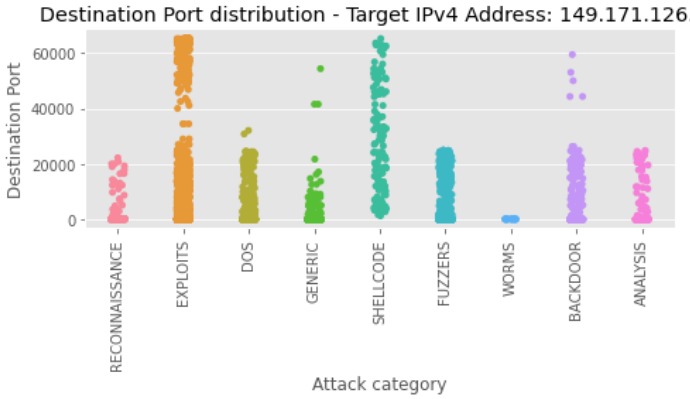

Destination Port distribution - Target IPv4 Address: 149.171.126.16
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category

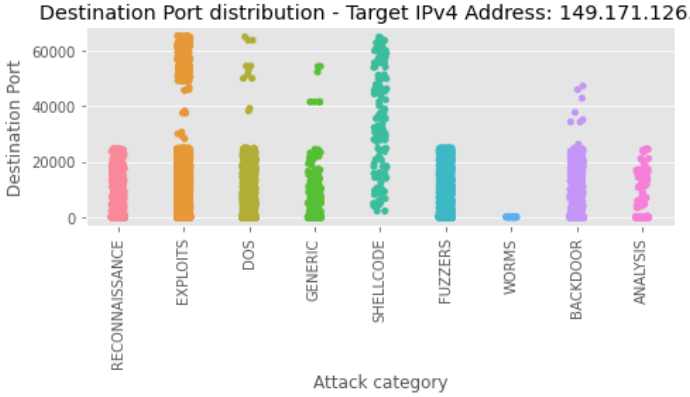

Destination Port distribution - Target IPv4 Address: 149.171.126.18
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category

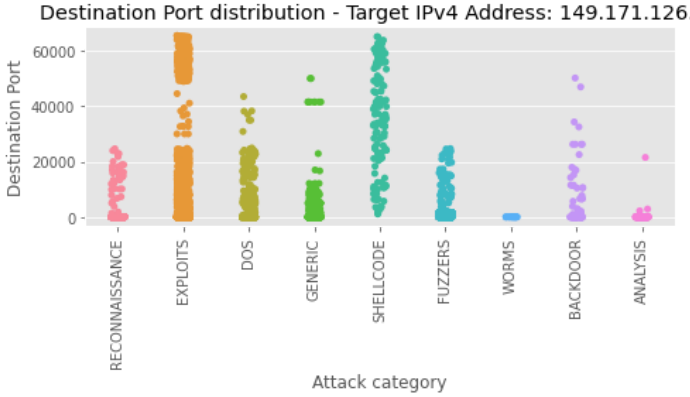

Destination Port distribution - Target IPv4 Address: 149.171.126.10
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category

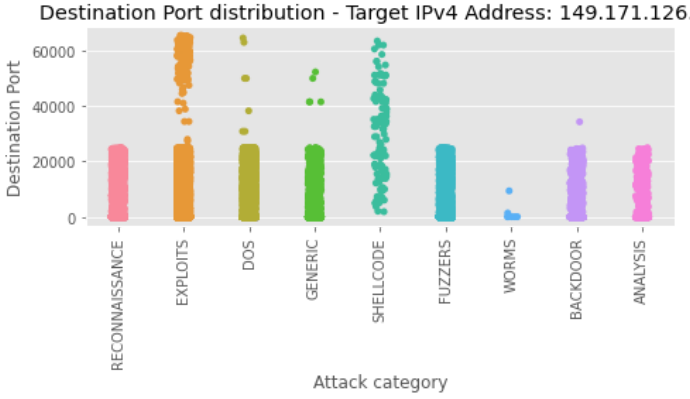

Destination Port distribution - Target IPv4 Address: 149.171.126.15
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category

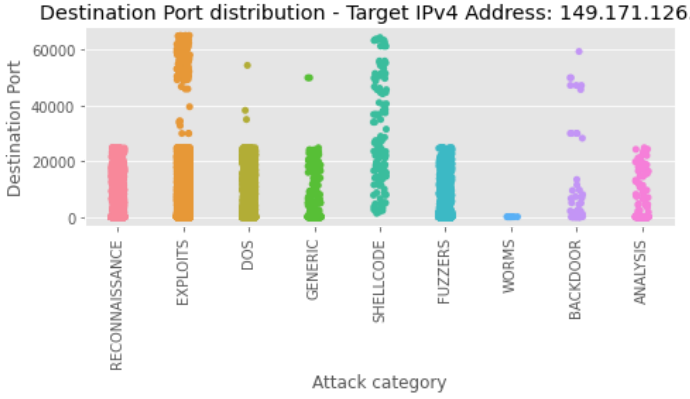

Destination Port distribution - Target IPv4 Address: 149.171.126.14
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category


Destination Port distribution - Target IPv4 Address: 149.171.126.12
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category


Destination Port distribution - Target IPv4 Address: 149.171.126.13
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category


Destination Port distribution - Target IPv4 Address: 149.171.126.11
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category


Destination Port distribution - Target IPv4 Address: 149.171.126.17
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category


Destination Port distribution - Target IPv4 Address: 149.171.126.19
Destination Port
0
20000
40000
60000
RECONNAISSANCE
EXPLOITS
DOS
GENERIC
SHELLCODE
FUZZERS
WORMS
BACKDOOR
ANALYSIS
Attack category