In [1]:

```
## --- 1.1.1 --- ##
from pyspark import SparkConf
from pyspark.sql import SparkSession

# run Spark in local mode with as many working processors as logical cores on the machine
master = "local[*]"
app_name = "Linux_ML"
spark_conf = SparkConf().setMaster(master).setAppName(app_name)
```

In [2]:

```
## --- 1.1.2 --- ##
# each file size is 60 - 65 MB, set maxPartitionBytes to 32MB so that each file can have
2 partitions
maxPartitionBytes = 32000000
spark = SparkSession.builder.config(conf=spark_conf).getOrCreate()
spark.conf.set("spark.sql.files.maxPartitionBytes", maxPartitionBytes)

sc = spark.sparkContext
sc.setLogLevel('ERROR')
```

In [16]:

```
## --- 1.2.1 --- ##
from pyspark.sql.types import FloatType, StringType, StructType, StructField

# set explicit schema to before reading in csv files
memory_schema = StructType([
    StructField("ts", FloatType()),
    StructField("PID", FloatType()),
    StructField("MINFLT", FloatType()),
    StructField("MAJFLT", FloatType()),
    StructField("VSTEXT", FloatType()),
    StructField("VSIZE", FloatType()),
    StructField("RSIZE", FloatType()),
    StructField("VGROW", FloatType()),
    StructField("RGROW", FloatType()),
    StructField("MEM", FloatType()),
    StructField("CMD", StringType()),
    StructField("attack", FloatType()),
    StructField("type", StringType())
])

process_schema = StructType([
    StructField("ts", FloatType()),
    StructField("PID", FloatType()),
    StructField("TRUN", FloatType()),
    StructField("TSLPI", FloatType()),
    StructField("TSLPU", FloatType()),
    StructField("POLI", StringType()),
    StructField("NICE", FloatType()),
    StructField("PRI", FloatType()),
    StructField("RTPR", FloatType()),
    StructField("CPUNR", FloatType()),
    StructField("Status", StringType()),
    StructField("EXC", FloatType()),
    StructField("State", StringType()),
    StructField("CPU", FloatType()),
    StructField("CMD", StringType()),
    StructField("attack", FloatType()),
    StructField("type", StringType())
])

# read in memory csv files with header and schema above, change null value to 'NA' as rea
ding in.
df_memory = spark.read.load("data/linux_memory_*.csv",
```

```
                     format="csv", nullValue='NA', schema=memory_schema, header="true")

df_process = spark.read.load("data/linux_process_*.csv",
                     format="csv", nullValue='NA', schema=process_schema, header="true")

# cache two tables
df_memory = df_memory.cache()
df_process = df_process.cache()

# row count
print('no of row in memory csv: ', df_memory.count())
print('no of row in process csv: ', df_process.count())
```

```
no of row in memory csv:  2000000
no of row in process csv:  1927968
```

In [20]:

```
## --- 1.2.2 --- ##
# check null / missing values for each dataframe
from pyspark.sql.functions import isnan, when, count, col
df_memory.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df_memory
.columns]).show()
df_process.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df_proce
ss.columns]).show()

df_memory.describe().toPandas().head()
```

```
+---+---+------+------+------+-----+-----+-----+-----+---+---+------+----+
| ts|PID|MINFLT|MAJFLT|VSTEXT|VSIZE|RSIZE|VGROW|RGROW|MEM|CMD|attack|type|
+---+---+------+------+------+-----+-----+-----+-----+---+---+------+----+
|  0|  0|  9737|  8800|  8800|    0| 9728|49552| 9737|  0|  0|     0|   0|
+---+---+------+------+------+-----+-----+-----+-----+---+---+------+----+

+---+---+----+-----+-----+----+----+---+----+-----+------+---+-----+---+---+------+----
+
| ts|PID|TRUN|TSLPI|TSLPU|POLI|NICE|PRI|RTPR|CPUNR|Status|EXC|State|CPU|CMD|attack|type|
+---+---+----+-----+-----+----+----+---+----+-----+------+---+-----+---+---+------+----
+
|  0|  0|   0|    0|    0|   0|   0|  0|   0|    0|     0|  0|    0|  0|  0|     0|   0|
+---+---+----+-----+-----+----+----+---+----+-----+------+---+-----+---+---+------+----
+
```

Out[20]:

	summary	ts	PID	MINFLT	MAJFLT	VSTEXT	
0	count	2000000	2000000	1990263	1991200	1991200	
1	mean	1.55624581707872E9	4999.360446	404.51371904115183	1108.8663392662706	2813.1901889062333	4496.41
2	stddev	984463.3693605846	4887.313351921498	17185.876916004923	5187.185230568393	8192.289024855518	9046.33
3	min	1.55421683E9	1007.0	0.0	0.0	0.0	
4	max	1.55835571E9	53096.0	8050000.0	107776.0	99992.0	

In [5]:

```
## --- 1.2.2 --- ##
from pyspark.sql.functions import col, mean

# according to the null / missing value check above, there are some values needed to be t
ransformed in df_memory
# calculate the mean value for those columns containing null or missing values
# then retrieve the tuple containing all the avg values in the collected list
mean_memory = df_memory.select(mean(col("MINFLT")), mean(col("MAJFLT")), mean(col("VSTEXT
")),
                 mean(col("RSIZE")), mean(col("VGROW")), mean(col("RGROW"))).collect()[0
]
```

```
# transform null / missing value to mean value of their column
df_memory = df_memory.na.fill({'MINFLT': mean_memory[0], 'MAJFLT': mean_memory[1], 'VSTE
XT': mean_memory[2],
                    'RSIZE': mean_memory[3], 'VGROW': mean_memory[4], 'RGROW': mean_memor
y[5]})

# check null / missing value once again after transformation
df_memory.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df_memory
.columns]).show()
```

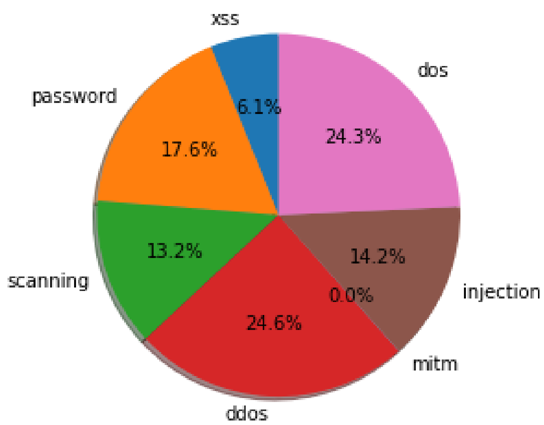
```
+---+---+------+------+------+-----+-----+-----+-----+---+---+------+----+
| ts|PID|MINFLT|MAJFLT|VSTEXT|VSIZE|RSIZE|VGROW|RGROW|MEM|CMD|attack|type|
+---+---+------+------+------+-----+-----+-----+-----+---+---+------+----+
|  0|  0|     0|     0|     0|    0|    0|    0|    0|  0|  0|     0|   0|
+---+---+------+------+------+-----+-----+-----+-----+---+---+------+----+
```

In [6]:

```
## --- 1.3.1 --- ##
print('Count Of Memory Attacks')
df_memory.groupby('attack').count().show()
print('Attack Rate: 11.5%')
print('')

print('Count Of Process Attacks')
df_process.groupby('attack').count().show()
print('Attack Rate: 17.8%')
print('')

print('Count Of Each Kind Of Attacks In Process Activity')
df_process.groupby('type').count().show()
print('Proportion of each kind of attack')
print('Type <xss> ≈ 6%')
print('Type <password> ≈ 17%')
print('Type <scanning> ≈ 13%')
print('Type <ddos> ≈ 24%')
print('Type <mitm> ≈ 0.03%')
print('Type <injection> ≈ 14%')
print('Type <dos> ≈ 24%')

# visualize the proportion using matplotlib
import matplotlib.pyplot as plt
labels = 'xss', 'password', 'scanning', 'ddos', 'mitm', 'injection', 'dos'
sizes = [17759, 51409, 38449, 71603, 112, 41311, 70721]
fig1, ax1 = plt.subplots()
ax1.pie(sizes, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
# Equal aspect ratio ensures that pie is drawn as a circle.
ax1.axis('equal')
plt.show()

# observation
print('')
print('#--- observation ---#')
print('1. The attack rate of process activities is 6.3% higher than the attack rate of me
mory activities')
print('2. In terms of the attacks of process activities...(There is class imbalance)')
print('   The <ddos> and <dos> are the most common attacks that both occupy 24% of all of
the attacks')
print('   As the smallest type of attack, the <mitm> accounts for only 0.03% of all of th
e attacks')
```

```
Count Of Memory Attacks
+------+-------+
|attack|  count|
+------+-------+
|   1.0| 205623|
|   0.0|1794377|
+------+-------+
```

```
Attack Rate: 11.5%

Count Of Process Attacks
+------+-------+
|attack|  count|
+------+-------+
|   1.0| 291364|
|   0.0|1636604|
+------+-------+

Attack Rate: 17.8%

Count Of Each Kind Of Attacks In Process Activity
+---------+-------+
|     type|  count|
+---------+-------+
|      xss|  17759|
| password|  51409|
| scanning|  38449|
|     ddos|  71603|
|   normal|1636604|
|     mitm|    112|
|injection|  41311|
|      dos|  70721|
+---------+-------+

Proportion of each kind of attack
Type <xss> ≈ 6%
Type <password> ≈ 17%
Type <scanning> ≈ 13%
Type <ddos> ≈ 24%
Type <mitm> ≈ 0.03%
Type <injection> ≈ 14%
Type <dos> ≈ 24%
```




```
#--- observation ---#
1. The attack rate of process activities is 6.3% higher than the attack rate of memory ac
tivities
2. In terms of the attacks of process activities...(There is class imbalance)
   The <ddos> and <dos> are the most common attacks that both occupy 24% of all of the at
tacks
   As the smallest type of attack, the <mitm> accounts for only 0.03% of all of the attac
ks
```

In [7]:

```
## --- 1.3.2 --- ##
# numeric features in memory activities
df_memory_numeric = df_memory.select('ts', 'PID', 'MINFLT', 'MAJFLT', 'VSTEXT', 'VSIZE',
                                     'RSIZE', 'VGROW', 'RGROW', 'MEM')
# display the basic statistics
df_memory_numeric.describe().toPandas().head()
```

Out[7]:

summary	ts	PID	MINFLT	MAJFLT	VSTEXT

	summary	ts	PID	MINFLT	MAJFLT	VSTEXT	
0	count	2000000	2000000	2000000	2000000	2000000	
1	mean	1.55624581707872E9	4999.360446	404.5137191086731	1108.8663392387334	2813.190188891452	4496.419
2	stddev	984463.3693605846	4887.313351921498	17143.991131743223	5175.760836661632	8174.246110893945	9046.338
3	min	1.55421683E9	1007.0	0.0	0.0	0.0	
4	max	1.55835571E9	53096.0	8050000.0	107776.0	99992.0	

In [8]:

```
## --- 1.3.2 --- ##
# non-numeric features in memory activities
df_memory_non_numeric = df_memory.select('CMD')
# display the top-10 values and the corresponding counts
df_memory_non_numeric.groupby('CMD').count().orderBy(col('count').desc()).show(10)
```

```
+----------+------+
|       CMD| count|
+----------+------+
|      atop|325985|
|   apache2| 89761|
| jfsCommit| 81714|
|  vmtoolsd| 77871|
|      Xorg| 49981|
|  nautilus| 48356|
|irqbalance| 44387|
|    compiz| 44356|
|  ostinato| 43024|
|     drone| 41392|
+----------+------+
only showing top 10 rows
```

In [9]:

```
## --- 1.3.2 --- ##
# numeric features in process activities
df_process_numeric = df_process.select('ts', 'PID', 'TRUN', 'TSLPI', 'TSLPU', 'NICE',
                                       'PRI', 'RTPR', 'CPUNR', 'EXC', 'CPU')
# display the basic statistics
df_process_numeric.describe().toPandas().head()
```

Out[9]:

	summary	ts	PID	TRUN	TSLPI	TSLPU	
0	count	1927968	1927968	1927968	1927968	1927968	
1	mean	1.5563198311846504E9	5068.709770597852	0.0632287465352122	3.508334163222626	3.6100184235422994E-4	4.63
2	stddev	771350.0251249488	4987.784329320458	0.24782587090415928	6.988459728531726	0.04421874419214571	8.4
3	min	1.55421683E9	1007.0	0.0	0.0	0.0	
4	max	1.55759296E9	53080.0	12.0	70.0	21.0	

In [10]:

```
## --- 1.3.2 --- ##
# non-numeric features in process activities
df_process_non_numeric = df_process.select('POLI', 'Status', 'State', 'CMD')
# display the top-10 values and the corresponding counts
df_process_non_numeric.select('POLI').groupby('POLI').count().orderBy(col('count').desc(
)).show(10)
df_process_non_numeric.select('Status').groupby('Status').count().orderBy(col('count').d
esc()).show(10)
df_process_non_numeric.select('State').groupby('State').count().orderBy(col('count').des
c()).show(10)
df_process_non_numeric.select('CMD').groupby('CMD').count().orderBy(col('count').desc())
```

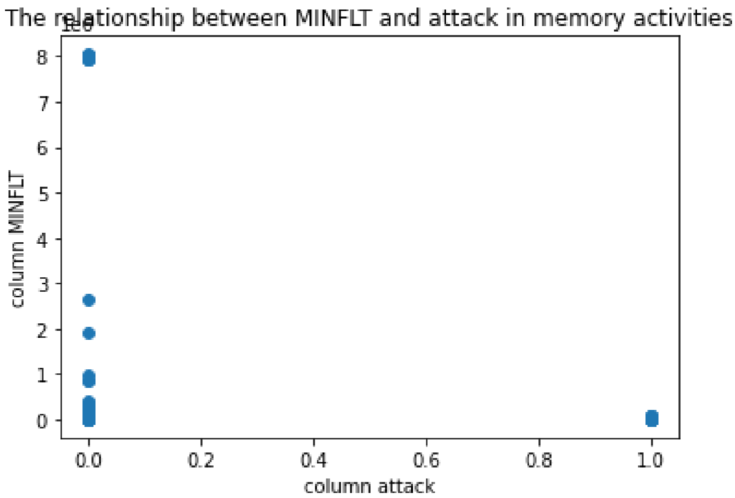
```
.show(10)
```

```
+----+-------+
|POLI|  count|
+----+-------+
|norm|1861558|
|   0|  53216|
|   -|  13194|
+----+-------+

+------+-------+
|Status|  count|
+------+-------+
|     -|1416322|
|     0| 438984|
|    NE|  48602|
|     N|  23313|
|    NS|    743|
|     C|      3|
|    NC|      1|
+------+-------+

+-----+-------+
|State|  count|
+-----+-------+
|    S|1676350|
|    I|  98986|
|    R|  84753|
|    E|  66410|
|    Z|   1118|
|    D|    344|
|    T|      7|
+-----+-------+

+--------------+------+
|           CMD| count|
+--------------+------+
|          atop|441180|
|       apache2|313143|
|      vmtoolsd|112029|
|          Xorg| 66813|
|      nautilus| 63449|
|gnome-terminal| 47628|
|        compiz| 44386|
|    irqbalance| 44324|
|      ostinato| 42979|
|         drone| 41390|
+--------------+------+
only showing top 10 rows
```

In [18]:

```
## --- 1.3.3 - Memory Activity Plot 1  --- ##
memory_plot_1 = df_memory.select('MINFLT', 'attack').take(2000000)

x_attack = []
y_minflt = []

# extract values to a list
for row in memory_plot_1:
    y_minflt.append(row[0])
    x_attack.append(row[1])

# scatter all of the records
plt.scatter(x_attack, y_minflt)
plt.xlabel('column attack')
plt.ylabel('column MINFLT')
plt.title('The relationship between MINFLT and attack in memory activities')
plt.show()

print('The description of the plot: I scatter all of the records to form a plot to examin
```

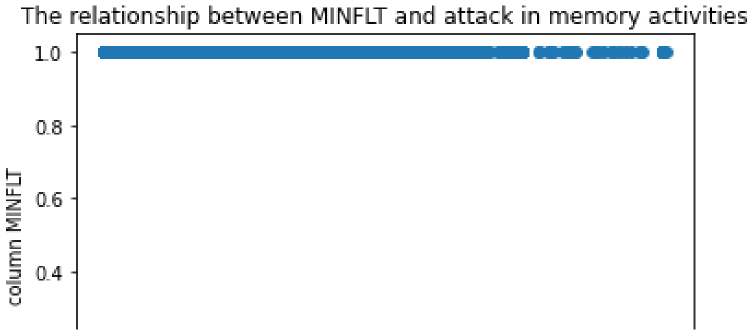
```
e the')
print('                              relationship between column MINFLT and column attack
')
print('The finding: According to the scatter chart, the values of MINFLT are between 0 an
d 8,')
print('             one interesting observation is that the value of MINFLT is really lo
w when there is an attack.')
print('             As indicated in the scatter chart, the values of MINFLT are around 0
- 0.5 when attacked',
                    end='\n\n')
print('Important Note: The values of MINFLT here are simplified values to make the plot m
ore clear, \
they are not real values')
```

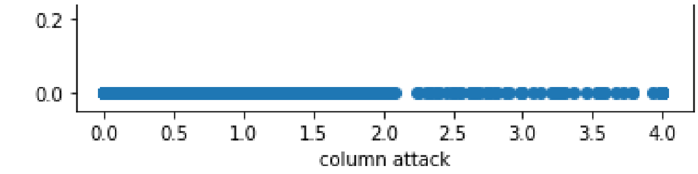



```
The description of the plot: I scatter all of the records to form a plot to examine the
                              relationship between column MINFLT and column attack
The finding: According to the scatter chart, the values of MINFLT are between 0 and 8,
             one interesting observation is that the value of MINFLT is really low when t
here is an attack.
             As indicated in the scatter chart, the values of MINFLT are around 0 - 0.5 w
hen attacked

Important Note: The values of MINFLT here are simplified values to make the plot more cle
ar, they are not real values
```

In [33]:

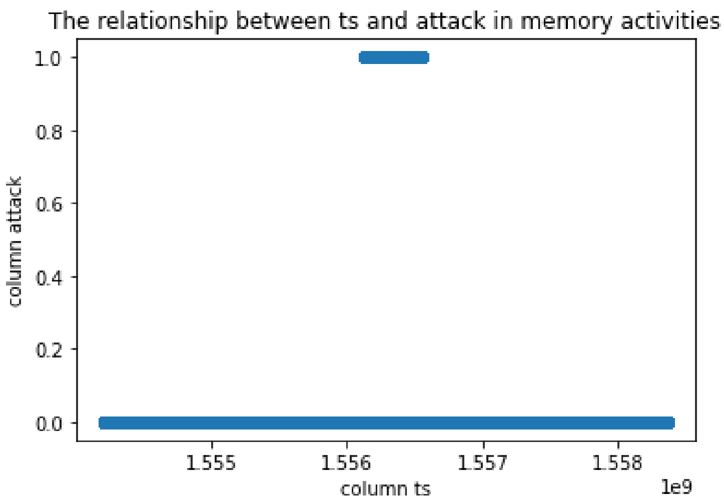
```
A = df_process.select('CPU', 'attack').take(1927968)

x_attack = []
y_minflt = []

# extract values to a list
for row in A:
    y_minflt.append(row[0])
    x_attack.append(row[1])

# scatter all of the records
plt.scatter(y_minflt, x_attack)
plt.xlabel('column attack')
plt.ylabel('column MINFLT')
plt.title('The relationship between MINFLT and attack in memory activities')
plt.show()
```





In [12]:

```
## --- 1.3.3 - Memory Activity Plot 2  --- ##
memory_plot_2 = df_memory.select('ts', 'attack').take(2000000)

x_ts = []
y_attack = []

# extract values to a list
for row in memory_plot_2:
    x_ts.append(row[0])
    y_attack.append(row[1])

# use all the records to plot a line chart
plt.scatter(x_ts, y_attack)
plt.xlabel('column ts')
plt.ylabel('column attack')
plt.title('The relationship between ts and attack in memory activities')
plt.show()

print('The description of the plot: I used all of the records to scatter a chart to exami
ne the')
print('                             relationship between column ts and column attack')

print('The finding: The ts started from 1554xxxxxx to 1558xxxxxx,')
print('             as can be seen in the scatter chart,')
print('             it is also obvious that all of the memory attacks are between 1556 -
1557')
print('             which is approximately between 04/23/2019 @ 6:13am (UTC) and 05/04/2
019 @ 8:00pm (UTC)')
```

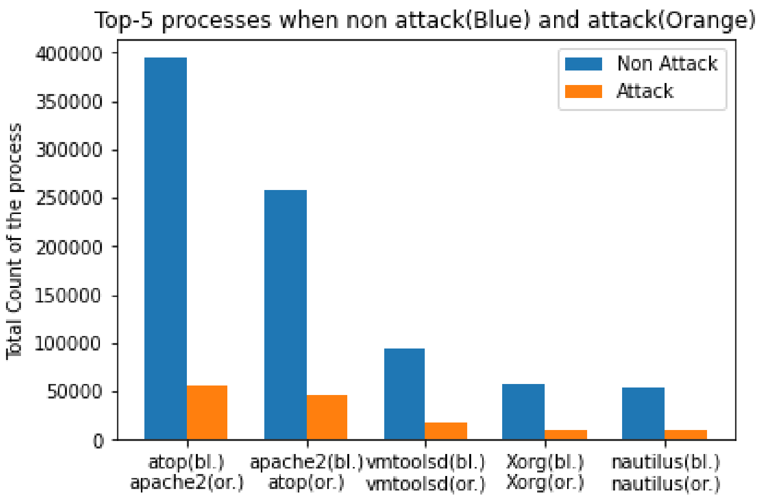



```
The description of the plot: I used all of the records to scatter a chart to examine the
                             relationship between column ts and column attack
The finding: The ts started from 1554xxxxxx to 1558xxxxxx,
             as can be seen in the scatter chart,
             it is also obvious that all of the memory attacks are between 1556 - 1557
             which is approximately between 04/23/2019 @ 6:13am (UTC) and 05/04/2019 @ 8:
00pm (UTC)
```

In [13]:

```
## --- 1.3.3 - Process Activity Plot 1  --- ##
import numpy as np
non_attack_top10_cmd = df_process.filter(col('attack')==0).select('CMD')\
                                  .groupby('CMD').count().orderBy(col('count').desc()).ta
ke(5)
```

```
attack_cmd = df_process.filter(col('attack')==1).select('CMD')\
                               .groupby('CMD').count().orderBy(col('count').desc()).take(5
)

x_process_name = [[], []]
y_total_count = [[], []]
labels = []

# extract values to a list
for row in non_attack_top10_cmd:
    x_process_name[0].append(row[0])
    y_total_count[0].append(row[1])

for row in attack_cmd:
    x_process_name[1].append(row[0])
    y_total_count[1].append(row[1])

# produce the labels for axis x of the bar chart
index = 0
for name in x_process_name[0]:
    name += '(bl.)' + '\n' + str(x_process_name[1][index]) + '(or.)'
    labels.append(name)
    index += 1

# the width of the bars
width = 0.35
# draw bar chart
x = np.arange(5)
fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, y_total_count[0], width, label='Non Attack')
rects2 = ax.bar(x + width/2, y_total_count[1], width, label='Attack')
ax.set_ylabel('Total Count of the process')
ax.set_title('Top-5 processes when non attack(Blue) and attack(Orange)')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend()
plt.show()

print('The description of the plot: I selected the top 5 total count from the column <CMD
> in two different scenarios,')
print('                              attack=0 and attack=1.  The totlal count of attack=1
are the orange bars in the')
print('                              chart while the total count of attack=0 are the blue
bars in the chart.')
print('The finding: According to the bar chart, the top-5 values when there is an attack
are almost the same as')
print('               there is no attack.  In other words, it is acceptable to say that the
most frequently used')
print('               processes may have higher count of attacks comparing with those rare
ly-used processes.')
```

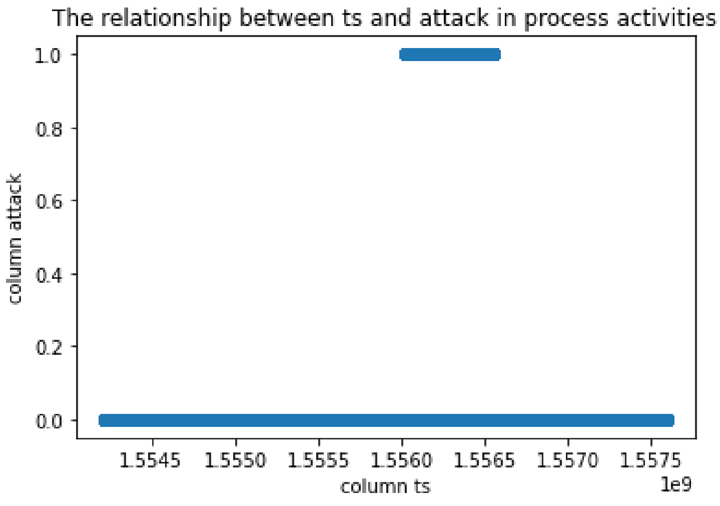



```
The description of the plot: I selected the top 5 total count from the column <CMD> in tw
o different scenarios,
                              attack=0 and attack=1.  The totlal count of attack=1 are th
e orange bars in the
```

chart while the total count of attack=0 are the blue bars i
n the chart.
The finding: According to the bar chart, the top-5 values when there is an attack are alm
ost the same as
            there is no attack. In other words, it is acceptable to say that the most f
requently used
            processes may have higher count of attacks comparing with those rarely-used
processes.

In [14]:

```
## --- 1.3.3 - Process Activity Plot 2  --- ##
process_plot_2 = df_process.select('ts', 'attack').take(1927968)

x2_attack = []
y2_nice = []

# extract values to a list
for row in process_plot_2:
    x2_attack.append(row[0])
    y2_nice.append(row[1])

# use all the records to plot a line chart
plt.scatter(x2_attack, y2_nice)
plt.xlabel('column ts')
plt.ylabel('column attack')
plt.title('The relationship between ts and attack in process activities')
plt.show()

print('The description of the plot: I used all of the records to scatter a chart to exami
ne the')
print('                              relationship between column ts and column attack')

print('The finding: The ts started from 1554xxxxxx to 15575xxxxx,')
print('             as can be seen in the scatter chart,')
print('             it is also obvious that all of the memory attacks are between 15560xx
xxx and 15565xxxxx')
print('             which is approximately between 04/23/2019 @ 6:13am (UTC) and 04/29/2
019 @ 1:06am (UTC)')
```



The description of the plot: I used all of the records to scatter a chart to examine the
                              relationship between column ts and column attack
The finding: The ts started from 1554xxxxxx to 15575xxxxx,
             as can be seen in the scatter chart,
             it is also obvious that all of the memory attacks are between 15560xxxxx and
15565xxxxx
             which is approximately between 04/23/2019 @ 6:13am (UTC) and 04/29/2019 @ 1:
06am (UTC)

In [15]:

```
## --- 2.1.1 --- ##
# split each dataset into 80% training and 20% testing
train_memory, test_memory = df_memory.randomSplit([0.8, 0.2], seed=2018)
```

```
train_process, test_process = df_process.randomSplit([0.8, 0.2], seed=2018)
```

In [16]:

```
## --- 2.1.2 --- ##

# --- rebalance training data for memory activities --- #

# extract 20% of attack records and all of the non-attack records
df_major_memory = train_memory.filter(col("attack") == 0)
df_minor_memory = train_memory.filter(col("attack") == 1).sample(1/5, seed=2020)

print('--- rebalance training data for memory activities ---')
print('')
#check the data ratio in the column 'attack'
memory_ratio = int(df_minor_memory.count()) / int(df_major_memory.count())
print('attack/non-attack = {}'.format(memory_ratio))

# undersampling the non-attack records to match the proper ratio
sampled_major_memory = df_major_memory.sample(1/22.1094, seed=2020)

# combine the sampled major records with the minor records
df_sampled_memory = sampled_major_memory.unionAll(df_minor_memory)

# check the new data ratio
new_df_major_memory = df_sampled_memory.filter(col("attack") == 0)
new_df_minor_memory = df_sampled_memory.filter(col("attack") == 1)
new_memory_ratio = int(new_df_minor_memory.count()) / int(new_df_major_memory.count())
print('After undersampling ...')
print('attack/non-attack = {}'.format(new_memory_ratio))

# cache the rebalanced data
df_sampled_memory.cache()

# display the number for each event
print('')
memory_attack_events = df_sampled_memory.filter(col("attack") == 1).count()
memory_non_attack_events = df_sampled_memory.filter(col("attack") == 0).count()
print('number of attack events in memory activities: {}'.format(memory_attack_events))
print('number of non-attack events in memory activities: {}'.format(memory_non_attack_ev
ents))
```

```
--- rebalance training data for memory activities ---

attack/non-attack = 0.022750351371567945
After undersampling ...
attack/non-attack = 0.5

number of attack events in memory activities: 32665
number of non-attack events in memory activities: 65330
```

In [17]:

```
## --- 2.1.2 --- ##

# --- rebalance training data for process activities --- #

# extract 20% of attack records and all of the non-attack records
df_major_process = train_process.filter(col("attack") == 0)
df_minor_process = train_process.filter(col("attack") == 1).sample(1/5, seed=2020)

print('--- rebalance training data for process activities ---')
print('')

# check the data ratio in the column 'attack'
process_ratio = int(df_minor_process.count()) / int(df_major_process.count())
print('attack/non-attack = {}'.format(process_ratio))

# undersampling the non-attack records to match the proper ratio
sampled_major_process = df_major_process.sample(1/14.18716, seed=2020)
```

```
# combine the sampled major records with the minor records
df_sampled_process = sampled_major_process.unionAll(df_minor_process)

# check the new data ratio
new_df_major_process = df_sampled_process.filter(col("attack") == 0)
new_df_minor_process = df_sampled_process.filter(col("attack") == 1)
new_process_ratio = int(new_df_minor_process.count()) / int(new_df_major_process.count())
print('After undersampling ...')
print('attack/non-attack = {}'.format(new_process_ratio))

# cache the rebalanced data
df_sampled_process.cache()

# display the number for each event
print('')
process_attack_events = df_sampled_process.filter(col("attack") == 1).count()
process_non_attack_events = df_sampled_process.filter(col("attack") == 0).count()
print('number of attack events in process activities: {}'.format(process_attack_events))
print('number of non-attack events in process activities: {}'.format(process_non_attack_e
vents))
```

```
--- rebalance training data for process activities ---

attack/non-attack = 0.03540664960855685
After undersampling ...
attack/non-attack = 0.5

number of attack events in process activities: 46370
number of non-attack events in process activities: 92740
```

In [133]:

```
## --- 2.2.1 --- ##
print('####################################################################################
############################')
print('For the Memory Activities, I would choose <MINFLT>, <MAJFLT>, <RGROW>, <VGROW>, <V
STEXT>, <PID>, <CMD>.')
print('')

print('The relationship between <MINFLT> and <attack> is shown in ## --- 1.3.3 - Memory A
ctivity Plot 1  --- ##')
print('According to the chart, the values of <MINFLT> are really low when there is an att
ack')
print('while the values of <MINFLT> are discrete when there is no attack.')
print('Since the values of <MINFLT> are different when the value of <attack> changes,')
print('it would be a good idea to add <MINFLT> to the feature columns.')
print('')

print('For the columns <MAJFLT> and <RGROW>, the reason why I choose them is similar to c
hoosing <MINFLT>.')
print('When an attack happens, the values of <MAJFLT> and <RGROW> tend to be more concent
rated than no attack happens.')
print('Therefore, I assume that they may contain valuable information for the prediction.
')
print('')

print('The columns <VGROW>, <VSTEXT> are not like <MAJFLT> and <RGROW>,')
print('which means the data distribution is more similar between the attack and non-attac
k scenarios.')
print('However, the data distribution of those three columns are still slightly different
')
print('between attack and non-attack, so I believe that it is worth to first add those th
ree columns into')
print('the features, then I can check if my assumption is correct when examining the feat
ure importance later.')
print('')

print('For the column <PID>, I chose this one based on my assumption that there might be
some processes')
print('which could be attacked more frequently than other processes.  It is like loophole
s within the system.')
print('')
```

```
print('<CMD> is a non-numeric column which contains the name of the process.')
print('The data in <CMD> has data skewness, some processes are used frequently')
print('while other processes are rarely executed.')
print('After I explored the data deeper, I found out that the most frequently used processes have higher')
print('attack count than those are rarely used, therefore, adding <CMD> into the feature columns would')
print('be a good idea because we may predict the attack through the frequency of the processes.')
print('')

print('Strategy of implementation >>> Divide the numeric and non-numeric column')
print('')

print('Implementation for the non-numeric columns - <CMD>')
print('StringIndexer -> OneHotEncoding -> Vector Assembler -> ML Algorithm')
print('')

print('Implementation for the numeric columns - <MINFLT>, <MAJFLT>, <RGROW>, <VGROW>, <MEM>, <VSTEXT>')
print('                                   Vector Assembler -> ML Algorithm')

print('#############################################################################################################')
print('')

print('For the Process Activities, I would choose <CMD>, <State>, <Status>, <PID>, <TRUN>, <TSLPI>, <TSLPU>')
print('')

print('For the columns <CMD>, <State>, and <Status>,')
print('All of the three columns have data skewness, and the most frequent values in <CMD>, <State>, and <Status>')
print('tend to have higher count of attack.  So I would add those three columns into the feature columns since')
print('they may contain valuable information for prediction.')
print('')

print('As for the column <PID>, I chose this one based on my assumption that there might be some processes')
print('which could be attacked more frequently than other processes.  It is like loopholes within the system.')
print('')

print('For the column <TRUN>, <TSLPI>, and <TSLPU>, I found out that the data distrbution patterns would change')
print('between the two scenarios, attack and non-attack.  Therefore, I assumed that those columns might contain')
print('valuable information for the prediction.')
print('')

print('Strategy of implementation >>> Divide the numeric and non-numeric column')
print('')

print('Implementation for the non-numeric columns - <CMD>, <State>, <Status>')
print('StringIndexer -> OneHotEncoding -> Vector Assembler -> ML Algorithm')
print('')
print('Implementation for the numeric columns - <PID>, <TRUN>, <TSLPI>, <TSLPU>')
print('                                   Vector Assembler -> ML Algorithm')

print('#############################################################################################################')
```

```
########################################################################################
#####################
For the Memory Activities, I would choose <MINFLT>, <MAJFLT>, <RGROW>, <VGROW>, <VSTEXT>,
<PID>, <CMD>.

The relationship between <MINFLT> and <attack> is shown in ## --- 1.3.3 - Memory Activity
Plot 1  --- ##
According to the chart, the values of <MINFLT> are really low when there is an attack
```

while the values of <MINFLT> are discrete when there is no attack.
Since the values of <MINFLT> are different when the value of <attack> changes,
it would be a good idea to add <MINFLT> to the feature columns.

For the columns <MAJFLT> and <RGROW>, the reason why I choose them is similar to choosing <MINFLT>.
When an attack happens, the values of <MAJFLT> and <RGROW> tend to be more concentrated t
han no attack happens.
Therefore, I assume that they may contain valuable information for the prediction.

The columns <VGROW>, <VSTEXT> are not like <MAJFLT> and <RGROW>,
which means the data distribution is more similar between the attack and non-attack scena
rios.
However, the data distribution of those three columns are still slightly different
between attack and non-attack, so I believe that it is worth to first add those three col
umns into
the features, then I can check if my assumption is correct when examining the feature imp
ortance later.

For the column <PID>, I chose this one based on my assumption that there might be some pr
ocesses
which could be attacked more frequently than other processes. It is like loopholes withi
n the system.

<CMD> is a non-numeric column which contains the name of the process.
The data in <CMD> has data skewness, some processes are used frequently
while other processes are rarely executed.
After I explored the data deeper, I found out that the most frequently used processes hav
e higher
attack count than those are rarely used, therefore, adding <CMD> into the feature columns
would
be a good idea because we may predict the attack through the frequency of the processes.

Strategy of implementation >>> Divide the numeric and non-numeric column

Implementation for the non-numeric columns - <CMD>
StringIndexer -> OneHotEncoding -> Vector Assembler -> ML Algorithm

Implementation for the numeric columns - <MINFLT>, <MAJFLT>, <RGROW>, <VGROW>, <MEM>, <VS
TEXT>
Vector Assembler -> ML Algorithm
###
#####################

For the Process Activities, I would choose <CMD>, <State>, <Status>, <PID>, <TRUN>, <TSLP
I>, <TSLPU>

For the columns <CMD>, <State>, and <Status>,
All of the three columns have data skewness, and the most frequent values in <CMD>, <Stat
e>, and <Status>
tend to have higher count of attack. So I would add those three columns into the feature
columns since
they may contain valuable information for prediction.

As for the column <PID>, I chose this one based on my assumption that there might be some
processes
which could be attacked more frequently than other processes. It is like loopholes withi
n the system.

For the column <TRUN>, <TSLPI>, and <TSLPU>, I found out that the data distrbution patter
ns would change
between the two scenarios, attack and non-attack. Therefore, I assumed that those column
s might contain
valuable information for the prediction.

Strategy of implementation >>> Divide the numeric and non-numeric column

Implementation for the non-numeric columns - <CMD>, <State>, <Status>
StringIndexer -> OneHotEncoding -> Vector Assembler -> ML Algorithm

Implementation for the numeric columns - <PID>, <TRUN>, <TSLPI>, <TSLPU>
Vector Assembler -> ML Algorithm

```
####################################################################################################
####################
```

In [134]:

```
## --- 2.2.2 --- ##
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoder
from pyspark.ml.feature import VectorAssembler

## --- Memory activities --- ##

# define categorical columns and implement the StringIndexer
inputCols = ['CMD']
outputCols = ['CMD_index']
memory_stage_1 = StringIndexer(inputCols=inputCols, outputCols=outputCols).setHandleInval
id("keep")

# implement the OneHotEncoder
inputCols_OHE = [x for x in outputCols]
outputCols_OHE = [f'{x}_vec' for x in inputCols]
memory_stage_2 = OneHotEncoder(inputCols=inputCols_OHE,
                               outputCols=outputCols_OHE)

# define the numeric columns and integrate with the output columns from OneHotEncoder
numeric_cols = ['MINFLT', 'MAJFLT', 'RGROW', 'VGROW', 'VSTEXT', 'PID']
assemblerInputs  = outputCols_OHE + numeric_cols
memory_stage_3 = VectorAssembler(inputCols=assemblerInputs,
                                 outputCol="features").setHandleInvalid("keep")

## --- Process activities --- ##

# define categorical columns and implement the StringIndexer
process_inputCols = ['CMD', 'State', 'Status']
process_outputCols = ['CMD_index', 'State_index', 'Status_index']
process_stage_1 = StringIndexer(inputCols=process_inputCols, outputCols=process_outputCol
s).setHandleInvalid("keep")

# implement the OneHotEncoder
process_inputCols_OHE = [x for x in process_outputCols]
process_outputCols_OHE = [f'{x}_vec' for x in process_inputCols]
process_stage_2 = OneHotEncoder(inputCols=process_inputCols_OHE,
                                outputCols=process_outputCols_OHE)

# define the numeric columns and integrate with the output columns from OneHotEncoder
process_numeric_cols = ['PID', 'TRUN', 'TSLPI', 'TSLPU']
process_assemblerInputs  = process_outputCols_OHE + process_numeric_cols
process_stage_3 = VectorAssembler(inputCols=process_assemblerInputs,
                                  outputCol="features").setHandleInvalid("keep")
```

In [22]:

```
## ---   Bonus Work For Process Activities    --- ##
## --- A custom transformer for column "POLI" --- ##

from pyspark import keyword_only
from pyspark.ml.param.shared import HasInputCol, HasOutputCol, Param
from pyspark.ml.util import DefaultParamsReadable, DefaultParamsWritable
from pyspark.sql.functions import udf
from pyspark.ml import Transformer
from pyspark.sql.types import IntegerType

class POLITransformer(Transformer, HasInputCol, HasOutputCol,DefaultParamsReadable, Defa
ultParamsWritable):
    @keyword_only
    def __init__(self, inputCol=None, outputCol=None):
        super(POLITransformer, self).__init__()
        kwargs = self._input_kwargs
        self.setParams(**kwargs)
```

```
    @keyword_only
    def setParams(self, inputCol=None, outputCol=None):
        kwargs = self._input_kwargs
        return self._set(**kwargs)

    def setInputCol(self, value):
        return self._set(inputCol=value)

    def setOutputCol(self, value):
        return self._set(outputCol=value)

    def _transform(self, dataset):
        keys = ["norm", "btch", "idle", "fifo", "rr", "0", "-"]
        index = range(0,7)
        poli_dict = {k:v for (k,v) in zip(keys, index)}

        @udf(IntegerType())
        def translate_poli(s):
            return poli_dict[s]

        out_col = self.getOutputCol()
        in_col = dataset[self.getInputCol()]
        return dataset.withColumn(out_col, translate_poli(in_col))

df_test = df_process
ct = POLITransformer(inputCol='POLI', outputCol='POLI_indexed')
ct.transform(df_test).groupby('POLI_indexed').count().show()
```

```
+------------+-------+
|POLI_indexed|  count|
+------------+-------+
|           6|  13194|
|           5|  53216|
|           0|1861558|
+------------+-------+
```

In [136]:

```
## --- 2.2.3 --- ##
from pyspark.ml.classification import DecisionTreeClassifier, GBTClassifier
dt = DecisionTreeClassifier(featuresCol = 'features', labelCol = 'label', maxDepth = 3)
memory_dt_pipeline = Pipeline(stages=[memory_stage_1, memory_stage_2, memory_stage_3, dt]
)
process_dt_pipeline = Pipeline(stages=[process_stage_1, process_stage_2, process_stage_3,
dt])

from pyspark.ml.regression import GBTRegressor
gbt = GBTClassifier(labelCol="label", featuresCol="features", maxIter=10)
memory_gbt_pipeline = Pipeline(stages=[memory_stage_1, memory_stage_2, memory_stage_3, gb
t])
process_gbt_pipeline = Pipeline(stages=[process_stage_1, process_stage_2, process_stage_3
, gbt])
```

In [137]:

```
## --- 2.3.1 --- ##
df_sampled_memory = df_sampled_memory.withColumnRenamed('attack', 'label')
df_sampled_process = df_sampled_process.withColumnRenamed('attack', 'label')

# dt model for memory activities
memory_dt_model = memory_dt_pipeline.fit(df_sampled_memory)

# dt model for process activities
process_dt_model = process_dt_pipeline.fit(df_sampled_process)

# gbt model for memory activities
memory_gbt_model = memory_gbt_pipeline.fit(df_sampled_memory)

# gbt model for process activities
process_gbt_model = process_gbt_pipeline.fit(df_sampled_process)
```

In [141]:

```
## --- 2.3.2 --- ##
test_memory = test_memory.withColumnRenamed('attack', 'label')
test_process = test_process.withColumnRenamed('attack', 'label')

# dt predictions for memory activities
print('dt predictions for memory activities')
dt_memory_attack_prediction = memory_dt_model.transform(test_memory)
dt_memory_attack_prediction.select('label', 'prediction').groupby('label', 'prediction')
.count().show()
print('')

# dt predictions for process activities
print('dt predictions for process activities')
dt_process_attack_prediction = process_dt_model.transform(test_process)
dt_process_attack_prediction.select('label', 'prediction').groupby('label', 'prediction'
).count().show()
print('')

# gbt predictions for memory activities
print('gbt predictions for memory activities')
gbt_memory_attack_prediction = memory_gbt_model.transform(test_memory)
gbt_memory_attack_prediction.select('label', 'prediction').groupby('label', 'prediction'
).count().show()
print('')

# gbt predictions for process activities
print('gbt predictions for process activities')
gbt_process_attack_prediction = process_gbt_model.transform(test_process)
gbt_process_attack_prediction.select('label', 'prediction').groupby('label', 'prediction
').count().show()
```

```
dt predictions for memory activities
+-----+----------+------+
|label|prediction| count|
+-----+----------+------+
|  1.0|       0.0| 31121|
|  0.0|       0.0|337015|
|  1.0|       1.0| 10253|
|  0.0|       1.0| 21560|
+-----+----------+------+


dt predictions for process activities
+-----+----------+------+
|label|prediction| count|
+-----+----------+------+
|  1.0|       0.0| 35688|
|  0.0|       0.0|291705|
|  1.0|       1.0| 23089|
|  0.0|       1.0| 35258|
+-----+----------+------+


gbt predictions for memory activities
+-----+----------+------+
|label|prediction| count|
+-----+----------+------+
|  1.0|       0.0| 27379|
|  0.0|       0.0|333866|
|  1.0|       1.0| 13995|
|  0.0|       1.0| 24709|
+-----+----------+------+


gbt predictions for process activities
+-----+----------+------+
|label|prediction| count|
+-----+----------+------+
|  1.0|       0.0| 30885|
|  0.0|       0.0|281884|
```

```
|  1.0|       1.0| 27892|
|  0.0|       1.0| 45079|
+-----+----------+------+
```

In [139]:

```
## --- 2.3.3 --- ##
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.mllib.evaluation import MulticlassMetrics

def metrics(prediction):
    # AUC
    evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction", labelCol=
'label')
    auc = evaluator.evaluate(prediction)

    # calculate metrics (precision, and recall) using RDD
    predictionRDD = prediction.select(['label', 'prediction']) \
                            .rdd.map(lambda line: (line[1], line[0]))
    metrics = MulticlassMetrics(predictionRDD)

    # statistics - Precision, Recall, and Accuracy
    precision_attack = metrics.precision(1)
    recall_attack = metrics.recall(1)
    accuracy_attack = metrics.accuracy

    return evaluator.getMetricName() + ': ' + str(auc) + '\n'\
            + 'Accuracy: ' + str(accuracy_attack) + '\n'\
            + 'Precision: ' + str(precision_attack) + '\n'\
            + 'Recall: ' + str(recall_attack)

# --- Calculate the AUC, accuracy, precision, and recall for DT and GBT predictions in ea
ch activity --- #

## AUC, accuracy, precision, and recall for DT in memory activities ##
print('---Decision Tree For Memory Activities---')
print(metrics(dt_memory_attack_prediction))
print('')

## AUC, accuracy, precision, and recall for DT in process activities ##
print('---Decision Tree For Process Activities---')
print(metrics(dt_process_attack_prediction))
print('')

## AUC, accuracy, precision, and recall for GBT in memory activities ##
print('---Gradient Boosted Tree For Memory Activities---')
print(metrics(gbt_memory_attack_prediction))
print('')

## AUC, accuracy, precision, and recall for GBT in process activities ##
print('---Gradient Boosted Tree For Process Activities---')
print(metrics(gbt_process_attack_prediction))
print('')

print('---Discuss which metric is more proper for measuring the model performance on iden
tifying attacks---')
print('')
print('Recall is the most proper metric to measure the model performance on identifying a
ttacks.')
print('The most important goal is to predict an attack when there is one, and the Recall
metric reflects')
print('the proportion of positive cases correctly judged to the total positive cases.  Th
erefore,')
print('I believe that the Recall is the metric that we should care about the most in this
case.')
```

```
---Decision Tree For Memory Activities---
areaUnderROC: 0.5190213331647937
Accuracy: 0.8682807057899882
Precision: 0.3222896300254613
Recall: 0.24781263595494754
```

```
---Decision Tree For Process Activities---
areaUnderROC: 0.3983387819512974
Accuracy: 0.8160781873801006
Precision: 0.3957187173290829
Recall: 0.3928237235653402

---Gradient Boosted Tree For Memory Activities---
areaUnderROC: 0.8125196515316262
Accuracy: 0.8697633948328412
Precision: 0.36159053327821417
Recall: 0.3382559095083869

---Gradient Boosted Tree For Process Activities---
areaUnderROC: 0.8007994923899779
Accuracy: 0.8030694250012962
Precision: 0.38223403817955076
Recall: 0.4745393606342617

---Discuss which metric is more proper for measuring the model performance on identifying
attacks---

Recall is the most proper metric to measure the model performance on identifying attacks.
The most important goal is to predict an attack when there is one, and the Recall metric
reflects
the proportion of positive cases correctly judged to the total positive cases.  Therefore
,
I believe that the Recall is the metric that we should care about the most in this case.
```

In [146]:

```
## --- 2.3.4 --- ##
import pandas as pd

def ExtractFeatureImp(featureImp, dataset, featuresCol):
    """
    method that returns the index, name, and score of the features in the dataset
    """
    list_extract = []
    for i in dataset.schema[featuresCol].metadata["ml_attr"]["attrs"]:
        list_extract = list_extract + dataset.schema[featuresCol].metadata["ml_attr"]["a
ttrs"][i]
    varlist = pd.DataFrame(list_extract)
    varlist['score'] = varlist['idx'].apply(lambda x: featureImp[x])
    return(varlist.sort_values('score', ascending = False))

print('Top-5 most important features of DT. in memory activities')
print('')
print(ExtractFeatureImp(memory_dt_model.stages[-1].featureImportances,
                        dt_memory_attack_prediction, "features").head(10))
print('---------------------------------------------------------')

print('Top-5 most important features of DT. in process activities')
print('')
print(ExtractFeatureImp(process_dt_model.stages[-1].featureImportances,
                        dt_process_attack_prediction, "features").head(10))
print('---------------------------------------------------------')

print('Top-5 most important features of GBT. in memory activities')
print('')
print(ExtractFeatureImp(memory_gbt_model.stages[-1].featureImportances,
                        gbt_memory_attack_prediction, "features").head(10))
print('---------------------------------------------------------')

print('Top-5 most important features of GBT. in process activities')
print('')
print(ExtractFeatureImp(process_gbt_model.stages[-1].featureImportances,
                        gbt_process_attack_prediction, "features").head(10))
print('---------------------------------------------------------')
print('')
```

```
print('---Discussion of which models are better---')
print('As can be seen in ## --- 2.3.3 --- ##, we care about the value of recall to better predict the attacks.')
print('The GBT. models for both activities performed better than the DT. models did.')
print('As we look into the values of recall, the DT. models got the values of .247 and .392, on the other hand,')
print('the GBT. models got the values of .338 and .474 which are much higher than DT. models.')
print('Further than recall value, as we look at the values of areaUnderROC which is a performance measurement')
print('for classification problem that can measure the ability of a model at predicting 0s as 0s and 1s as 1s.')
print('The values of areaUnderROC in DT. models are .519 and .398,')
print('on the other side, the GBT. models got .812 and .800 which are much higher than DT. models as well.')
print('According to the chart below, we can say that performance of the DT. models measured by areaUnderROC')
print('are (F) and below (F) while the performance of GBT. models are (B) and (B).')
print('In conclusion, I would choose GBT. models for both activities since the performance of GBT. models measured by')
print('recall and areaUnderROC are much better than the DT. models.')
print('')

print('---Performance measured by areaUnderROC---')
print('.90-1   = excellent (A)')
print('.80-.90 = good (B)')
print('.70-.80 = fair (C)')
print('.60-.70 = poor (D)')
print('.50-.60 = fail (F)')
print('')

print('---Discussion of whether to select <ts> or not---')
print('I will not include the <ts> column in my selected models.  As can be seen in the plot from')
print('## --- 1.3.3 - Memory Activity Plot 2  --- ## and ## --- 1.3.3 - Process Activity Plot 2  --- ##,')
print('All of the attacks for both memory and process activities concentrated in the specific time.')
print('If I include the <ts> into my models, the performance for all the models would be')
print('much better than without <ts>.  However, the feature importance of <ts> column would be')
print('exetremely high(even 1.0), which means the model only use <ts> to predict the attack.')
print('This is not a good thing to my models since we will not know "when" the attacks will happen')
print('if we use the models to predict the cyber attacks in the future.')
print('I am not saying that the columns containing the information like <ts> is useless,')
print('there is still some cases that we can choose columns like <ts> to build our model.')
print('For instance, the <ts> columns indicates the frequency of the cyber attacks such as once a week,')
print('or the <ts> column indicates that the system is more possible to be attacked on Sunday.')
print('Those are actually valuable information for building a prediction model, however in this case,')
print('the <ts> columns from both use case only indicated that all of the cyber attacks focused on')
print('a certain period of time.  And this information is not going to help us predict the coming attacks, therefore,')
print('I will not selecte <ts> column as one of the feature columns for both use case.')
print('')

print('Reference List')
print('http://gim.unmc.edu/dxtests/roc3.htm')
```

```
Top-5 most important features of DT. in memory activities

     idx                 name     score
5    428                  PID  0.403824
7      1        CMD_vec_apache2  0.402521
```

```
0    423                  MINFLT  0.139952
4    427                  VSTEXT  0.053703
282  276      CMD_vec_gvfsd-burn  0.000000
293  287        CMD_vec_<dirname>  0.000000
292  286       CMD_vec_worer/3:1  0.000000
291  285  CMD_vec_unity-fallback  0.000000
290  284   CMD_vec_unity-fallbac  0.000000
289  283           CMD_vec_picup  0.000000
---------------------------------------------------------
Top-5 most important features of DT. in process activities

     idx                    name     score
453  449            Status_vec_-  0.509243
0    455                     PID  0.437300
449  445             State_vec_E  0.053457
301  297  CMD_vec_oneconf-servic  0.000000
313  309          CMD_vec_<node>  0.000000
312  308       CMD_vec_<mlocate>  0.000000
311  307   CMD_vec_<invoke-rc.d>  0.000000
310  306         CMD_vec_<gdbus>  0.000000
309  305         CMD_vec_<fuser>  0.000000
308  304       CMD_vec_<firefox>  0.000000
---------------------------------------------------------
Top-5 most important features of GBT. in memory activities

     idx                    name     score
5    428                     PID  0.402224
0    423                  MINFLT  0.232126
7      1         CMD_vec_apache2  0.077600
44    38        CMD_vec_<vsftpd>  0.054695
4    427                  VSTEXT  0.054267
2    425                   RGROW  0.043172
49    43         CMD_vec_firefox  0.019082
42    36  CMD_vec_indicator-appl  0.016799
23    17         CMD_vec_tcpdump  0.014596
145  139  CMD_vec_kworker/3:2-cg  0.013476
---------------------------------------------------------
Top-5 most important features of GBT. in process activities

     idx                 name     score
0    455                  PID  0.439076
453  449         Status_vec_-  0.100043
2    457                TSLPI  0.078192
22    18      CMD_vec_tcpdump  0.059260
4      0         CMD_vec_atop  0.046468
447  443          State_vec_I  0.032090
454  450         Status_vec_0  0.027229
446  442          State_vec_S  0.024691
16    12  CMD_vec_hud-service  0.023799
24    20     CMD_vec_node-red  0.022904
---------------------------------------------------------
```

Discussion of which models are better
As can be seen in ## --- 2.3.3 --- ##, we care about the value of recall to better predic
t the attacks.
The GBT. models for both activities performed better than the DT. models did.
As we look into the values of recall, the DT. models got the values of .247 and .392, on
the other hand,
the GBT. models got the values of .338 and .474 which are much higher than DT. models.
Further than recall value, as we look at the values of areaUnderROC which is a performanc
e measurement
for classification problem that can measure the ability of a model at predicting 0s as 0s
and 1s as 1s.
The values of areaUnderROC in DT. models are .519 and .398,
on the other side, the GBT. models got .812 and .800 which are much higher than DT. model
s as well.
According to the chart below, we can say that performance of the DT. models measured by a
reaUnderROC
are (F) and below (F) while the performance of GBT. models are (B) and (B).
In conclusion, I would choose GBT. models for both activities since the performance of GB
T. models measured by
recall and areaUnderROC are much better than the DT. models.

```
---Performance measured by areaUnderROC---
.90-1   = excellent (A)
.80-.90 = good (B)
.70-.80 = fair (C)
.60-.70 = poor (D)
.50-.60 = fail (F)

I will not include the <ts> column in my selected models.  As can be seen in the plot fro
m
## --- 1.3.3 - Memory Activity Plot 2  --- ## and ## --- 1.3.3 - Process Activity Plot 2
--- ##,
All of the attacks for both memory and process activities concentrated in the specific ti
me.
If I include the <ts> into my models, the performance for all the models would be
much better than without <ts>.  However, the feature importance of <ts> column would be
exetremely high(even 1.0), which means the model only use <ts> to predict the attack.
This is not a good thing to my models since we will not know "when" the attacks will happ
en
if we use the models to predict the cyber attacks in the future.
I am not saying that the columns containing the information like <ts> is useless,
there is still some cases that we can choose columns like <ts> to build our model.
For instance, the <ts> columns indicates the frequency of the cyber attacks such as once
a week,
or the <ts> column indicates that the system is more possible to be attacked on Sunday.
Those are actually valuable information for building a prediction model, however in this
case,
the <ts> columns from both use case only indicated that all of the cyber attacks focused
on
a certain period of time.  And this information is not going to help us predict the comin
g attacks, therefore,
I will not selecte <ts> column as one of the feature columns for both use case.

Reference List
http://gim.unmc.edu/dxtests/roc3.htm
```

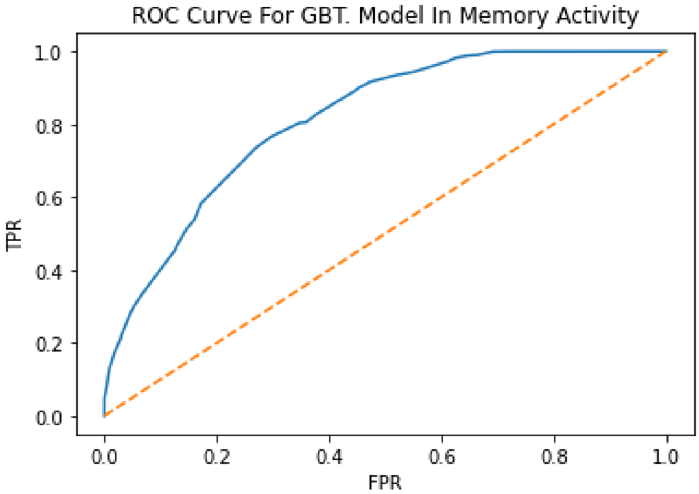
In [213]:

```python
## --- 2.3.4 --- ##
import pyspark.sql.functions as F
import pyspark.sql.types as T

# visualize the ROC curve for the selected Pipeline models

def confusion_matrix(predictions):
    # Calculate the elements of the confusion matrix
    TN = predictions.filter('prediction = 0 AND label = 0').count()
    TP = predictions.filter('prediction = 1 AND label = 1').count()
    FN = predictions.filter('prediction = 0 AND label = 1').count()
    FP = predictions.filter('prediction = 1 AND label = 0').count()
    return TP,TN,FP,FN

def tpr_fpr_all_thresholds(thresholds, prob_df):
    """
    a method that loops through all of the given thresholds and returns the TPR and FPR a
s two lists.
    """
    tpr = []
    fpr = []

    # loop through all the given thresholds and compute the tpr, fpr
    for threshold in thresholds:
        prob_df = prob_df.withColumn('prediction',F.when(prob_df.positive_prob > thresho
ld,1).otherwise(0))
        prob_df.cache()
        tp,tn,fp,fn = confusion_matrix(prob_df)
        prob_df.unpersist()
        tpr.append(tp/(tp+fn))
        fpr.append(fp/(fp+tn))

    return tpr, fpr
```

```
to_array = F.udf(lambda v: v.toArray().tolist(), T.ArrayType(T.FloatType()))
thresholds = np.linspace(0, 1, 100)

# compute TPR, FPR for memory activity
# Splitting the probability to 2 parts using the UDF
df = gbt_memory_attack_prediction.withColumn('probability', to_array('probability'))
# A new df which contains the probabilites in separate columns
prob_df = df.select(df.probability[0].alias('negative_prob'),df.probability[1].alias('po
sitive_prob'),'label')
tpr_memory, fpr_memory = tpr_fpr_all_thresholds(thresholds, prob_df)

# compute TPR, FPR for process activity
# Splitting the probability to 2 parts using the UDF
df = gbt_process_attack_prediction.withColumn('probability', to_array('probability'))
# A new df which contains the probabilites in separate columns
prob_df = df.select(df.probability[0].alias('negative_prob'),df.probability[1].alias('po
sitive_prob'),'label')
tpr_process, fpr_process = tpr_fpr_all_thresholds(thresholds, prob_df)
```

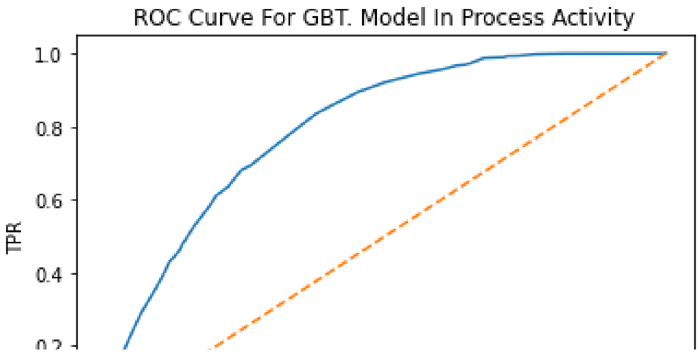
In [214]:

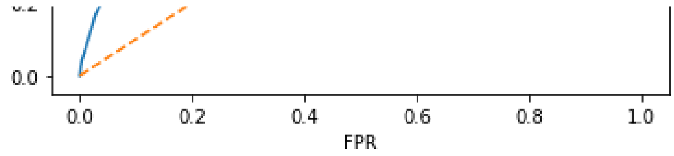
```
## --- 2.3.4 --- ##
x = [i/10 for i in range(11)]
y = [i/10 for i in range(11)]
plt.plot(fpr_memory, tpr_memory)
plt.plot(x, y, linestyle='dashed')
plt.ylabel('TPR')
plt.xlabel('FPR')
plt.title('ROC Curve For GBT. Model In Memory Activity')
plt.show()
```



In [215]:

```
## --- 2.3.4 --- ##
x = [i/10 for i in range(11)]
y = [i/10 for i in range(11)]
plt.plot(fpr_process, tpr_process)
plt.plot(x, y, linestyle='dashed')
plt.ylabel('TPR')
plt.xlabel('FPR')
plt.title('ROC Curve For GBT. Model In Process Activity')
plt.show()
```





In [216]:

```
## --- 2.3.5 --- ##
# get the bigger training dataset for memory activity
training_memory_attack = train_memory.filter(col("attack") == 1)
training_memory_non_attack = train_memory.filter(col("attack") == 0).sample(0.2289626, s
eed=2020)
# combine the attack and non-attack to create a new training dataset
new_trainig_memory = training_memory_attack.unionAll(training_memory_non_attack)
# check the ratio of attack and non-attack
memory_attack = new_trainig_memory.filter(col("attack") == 1).count()
memory_non_attack = new_trainig_memory.filter(col("attack") == 0).count()
print('Ratio of training dataset in memory activity')
print('Attack / Non-Attack = ', memory_attack/memory_non_attack)
new_trainig_memory.cache()

# get the bigger training dataset for process activity
training_process_attack = train_process.filter(col("attack") == 1)
training_process_non_attack = train_process.filter(col("attack") == 0).sample(0.35529648
, seed=2020)
# combine the attack and non-attack to create a new training dataset
new_trainig_process = training_process_attack.unionAll(training_process_non_attack)
# check the ratio of attack and non-attack
process_attack = new_trainig_process.filter(col("attack") == 1).count()
process_non_attack = new_trainig_process.filter(col("attack") == 0).count()
print('Ratio of training dataset in process activity')
print('Attack / Non-Attack = ', process_attack/process_non_attack)
new_trainig_process.cache()

# rename the label column
new_trainig_memory = new_trainig_memory.withColumnRenamed('attack', 'label')
# retrain gbt model for memory activities
new_memory_gbt_model = memory_gbt_pipeline.fit(new_trainig_memory)
# rename the label column
new_trainig_process = new_trainig_process.withColumnRenamed('attack', 'label')
# retrain gbt model for process activities
new_process_gbt_model = process_gbt_pipeline.fit(new_trainig_process)
```

```
Ratio of training dataset in memory activity
Attack / Non-Attack =  0.5
Ratio of training dataset in process activity
Attack / Non-Attack =  0.5
```

In [217]:

```
## --- 3.1 --- ##
from pyspark.ml.clustering import KMeans

iris_df = spark.createDataFrame([(4.7, 3.2, 1.3, 0.2),(4.9, 3.1, 1.5, 0.1),
                                  (5.4, 3.9, 1.3, 0.4),(5.0, 3.4, 1.6, 0.4),
                                  (5.1, 3.8, 1.6, 0.2),(4.9, 2.4, 3.3, 1.0),
                                  (6.6, 2.9, 4.6, 1.3),(5.6, 3.0, 4.5, 1.5),
                                  (5.7, 2.6, 3.5, 1.0),(5.8, 2.6, 4.0, 1.2),
                                  (5.8, 2.8, 5.1, 2.4),(6.2, 2.8, 4.8, 1.8),
                                  (6.0, 3.0, 4.8, 1.8),(6.7, 3.1, 5.6, 2.4),
                                  (6.7, 3.0, 5.2, 2.3),(6.2, 3.4, 5.4, 2.3)],
                                 ['sepal_length', 'sepal_width',
                                  'petal_length', 'petal_width'])

assembler = VectorAssembler(inputCols=['sepal_length', 'sepal_width',
                                       'petal_length', 'petal_width'],
                            outputCol='features')

kmeans = KMeans(k=3).fit(assembler.transform(iris_df))
```

```
print("There are 10 jobs observed when training the KMeans clustering model above.")
```

There are 10 jobs observed when training the KMeans clustering model above.

Completed Jobs (267)

Page: 1 2 3 > 3 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
266	collect at ClusteringSummary.scala:49 collect at ClusteringSummary.scala:49	2020/09/27 23:30:14	2 s	2/2	202/202
265	collectAsMap at KMeans.scala:300 collectAsMap at KMeans.scala:300	2020/09/27 23:30:13	0.1 s	2/2	4/4
264	collectAsMap at KMeans.scala:300 collectAsMap at KMeans.scala:300	2020/09/27 23:30:13	0.1 s	2/2	4/4
263	countByValue at KMeans.scala:418 countByValue at KMeans.scala:418	2020/09/27 23:30:13	0.1 s	2/2	4/4
262	collect at KMeans.scala:395 collect at KMeans.scala:395	2020/09/27 23:30:13	28 ms	1/1	2/2
261	sum at KMeans.scala:390 sum at KMeans.scala:390	2020/09/27 23:30:13	26 ms	1/1	2/2
260	collect at KMeans.scala:395 collect at KMeans.scala:395	2020/09/27 23:30:13	21 ms	1/1	2/2
259	sum at KMeans.scala:390 sum at KMeans.scala:390	2020/09/27 23:30:13	25 ms	1/1	2/2
258	takeSample at KMeans.scala:370 takeSample at KMeans.scala:370	2020/09/27 23:30:13	19 ms	1/1	2/2
257	takeSample at KMeans.scala:370 takeSample at KMeans.scala:370	2020/09/27 23:30:12	0.2 s	1/1	2/2

In [218]:

```
## --- 3.2 --- ##
print("Job ID 257, 258: The input of a set of points from assembled DataFrame,")
print("                 and place the three centroids randomly")
print("")
print("Job ID 259-262: Execute iterations of Lloyd's algorithm until converged")
print("                Start mapping the centers, statistics, and dimensions,")
print("                and calculate the distance between points and centroids to")
print("                find out which point is the nearest to which centroid")
print("                After that, recompute the new center for each cluster")
print("                by compute the average the center of gravity for each cluster")

print("Note: In this step: The val clusterWeightSum is needed to calculate the new cluste
r centers")
print("      e.g. cluster center = sample1 * weight1/clusterWeightSum + sample2 * weight2
/clusterWeightSum + ...")
print("")

print("Job ID 263-265: After converging, reduce the mapped data and")
print("                update the cluster centers and costs(e.g.iterationTimeInSeconds ).
")
print("")

print("Job ID 266: Collect the summary of clustering algorithms at ClusteringSummary to g
et the")
print("            variables such as cluster(Cluster centers of the transformed data) and
")
print("            clustersizes(number of data points in each cluster.)")

print("")
print("Reference List")
print("1. https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spa
rk/mllib/clustering/KMeans.scala#L357")
print("2. https://spark.apache.org/docs/3.0.1/api/scala/org/apache/spark/ml/clustering/Cl
usteringSummary.html")
```

```
Job ID 257, 258: The input of a set of points from assembled DataFrame,
                 and place the three centroids randomly

Job ID 259-262: Execute iterations of Lloyd's algorithm until converged
                Start mapping the centers, statistics, and dimensions,
```

```
                and calculate the distance between points and centroids to
                find out which point is the nearest to which centroid
                After that, recompute the new center for each cluster
                by compute the average the center of gravity for each cluster
Note: In this step: The val clusterWeightSum is needed to calculate the new cluster cente
rs
      e.g. cluster center = sample1 * weight1/clusterWeightSum + sample2 * weight2/cluste
rWeightSum + ...

Job ID 263-265: After converging, reduce the mapped data and
                update the cluster centers and costs(e.g.iterationTimeInSeconds ).

Job ID 266: Collect the summary of clustering algorithms at ClusteringSummary to get the
            variables such as cluster(Cluster centers of the transformed data) and
            clustersizes(number of data points in each cluster.)
```

Reference List

1. https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/KMeans.scala#L357
2. https://spark.apache.org/docs/3.0.1/api/scala/org/apache/spark/ml/clustering/ClusteringSummary.html

In [ ]:

```

```