Data Mining For Networks

Project : Anomaly detection 2019-2020

at

Université Côte d'Azur - Sophia Antipolis, France

Student:
Abdelkarim HAFID
Abdelhadi Lebbar
Bernard Tamba

Supervisor:
GIROIRE Frederic

Our goal in this project is to detect detect anomalous flows in IP traffic. Detecting those flows is of a great importance since they might help to prevent some network attacks such as port scan, denial of service an so on. We aim in this project to associate the flows to graphs call *graphlets* in an optimized way, I.e not considering all flows related to an end host but only significant one, while maintaining the characteristics of the end host. We then use SVM along with kernels in order to train a model and to do further predictions on endhosts behavior. To do the work, we used networkx and scikit learn libraries mainly.

ANALYZING THE PAPER

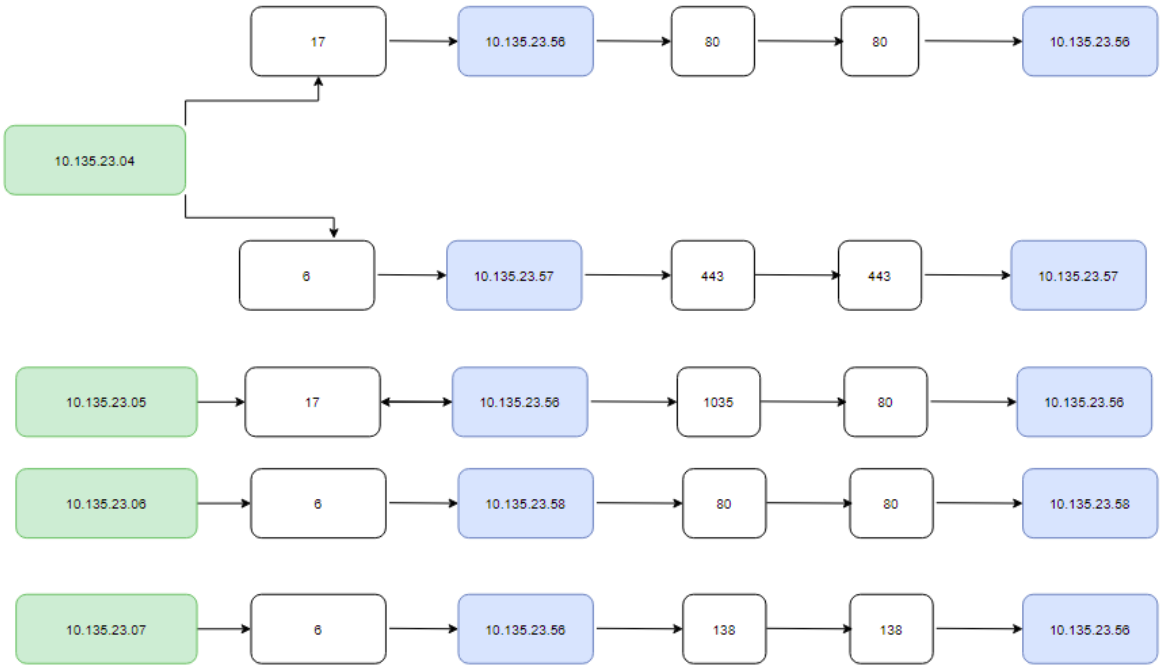
To recall a graphlet of an end host is all the flows related to this end host. In another words all the flows that have this end host as their source IP. In their paper, Karagiannis and al. explain the concept of profiling an end host using graphlets. Profiling an end host using those graphlets refers to building all the activities related to an end host as a graph. Hence, this graph associated to that end host will help to detect some anomalous traffic or an attack against our end host. For instance, several traffic tcp connections from a single host, might let us know that it is a port scanning against our end host. Some other attacks can also be detected in that way such as denial of service or even distributed denial of service, by having several connections from huge amount of IP addresses, and so on.

In the paper, they optimize the way of profiling end host by introducing the notions of activity graphlets and profile graphlets. Activity graphlets contain all the activity associated to an end host(where this end host is the source IP) and the profile one contains all the significant flows of the end host. Simply put, it is the compression of the activity graphlet. The profiling is then done with profile graphlet, since it contains less data and has almost the same characteristics as the original data. This profile using profile graphlets will help us to capture representative information of the end host. It will also help to have a compact representation of the end host activities.

In order to highlight, the concept of graphlets, we plotted the graphlets associated to the following flows.

format: srcIP	protocol	dstIP	sPort	dPort
10.135.23.04	17	10.135.23.56	80	80
10.135.23.05	17	10.135.23.56	1035	80
10.135.23.04	6	10.135.23.57	443	443
10.135.23.06	6	10.135.23.58	80	80
10.135.23.07	6	10.135.23.56	138	138

Something to be noticed is that given a flow, a graphlet is build based on source IPs, i.e to a given source IP is associated a given graphlet. Given thus a set of flows, an end host(or a given source IP) can contains more than one flow. For instance in our case here, the IP address 10.135.23.04 will have two flows and all the others will contain only one flow each in their graphlet. The graphlets associated to each of them is the following.



PRACTICAL WORK

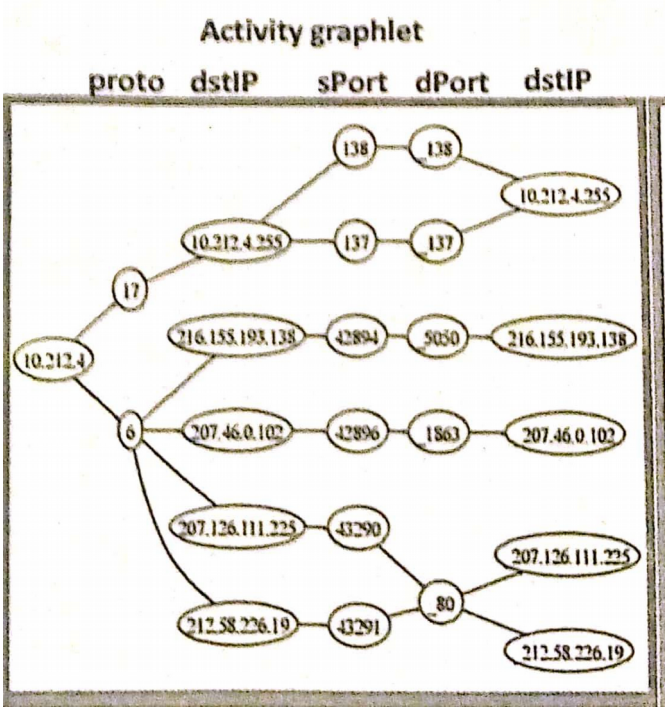
In this practical work, we worked with two sets of data called annotated-trace and not-anotated-trace. The first one helps us to build our model in order to detect malicious end hosts. While the second one helps us to do predictions on our model. To notice that our model was built using Support Vector Machine(SVM) and Kernels.

Building End Hosts Profiles(Question 4,5**): Activity Graphlet**

Having our first data set, we profiled the end hosts contained in it. To do so, we used to method of Karagiannis and al in their paper. We started by building an activity graphlet of all our end hosts. We implemented a function that takes in parameter an IP address(an end host) and returns its activity graphlet. In another words, this function gives all the flows associated to each end host. Nodes of the output graph are protocol(TCP or UDP), Destination IP, Source IP, Destination Port, Source Port and the Destination IP(again). The vertices between them only means that they belong to the same flow as it is seen in the examples below. The code of the function is given in the figure below.

```
def Activity(source):
    Specific_data = Data.loc[Data[0] == int(source),:]
    G=nx.Graph()
    G.add_nodes_from(["src_ip"+str(source)])
    for i in range(len(Specific_data)):
        G.add_nodes_from(["dest_ip:"+str(Specific_data.iloc[i,1]),"dest_ipp:"+str(Specific_data.iloc[i,1]),
                          "prot:"+str(Specific_data.iloc[i,2]),"src_port"+str(Specific_data.iloc[i,3]),
                          "des_port"+str(Specific_data.iloc[i,4])])
        G.add_edges_from([("src_ip"+str(source),"prot:"+str(Specific_data.iloc[i,2])),
                          ("prot:"+str(Specific_data.iloc[i,2]),"dest_ip:"+str(Specific_data.iloc[i,1])),
                          ("dest_ip:"+str(Specific_data.iloc[i,1]),"src_port"+str(Specific_data.iloc[i,3])),
                          ("src_port"+str(Specific_data.iloc[i,3]),"des_port"+str(Specific_data.iloc[i,4])),
                          ("des_port"+str(Specific_data.iloc[i,4]),"dest_ipp:"+str(Specific_data.iloc[i,1]))])
    return G
```

In order to test whether our function works correctly, we tried to test it on this example taken from the paper.

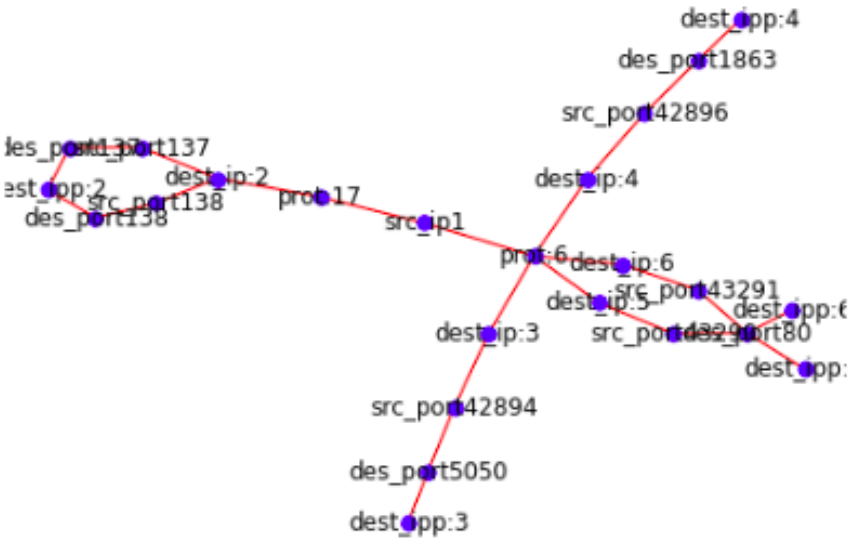


To do the test, we convert this graph as a table containing a set of flows as shown below.

	A	B	C	D	E
1	1	2	17	138	138
2	1	2	17	137	137
3	1	3	6	42894	5050
4	1	4	6	42896	1863
5	1	5	6	43290	80
6	1	6	6	43291	80

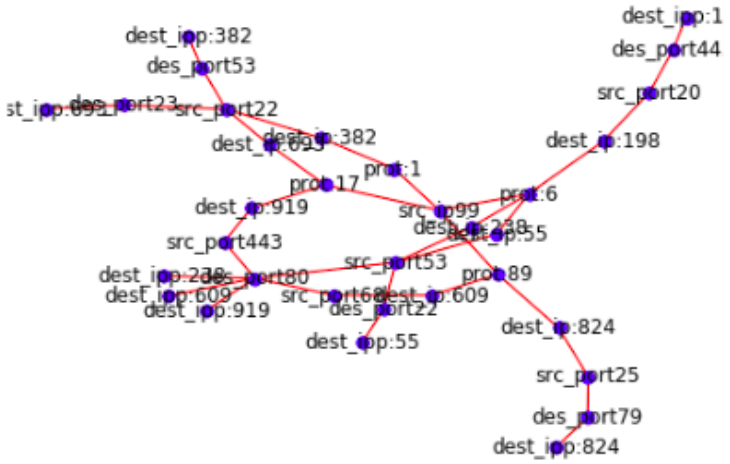
Finally, giving this data to our function, we obtained the same output activity graphlet as the one of the paper as shown below.

```
: nx.draw(Activity(1), with_labels=True, node_color="b",edge_color="r",node_size=50)
```



Afterwards, we apply the function on our real data set, since we were sure that the function gave the correct output. One example of the execution on an end host of our data set is given below. This figure gives, all the activities associated to the host having the Ip address 99.

```
Data = (pd.read_csv('annotated-trace.csv',header=None)).iloc[:,0:6].drop_duplicates()
nx.draw(Activity(99), with_labels=True, node_color="b",edge_color="r",node_size=50)
```



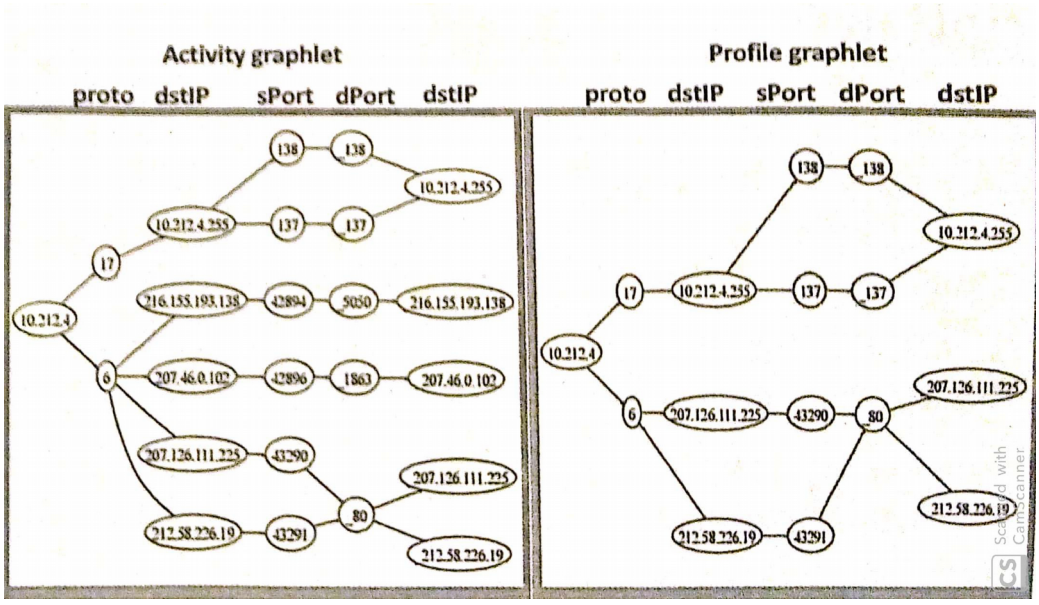
Building End Hosts Profiles(Question 4,5)**: Profile Graphlet**

Based on the paper, we figure out that the activity graphlet was not an efficient way to build a profile of an end host, since it contains lot of information that are not all useful. We used instead the profile graphlet by only considering the significant flows from the activity graphlet. A significant flow is nothing but a flow containing at least one significant node, which is a node with in degree or out degree larger than 1.

To do so, we build a function taking as parameter an IP address and then return as output the profile graphlet. The profile graphlet is built base on the idea of the significant nodes. For a given IP address, if we find at least the same node in different flows associated to that IP address, the latter is considered as significant node. Hence, each flows containing this node is considered as a significant flow and then automatically putted in the profile graphlet. Our code for building a profile graphlet is given below.

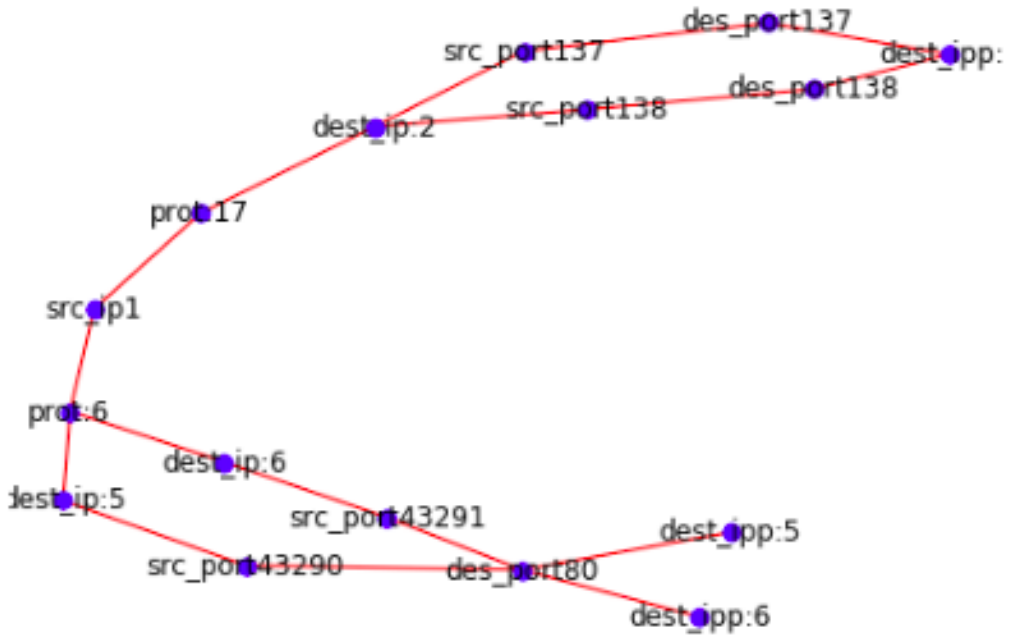
```
def Profile(source,state=True):
    if state:
        Specific_data = Data.loc[Data[0] == source,:]
    else:
        Specific_data = predict_data.loc[predict_data[0] == source,:]
    Specific_data.index = [i for i in range(len(Specific_data))]
    Significant_Flows=[]
    Non_Significant_Flows=[]
    for i in range(len(Specific_data)):
        for j in range(len(Specific_data)):
            if i==j:
                continue
            if Specific_data.iloc[i,1] == Specific_data.iloc[j,1] or Specific_data.iloc[i,
                           3] == Specific_data.iloc[j,3] or Specific_data.iloc[i,4] == Specific_data.iloc[j,4]:
                Significant_Flows.append(i)
                break

    for i in Specific_data.index:
        if i not in Significant_Flows:
            Non_Significant_Flows.append(i)
    Specific_data = Specific_data.drop(Non_Significant_Flows)
    #Ploting the graph with networkx

    G=nx.Graph()
    G.add_nodes_from(["src_ip"+str(source)])
    for i in range(len(Specific_data)):
        G.add_nodes_from(["dest_ip:"+str(Specific_data.iloc[i,1]),"dest_ipp:"+str(Specific_data.iloc[i,1]),
                          "prot:"+str(Specific_data.iloc[i,2]),"src_port"+str(Specific_data.iloc[i,3]),
                          "des_port"+str(Specific_data.iloc[i,4])])
        G.add_edges_from([("src_ip"+str(source),"prot:"+str(Specific_data.iloc[i,2])),
                          ("prot:"+str(Specific_data.iloc[i,2]),"dest_ip:"+str(Specific_data.iloc[i,1])),
                          ("dest_ip:"+str(Specific_data.iloc[i,1]),"src_port"+str(Specific_data.iloc[i,3])),
                          ("src_port"+str(Specific_data.iloc[i,3]),"des_port"+str(Specific_data.iloc[i,4])),
                          ("des_port"+str(Specific_data.iloc[i,4]),"dest_ipp:"+str(Specific_data.iloc[i,1]))])
    return G
```

As an application of the script, we applied this function to the following example presented in the article.



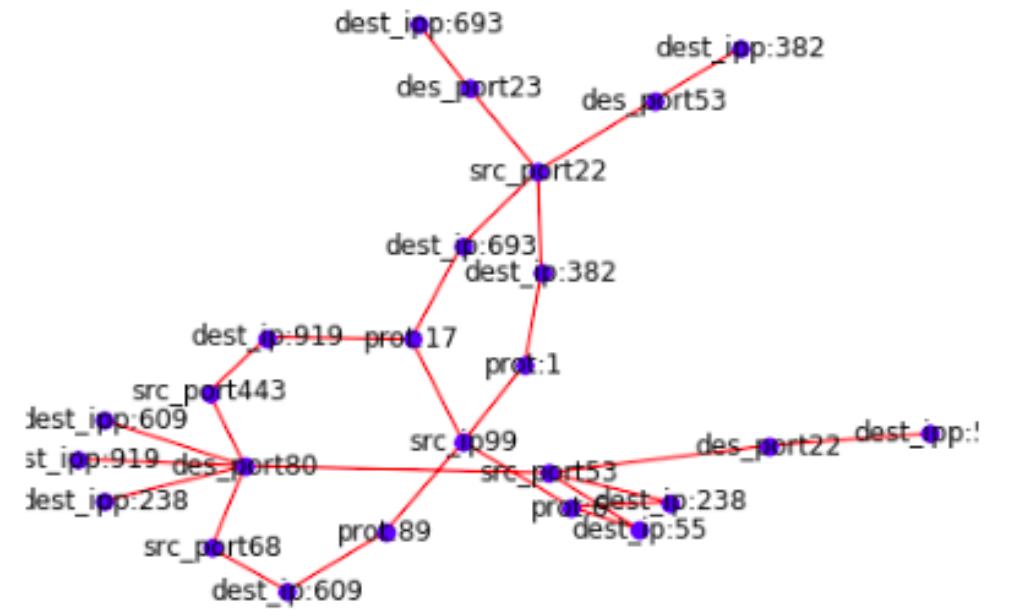
By applying it on this, our function gives the same output for the profile graphlet as shown below.

```
nx.draw(Profile(1), with_labels=True, node_color="b",edge_c
```



After doing this execution, we applied this function on an IP address (end host) chosen randomly from our data set.

```
: nx.draw(Profile(99), with_labels=True, node_color="b",edge_c
```



Comparing the output of the activity graphlet and the profile one on the same end host, one can notice from the figures above that the profile graphlet is more

representative and more compact than the activity one. **All the following questions are hence done using Profile graphlets.**
One must notice that, the idea behind choosing one IP address as argument is to simplify the way of drawing the end host graphlet. Due to visibility issue, we thought that it is more compact and convenient to show a specific chosen IP address graphlet instead of all the graphlets at the same time.

RANDOM WALK KERNELS

After building the end hosts profiles, we want in this part to exploit them in order to find good insights. We want to predict normal and malicious end hosts. In all the subsequent works, we considered an end host to be malicious, **if it is related to at least one malicious flow.** To do so, we used the Random Walk Kernel in two ways. The first one consisted of using the Kernel Trick in order to avoid the mapping of our graphlets in a higher dimensional space, by simply using the following formula:

$$K(G1,G2)=e^{T}(I-lambda*A_x)^{-1}*e$$

where $e^{T}=[1\ 1\ \dots\ 1]$, I is the identity matrix and A_x the dot product of G1 and G2.
The second way consisted of explicitly mapping the graphlets into high dimensional space, I.e we take the power of the adjacency matrix. The adjacency matrix is taken to the power of the length of random walk kernel according to this formula:

$$\sum\left(\sum \lambda^{k} A^{k}\right)$$

To recall the kernel gives the degree of similarity between two graphs. The higher the kernel, the similar are the two graphs.

Kernel Trick(Question 6)

We started by by the method of the kernel trick using the formula above. Since the kernel trick is done between two graphs, we used the profile graphlets of each end host(source IP). For each end host, we computed its kernel trick with all the other end hosts. We ended up thus with a matrix of kernel tricks, looking like the following.

$$\begin{bmatrix} K(G_1,G_1), K(G_1,G_2), \dots, K(G_1,G_n) \\ K(G_2,G_1), K(G_2,G_2), \dots, K(G_2,G_n) \\ \dots \\ K(G_n,G_1), K(G_n,G_2), \dots, K(G_n,G_n) \end{bmatrix}$$

We implemented two functions, the first one called Kernel, takes two graphs and lambda as parameters and it outputs the kernel trick of the two graphs based on the formula showed above. The second one call kernel Trick builds the matrix of kernel tricks by calling for each pair of graphs the first function. The codes of the two functions are given below.

```
def Kernel(G1,G2,landa):
    a=nx.tensor_product(G1,G2)
    a=nx.adjacency_matrix(a).todense()
    ep=np.column_stack(np.ones(len(a)))

    iden=np.eye(len(a),len(a))
    inverse= (iden-landa*a).I
    z=np.dot(ep,inverse)
    value =np.dot(z,ep.transpose())
    return value
```

```
def kernelTrick(landa):
    y=[]
    c1=0
    c2=0
    src_ip=Data.loc[:,0].drop_duplicates()
    M=np.zeros((len(src_ip),len(src_ip)))
    for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
        if i in src_ip.keys():
            y.append(Malicious(src_ip[i]))
            G1=Profile(src_ip[i])
            for j in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
                if j in src_ip.keys():
                    G2=Profile(src_ip[j])
                    M[c1][c2]=Kernel(G1,G2,landa)
                    c1+=1
            c2+=1
            c1=0
    return (M,y)
```

Since that the computation time to execute these functions on our data set is very very high, we reduced our data set in order to reduce the execution time. Even after reducing the size of our data to almost 1000 flows and 630 end hosts, the execution was still high, it was at the range of 20 minutes. The result of the execution is given below.

```
: c,m=kernelTrick(0.6)
  print(c)
  print(m)
```

```
[[1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 ...
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]]
```

Kernel Trick+SVM(Question 6)

Finally, we used SVM algorithm from the scikit learn library in order to train a model that will separate normal and malicious end hosts. Since we already computed our kernel using the kernel trick, we set the kernel parameter of SVM to *precomputed.* In order to train a model, SVM takes as parameter feature values and target values. The feature values in our case is the kernel that is computed in the figure above. The target value tells whether an end host is malicious or not. The target value is 0 when an end host is normal and it is 1 when an end host is malicious. To have the latter, we build a function call

Malicious that takes as parameter an IP address and tells whether that IP address is malicious or not. That function is called on all the IP addresses and the target values are computed as shown in the figure below.

```
def Malicious(ip):
    for i in range(len(Data)):
        if(Data.iloc[i,0]==ip and Data.iloc[i,5]=="anomaly"):
            return 1
    return 0
def kernelTrick(landa):
    y=[]
    c1=0
    c2=0
    src_ip=Data.loc[:,0].drop_duplicates()
    M=np.zeros((len(src_ip),len(src_ip)))
    for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
        if i in src_ip.keys():
            y.append(Malicious(src_ip[i]))
            G1=Profile(src_ip[i])
            for j in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
                if j in src_ip.keys():
                    G2=Profile(src_ip[j])
                    M[c1][c2]=Kernel(G1,G2,landa)
                    c1+=1
            c2+=1
            c1=0
    return (M,y)
```

The execution gives the following:

```
c,m=kernelTrick(0.6)
print(c)
print(m)
```

```
[[1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 ...
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]]
```

Feature values

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0]
```

Target Values

We finally give these values to the SVM in order to build our model. The code of the SVM is given below.

```
from sklearn import svm
kernel_trick_Svm=svm.SVC(kernel='precomputed')
kernel_trick_Svm.fit(c,m)
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='precomputed', max_iter=-1, probability=False, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
```

This built model based on the kernel trick and SVM, helps to separate normal from malicious end hosts.

Transformation of graphlets into higher dimensional space(Question 7)

As for th kernel trick, we wrote a function that computes the random walk kernel of end hosts graphlets. The difference in this part is that, we build two functions by taking into consideration the length of the random walk kernel, I.e we should computer powers of the adjacency matrix of the dot product graph, according to the formula given above.

The first function call *RandomWalkKernel*, takes as parameters two graphs G1 and G2, the length of walks and lambda. This function first the dot product of the two graphs in order to compute the random kernel kernel. Hence, its output is the random walk kernel of the graphs given as parameters.

The second function call *Kernel2* takes as parameter the length of walks and compute the kernel of all pair of graphs using the first function(*RandomWalkKernel).* It also displays the target values, this value is 1 if an end host is malicious and 0 if not.

The code of the two functions are given below.

```
from numpy.linalg import matrix_power
def RandomWalkkernel(G1,G2,k,landa=0.5):
    a1=nx.convert_matrix.to_numpy_array(G1)
    b1=nx.convert_matrix.to_numpy_array(G2)
    dot_product=nx.tensor_product(G1,G2)
    dot_product=nx.adjacency_matrix(dot_product).todense()
    result=0
    val=np.zeros((len(a1)*len(b1),len(a1)*len(b1)))
    for l in range(k):
        val+=(landa**l)*matrix_power(dot_product, l)

    for i in range(len(a1)*len(b1)):
        for j in range(len(a1)*len(b1)):
            result+=val[i][j]
    return result
```

```python
def kernel2(k):
    y=[]
    c1=0
    c2=0
    src_ip=Data.loc[:,0].drop_duplicates()
    M=np.zeros((len(src_ip),len(src_ip)))
    for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
        if i in src_ip.keys():
            y.append(Malicious(src_ip[i]))
            G1=Profile(src_ip[i])
            for j in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
                if j in src_ip.keys():
                    G2=Profile(src_ip[j])
                    M[c1][c2]=RandomWalkkernel(G1,G2,k)
                    c1+=1
            c2+=1
            c1=0
    return (M,y)
```

Due to the high execution time of the above script, we chose a subset of our dataset, that contains almost 1000 rows. Even though the execution time was higher.

We run the functions by using several values of length of walks that are 4, 10,20,50,100. We noticed that the more k is higher ,the more the execution time is longer. The execution of k=4 and k=10 are given below.

```python
c,m=kernel2(4)
print(c)
print(m)
```

```
[[1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 ...
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
c,m=kernel2(10)
print(c)
print(m)
```

```
[[1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 ...
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]
 [1. 1. 1. ... 1. 1. 1.]]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Afterwards, we directly trained our model on applying the SVM algorithm on our output for all cases of K. To recall the feature values of the SVM is the kernel and the target says whether an end host is malicious or not.

```
kernel4=svm.SVC(kernel='precomputed')
kernel4.fit(c,m)
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='precomputed', max_iter=-1, probability=False, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
```

```
kernel10=svm.SVC(kernel='precomputed')
kernel10.fit(c,m)
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='precomputed', max_iter=-1, probability=False, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
```

```
c,m=kernel2(20)
print(c)
print(m)
```

```
kernel20=svm.SVC(kernel='precomputed')
kernel20.fit(c,m)
```

```
c,m=kernel2(50)
print(c)
print(m)
```

```
kernel50=svm.SVC(kernel='precomputed')
kernel50.fit(c,m)
```

```
c,m=kernel2(100)
print(c)
print(m)
```

```
kernel100=svm.SVC(kernel='precomputed')
kerel100.fit(c,m)
```

For this one, as K is higher, the execution is longer, while for the one of the kernel trick the execution time is fixed. We then concluded that the execution time of the kernel trick is less than the one depending of the number of walks of length k.

DETECTION PHASE

The idea behind the detection phase is to be able to detect anomalous traffic based on the models we have already built. Giving a set of flows, this phase through our models helps to predict in a set of flows, what are the malicious end host and thus preventing attacks. To do so, we have a dataset call *not-annotated -trace.csv* which contains flows where some flows are annotated and some are not(i.e we do not any clue about whether they are malicious or not). The goal is to allow our built model to predict malicious end hosts inside this dataset. Due to the execution time issue, we picked out a subset of our dataset.

Application on the model(Question 9)

To apply the dataset on the model , we build a function called *PredictkernelTrick* in order to get the kernel matrix of the end hosts. This matrix is considered as the meta-features used in our prediction model. In another words, the model takes as input the kernel matrix and gives the prediction of malicious and normal end hosts.

The code of the function we built as well as the prediction phase on our model are given below.

```
def PredictkernelTrick(landa=0.6):
    c1=0
    c2=0
    y=[]
    src_ip=predict_data.loc[:,0].drop_duplicates()
    M=np.zeros((len(src_ip),len(src_ip)))
    for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
        if i in src_ip.keys():
            G1=Profile(src_ip[i],state=False)
            for j in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
                if j in src_ip.keys():
                    G2=Profile(src_ip[j],state=False)
                    M[c1][c2]=Kernel(G1,G2,landa)
                    c1+=1
            c2+=1
            c1=0
    return M
```

```
kernel4_predict=kernel4.predict(X_test)
display(kernel4_predict)
```

```
kernel10_predict=kernel10.predict(X_test)
display(kernel10_predict)
```

```
kernel20_predict=kernel20.predict(X_test)
display(kernel20_predict)
```

```
kernel50_predict=kernel50.predict(X_test)
display(kernel50_predict)
```

```
kernel100_predict=kernel100.predict()
display(kernel100_predict)
```

Due to the difference in terms of size between the training data(subset of *annotated-trace*) and the test data(subset of *not-annotated-trace)*, we opted for another approach in order to predict our new dataset. This approach aims at

using SVM in it basic form without using any kernel. The goal being to find whether it could detect some malicious flows. The code is given below.

```
from sklearn import svm
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
columns=['srcIp', 'dstIp','proto','portSrc', 'portDst','anomaly']
OurData=pd.read_csv("annotated-trace.csv",names=columns)
x=OurData.drop('anomaly',axis=1)
y=OurData['anomaly']

x_train, x_test, y_train, y_test =train_test_split(x, y, test_size=0.10)

OurData=pd.read_csv("not-annotated-trace.csv",names=columns)
#x_test=OurData.drop('anomaly',axis=1)
#y_test=OurData['anomaly']


Svc_Cl=svm.SVC(kernel='poly',degree=10)
Svc_Cl.fit(x_train,y_train)
y_pred=Svc_Cl.predict(x_test)

print(y_pred)
dataframe=pd.DataFrame(y_pred)
dataframe.to_excel("output2.xlsx",
             sheet_name='Sheet_name_1')
print(classification_report(y_test,y_pred))
```

Potential False Positives and False Negatives(Question 10)

To recall a false positive is when the prediction is anomalous while the reality in *not-annotated-trace*, the same observation is seen as normal. The false negative is when the prediction is normal while the the reality in *not-annotated-trace*, the same observation is seen as malicious.

In order to detect false positive and false negative, we build a function called *Malicious_predict* that gives the behavior of each flow in the test data. If a flow is normal, the function puts at the ranking of that flow 0, if it is malicious, 1 is putted, otherwise if there are missing values, 0.5 is assigned. The output of this function is a list containing the behavior of all the flows in *not-annotated-trace*.

In order to detect those false positive/negative, we compare the output of this function considered Y_test and the output of the SVM after the prediction called Y_pred. A false positive occurs, when Y_pred[i] is anomalous and Y_test[i] is normal and the false negative occurs when Y_pred[i] is normal and Y_test[i] is malicious. The code of *Malicious_predict* function is given in the figure below.

```
def Malicious_predict():
    def predict(ip):
        for i in range(len(predict_data)):
            if (predict_data.isnull().iloc[i,5]):
                return 0.5
            if(predict_data.iloc[i,0]==ip and predict_data.iloc[i,5]=="anomaly"):
                return 1
        return 0
    Y_test=[]
    src_ip=predict_data.loc[:,0].drop_duplicates()
    for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
        Y_test.append(predict(src_ip[i]))
    return Y_test
```

IDEAL KERNEL(Question 11)

In this part, we redo the modeling and the detection phases of malicious end hosts by using Ideal Kernels. The ideal kernel gives how similar subgraphs of two different graphs are. Using the ideal kernel, we can find the degree of similarity between two graphs. We will then use those features of the ideal kernel for subgraphs of sizes 5 in order to compare how similar our end hosts graphlets are. The type of subgraph we chose is a chain of size 5, since all our graphlets are chain-like.

In this part, we start by doing the modeling phase, I.e we build our SVM model that will learn based on the ideal kernels what are the malicious and the normal end hosts. Afterwards, we did the detection phase, I.e we make predictions on our SVM model on a new data set.

Modeling Phase: Building the ideal kernel of the graphs

To do the modeling, we build two functions. The first one called *idealkernel* takes as parameter a graph and compute its ideal kernel. That function for a given graph goes through all the nodes of the graph and for each node puts all the paths of length 5 from that node in a list. At the end redundant entries are removed from the lists and the length of the resulted graph is considered as the ideal kernel of length 5. The code of this function is given below.

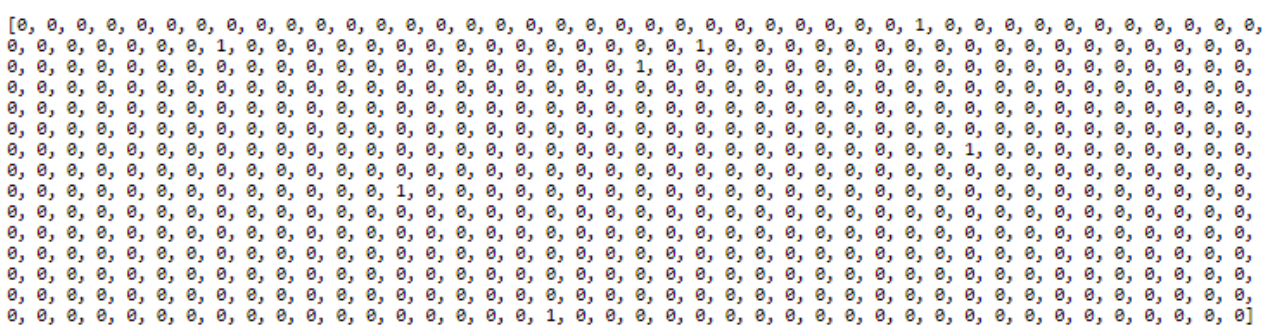
```
def idealKernel(G):
    def idealK(i):
        Ma = []
        for j in range(len(G.nodes)):
            Ma2=[]
            A = list(nx.all_simple_paths(G,i,j))
            for k in A:
                if(len(k)==5):
                    Ma2.append(k)
            Ma.extend(Ma2)
        return Ma

    Final=[]
    for i in range(len(G2.nodes)):
        Inter = Node(i)
        Final.extend(Inter)
    return int(len(Final)/2)
```

The second function called *idealkernelT* loops over all our endhosts and for each of them compute their graphlet and then call upon them the first function. Moreover it also computes the behavior of each host, meaning that for each host it gives whether it is malicious(1) or normal(0). It does this by calling for each end host a function called *Malicious* that gives 0 if the end host is malicious and 0 if not. Hence, the outputs of this function are two lists. The first one containing all the idea kernels of sizes 5 of all our endhosts will be considered as our feature value. The second one containing the behavior of each end host will be considered as our target value to the SVM(cf.section below). The code of this function is given below.

```
def Malicious(ip):
    for i in range(len(Data)):
        if(Data.iloc[i,0]==ip and Data.iloc[i,5]=="anomaly"):
            return 1
    return 0
def idealkernelT():
    y=[]
    src_ip=Data.loc[:,0].drop_duplicates()
    M=list(np.zeros(len(src_ip)))
    for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
        if i in src_ip.keys():
            y.append(Malicious(src_ip[i]))
            G1=Profile(src_ip[i])
            M.append(idealKernel(G1))

    return (M,y)
```

We then finally apply these functions to our dataset. The first output(ideal kernels: our feature values for SVM) is given below.

The second output(the behavior: our target value is also given below)



Modeling Phase: Applying SVM to ideal kernels

After having computed the feature and the target values through the above function, it is the time to train our SVM model. To do so, we use SVM in its very basic form without the use of any type kernel since we already have our features and targets. The SVM takes the ideal kernels as feature value and the behavior(0 or 1 for each corresponding feature value) as target value. The code of our SVM model in given below.

```
from sklearn import svm
clf=svm.SVC()
x=np.array(x_train)
x_train1 = np.reshape(x, (-1, 2))
clf.fit(x_train1,y_train)
```

```
C:\Users\user\Documents\temp\lib\site-packages\sklearn\svm\base.py:193: FutureWarning: The default va
lue of gamma will change from 'auto' to 'scale' in version 0.22 to account better for unscaled featur
es. Set gamma explicitly to 'auto' or 'scale' to avoid this warning.
  "avoid this warning.", FutureWarning)
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
    kernel='rbf', max_iter=-1, probability=False, random_state=None,
    shrinking=True, tol=0.001, verbose=False)
```

Detection Phase

Finally we apply our test data(the *not-annotated-trace* one) on our model in order to find what the predictions made on our data. To do so, we started by building a function that compute the ideal kernels of our subset data taken from the test data. The built function called *predictidealkernel* is based on the *idealkernel* function previously built. The only difference is that the first one compute the ideal kernel for the test data and the last one does it from the train data. The code of this function is given below.

```
def PredictIdealKernel():
    src_ip=predict_data.loc[:,0].drop_duplicates()
    M=list(np.zeros((len(src_ip))))
    for i in range(src_ip.keys()[0],src_ip.keys()[-1]+1):
        if i in src_ip.keys():
            G1=Profile(src_ip[i],state=False)
            M.append(idealKernel(G1))
    return M
```

We then call this function on our test dataset and the result is given below.

```
x_test=PredictIdealKernel()
print(x_test)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6]
```

We finally reach our goal that is to make our model to do predictions on our test dataset. The script below shows how do those predictions are made using SVM.

```
x_test=PredictIdealKernel()
```

```
x=np.array(x_test)
x_test1 = np.reshape(x, (-1, 2))
value_ideal_kernel=clf.predict(x_test1)
```

By running this script,as shown in the output below, we find out that no malicious behavior was predicted. This is due to the fact for the training and the predictions phases we only used a small dataset due to long time the kernels were taking.

```
value_ideal_kernel=clf.predict(x_test1)
display(value_ideal_kernel)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Finally, in terms of execution time, the ideal kernel was executed in less than a minute, I.e ideal kernel is almost more than 20 (or even more for the kernels depending on k=length of walks) minutes faster than the kernel approach for detecting malicious in the dataset. However, this ideal kernel is not that efficient since it only captures gross similarity of subgraphs of sizes 5 between the graphs. The latter does not hence capture the details of graphs as do by the kernel types.

Conclusion

To conclude, this project is of great importance, as it helps to build a profile of end hosts within a network as graphlets and then captures some malicious behavior in those data sets. This can help for instance network administrators and IT security people to easily detect some malicious behavior and moreover predicting them and then prevent them before they happens. However, during the realization of the project, we were limited in terms of hardware resources, as the execution times were really high and we could not do experimentations on all our dataset, but only chose a small subset of this dataset.