**Familiarisation & Visualisation Task**

Basic Idea: We present a frequency based analysis to visualise the detection task and familiarize ourselves with the structure of the different PEs (i.e. Malicious and Benign), eventually juxtaposing it with adversarial samples that are generated. The plots show total number of occurrences of 1 values for different features (i.e. the API calls that were made) in the binary feature vectors of the PEs. These are Sorted according to maximum difference so as to highlight the differences in structuring of Malicious/benign PEs.

**Before Training:**

Malicious vs Benign: We start by looking at the difference between the feature space of Malicious and Benign PEs. It is fair to observe that the nature of a malicious PE is obtained by keeping some common features as 1 (let's call these as malicious API calls) which correspond to the features (or API calls) with high counts in the structure of a malicious PE. Example: API call at 53 or 203 which seems to be present in > 60% of the malicious PEs.A screenshot of a cell phone

Description automatically generated

Figure 1: Count plot of the feature space of Malicious vs Benign PEs - untrained

Malicious vs Adversarial: By looking at the same set of calls for the adversarial samples (superimposed against the Malicious PEs), it can be observed that the adversaries that get generated, try to match the structure of the malicious samples. This is apparent when looking at the features with high counts in Malicious and observing that the adversarial samples tend to have a similar count for those features! We again draw attention to counts at 53 and 203 (and the rest as well!) where the adversarial counts and malicious counts end up having the same heights. Additionally this pattern with the adversarial samples seems to follow if more than 50% (roughly) malicious samples had that particular feature set to 1. This is due to the constraint that the adversarial samples must maintain their malicious behaviour and must contain 1s for the same API calls present as the malicious data.

A screenshot of a cell phone

Description automatically generated

Figure 2: Count plot of the feature space of Malicious vs Adversarial PEs - untrained

Benign vs Adversarial: Keeping this in mind and superimposing the adversarial counts against Benign samples, another interesting pattern can be observed. The adversarial samples that get generated, start showing increased counts for those features which had high(-ish) counts in the benign samples, as if trying to mimic their structure. This is further highlighted when we look at the malicious vs adversarial plot again to see that counts at feature 107 or 174 we observed to be more than twice as much in adversarial samples than in malicious. Eventhough this is an admittedly naive way to try and fool the defending system, it can be intuitively observed that engineering a successful adversarial sample (ideally) requires enhancing a malicious sample by trying to increasingly mimic the structure of a benign sample while maintaining its inherent malicious nature. A screenshot of a cell phone

Description automatically generated

Figure 3: Count plot of the feature space of Benign vs Adversarial PEs - untrained

**After Training:**

Malicious vs Adversarial: After training the network for 50 epochs, the adversarial samples are almost identical to the malicious samples as expected (the trend that we saw earlier even without training) and is clear from the plot below. The counts of various features is quite similar.

A screenshot of a cell phone

Description automatically generated

Figure 4: Count plot of the feature space of Malicious vs Adversarial PEs - trained

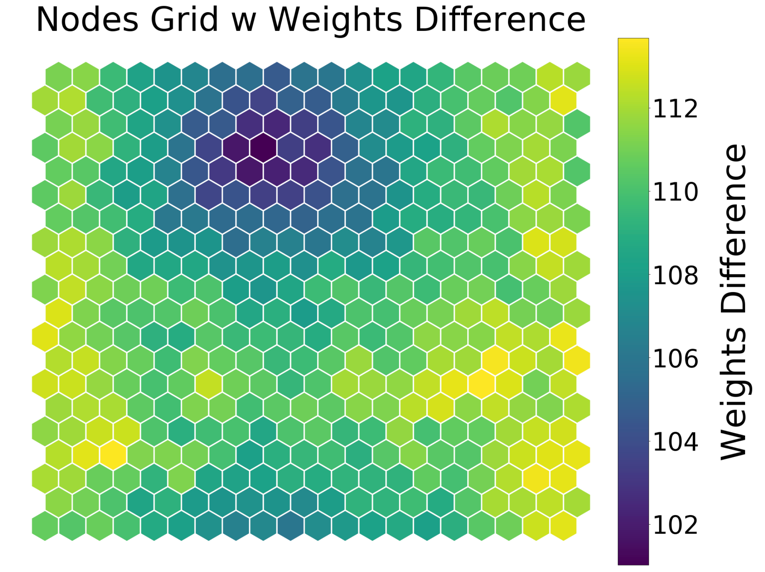
Benign vs Adversarial: Comparing with the benign samples after training, it seems that the adversarial samples take a different (possibly smarter) approach than before, which was to blatantly try to mimic the structure of the benign samples. This is highlighted by looking at the feature counts at 107 or 174 (just as before in the benign vs adversarrial plot w/o training) where the counts have reduced. This is an indication of the fact that the system generating these adversarial attacks has moved on from the blatant copying (of benign structures) to a more intricate way of trying to get past the defending system! This is further supported by the higher evasion rate observed by testing it against a defender that is not trained on adversarial samples.

A screenshot of a cell phone

Description automatically generated

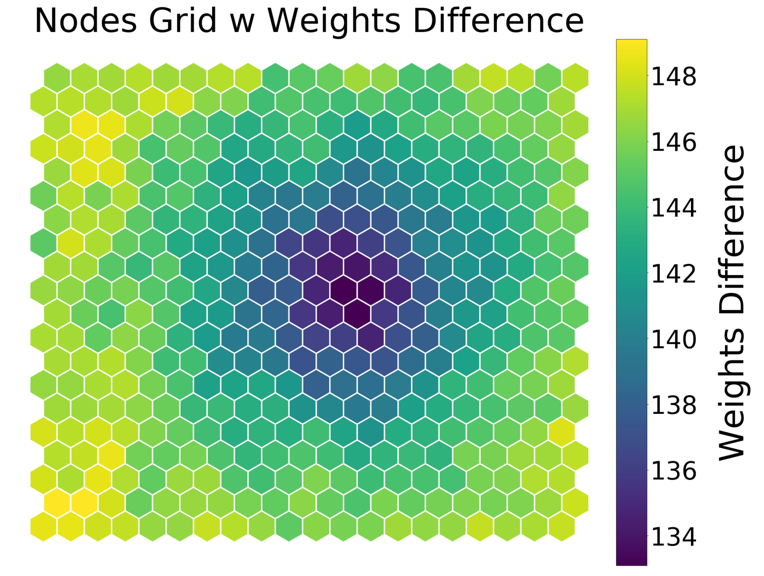
Figure 5: Count plot of the feature space of Benign vs Adversarial PEs - trained

**Visualising Self-Organising-Maps for the different data types:**

A picture containing honeycomb

Description automatically generated

A picture containing honeycomb

Description automatically generated

Sub-Fig 1 SOM: Malicious

Sub-Fig 4 SOM: Adversarial (After Training - 50 Epochs)

Sub-Fig 2 SOM: Benign

Sub-Fig 3 SOM: Adversarial

**Analysis of the Self-Organising Maps or SOMs:**

To support our observations further, we dig deeper with the help of Self-Organising Maps or SOMs so as obtain a 2-d visualisation of the feature space.

Firstly, we look at the SOMs for Malicious and Benign samples which suggests a difference in their feature space. This is quite understandable given that these samples have different structures representing their malicious or benign nature respectively.

Secondly, we investigate the two SOMs generated from the Adversarial samples. The first adversarial SOM, generated using adversarial samples from an untrained network, closely resembles the structure of the malicious SOM as can be seen from the figure. This is in line with our inference from the frequency plots as well which was that the basic idea of generating an adversarial sample is to preserve malicious structure.

It is even more interesting to analyse the SOM for the second set of adversarial samples, i.e. the ones generated from a network trained for 50 epochs. It seems to have shifted into a different space as compared to the adversarial SOM suggesting that it is taking a different and possibly more robust approach towards engineering an adversarial sample so as to fool the defence.

**Task 2:**

**A) Code Changes that we made.**

Code provided in the *grams()* and topkplus functions in inner\_maximizer.py have been added by us. Additionally some code was modified in framework.py to obtain data samples of benign malicious and adversarial and to calculate run times.

InnerMaximizer.py-:

def grams(x, y, model, loss\_fct, k=8):

# Basically just gets x... initialize

#Cuda-Proofing the code

if next(model.parameters()).is\_cuda:

x = x.cuda()

y = y.cuda()

y = Variable(y)

x\_next = get\_x0(x, False)

# x\_orig = get\_x0(x, False)

while k > 0.5:

x\_var = Variable(x\_next, requires\_grad=True)

y\_model = model(x\_var)

loss = loss\_fct(y\_model, y)

grad = torch.autograd.grad(loss.mean(), x\_var)

grad = grad[0].data - x \* grad[0].data

sign = torch.gt(grad, 0).float()

# grad = torch.abs(grad[0].data - x \* grad[0].data)

grad = (sign - x\_next) \* grad

sign = sign + x

topk\_vals, indices = grad.topk(k= min(1024, max(1, k)), dim=1)

x\_next.scatter\_(dim=1, index=indices, src=sign.gather(dim=1, index=indices))

loss\_o = loss\_fct(model(x\_var), y).data

loss\_prime = loss\_fct(model(Variable(x\_next)), y).data

# print("Natural loss (%.4f) vs Adversarial loss (%.4f), Difference: (%.4f)" %

# (loss\_o.mean(), loss\_prime.mean(), loss\_prime.mean() - loss\_o.mean()))

flag = 0

replace\_flag = (loss\_prime < loss\_o).unsqueeze(1).expand\_as(x\_next)

x\_next[replace\_flag] = x[replace\_flag]

if (loss\_prime < loss\_o).sum() and k < 1024:

# x\_var = x.data

k = 2 \* k

else:

k = k//2

if x\_next.is\_cuda:

x\_next = x\_next.cpu()

return x\_next

def topkplus(x, y, model, loss\_fct, k=8):

xx = grams(x, y, model, loss\_fct, k=8)

no\_improve =0

if next(model.parameters()).is\_cuda:

xx = xx.cuda()

y = y.cuda()

y = Variable(y)

best\_x = xx

best\_x = Variable(best\_x)

while no\_improve<10:

x\_var = Variable(xx, requires\_grad=True)

y\_model = model(x\_var)

loss = loss\_fct(y\_model, y)

grad = torch.autograd.grad(loss.mean(), x\_var)[0].data.cuda()

sign = torch.gt(grad, 0).float()

topk\_vals, indices = grad.topk(k= random.randrange(1,20), dim=1)

xx.scatter\_(dim=1, index=indices, src=sign.gather(dim=1, index=indices).float())

x\_var=(xx)

x\_var = Variable(x\_var)

loss\_o = loss\_fct(model(x\_var), y).data

loss\_prime = loss\_fct(model(best\_x), y).data

replace\_flag = (loss\_prime < loss\_o).unsqueeze(1).expand\_as(x\_var)

rows=torch.sum(replace\_flag)

if rows!=0:

no\_improve=0

else:

no\_improve+=1

best\_x[replace\_flag] = x\_var[replace\_flag]

return grams(best\_x.data, y.data, model, loss\_fct, k=8)

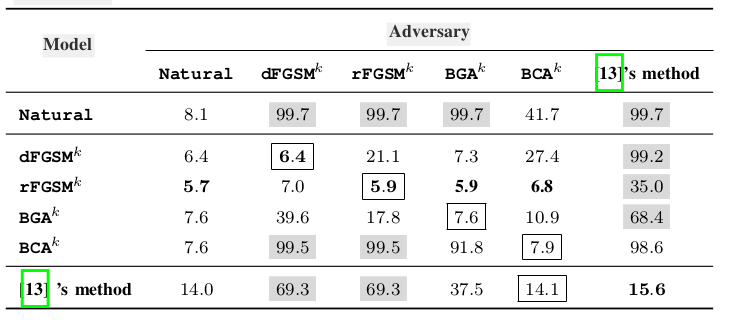
Framework.py-

B) Run time and evasion rate analysis-

We propose the GRAMS algorithm to generate adversarial samples and ran some comparative tests on it as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | *Grams* | | |
| **accuracy** | **f1** | **evasion** |
| *Natural* | 100 | 0.8 | 32 |
| *Untrained* | 100 | 0.8 | 12.7 |
| *dfgsm\_k* | 100 | 0.9 | 14.9 |
| *rfgsm\_k* | 100 | 1 | 6.2 |
| *grams* | 100 | 1 | 5.6 |
| *bga\_k* | 100 | 1 | 6.8 |

|  |  |
| --- | --- |
| **Method** | **Run-time**  (seconds) |
| *Natural* | 0.78 |
| *dfgsm\_k* | 9.02 |
| *rfgsm* | 8.68 |
| *bga\_k* | 14.25 |
| *Grams* | 1.81 |

  
Illustration 1: Baseline of existing methods

Run-time using *average processing time* for batches:-

Run-time is an important aspect of adversarial networks as it directly relates to the model being robust to new adversarial samples and its ability to thwart them during defence. Correspondingly it translates to being able to generate more obscure adversarial samples during attack. We observed that our GRAMS implementation shows shorter batch processing time than the baseline methods like the *top\_k* implementations like the *fgsm* variants and *bga\_k*. Admittedly, the natural (naïve) implementation is faster.

Analysing performance (Evasion rate, Accuracy and f1-score):-

Concerning the performance of the topk/grams algorithm in terms of evasion rate and f1-score, we can see that the evasion rate against the natural model is the best which is as expected as it’s not trained with any defense. However the evasion rates against the other defended models are not as good, with dfgsm\_k being the best defended model against which the topk attack can be used. The other defended models such as bga and rfgsm\_k are all below 7 percent evasion which isn’t as good as the other baseline methods such as BGA. However when looking at the tradeoff between both runtime and evasion rate, topk is quite a good choice.