.DIEM

Progetto per il corso di Artificial Intelligence for Cybersecurity

Realizzazione di un sistema di malware detection

Gruppo 6

Luigi Ferraioli 0622701853

Dario Picone 0622701750

Mario Petagna 0622701757

Federica Mazzone 0622701836

Indice

Descrizione del progetto

Traccia 1 – Realizzazione di un sistema di malware detection

Realizzare un sistema di malware detection robusto utilizzando metodologie di machine learning e deep learning.

Formuliamo il problema dell'analisi e del rilevamento del malware come un problema di classificazione binaria in cui malware e benigno sono le due classi target.

Il lavoro si sviluppa come segue :

- *Sezione I* → Preprocessing dei dataset
- *Sezione II* → Approcci scelti e funzionamento
- *Sezione III* → Addestramento e validazione dei tre metodi sul dataset 1 e valutazione dell'accuratezza
- *Sezione IV* → Valutazione capacità di generalizzazione dei tre metodi sul dataset 2
- *Sezione V* → Generazione di campioni offuscati
- *Sezione VI* → Valutazione della robustezza del sistema ai campioni offuscati

1. Preprocessing del dataset

1.1 Dataset

Lo sviluppo del progetto prevede l'uso di dataset composti da file malware e benigni.

In particolare analizzeremo le seguenti classi di malware:

- *Adware*. Malware progettato per generare automaticamente pubblicità online. Questo tipo di malware genera entrate per il suo sviluppatore visualizzando annunci pubblicitari sull'interfaccia utente o sullo schermo.
- *Crypto miner*. Malware progettato per criptare le risorse informatiche del bersaglio per estrarre criptovalute come bitcoin
- *Downloader*. Lo scopo di un programma downloader è scaricare e installare ulteriori programmi dannosi.
- *Dropper*. È un programma dannoso progettato per fornire altri malware ai dispositivi di una vittima.
- *File infector*. È un tipo di malware che infetta i file eseguibili con l'intento di causare danni permanenti o renderli inutilizzabili.
- *Flooder*. È un trojan che consente a un utente malintenzionato di inviare enormi quantità di dati a un obiettivo specifico per inondare di messaggi "spazzatura" (privi di senso) altri canali di rete.
- *Installer*. È un trojan che consente a un utente malintenzionato di infettare dispositivi tramite la diffusione di finti installer.
- *Packed*. Malware offuscati tramite tecnica di impacchettamento.
- *Ransomware*. Software dannoso che limita l'accesso degli utenti al sistema informatico crittografando i file o bloccando il sistema mentre richiede un riscatto per il suo rilascio.
- *Spyware*. Software per computer che spia e raccoglie informazioni sensibili senza autorizzazione dal computer di una vittima. Gli esempi includono keylogger, password graver e sniffer.
- *Worm*. Un tipo di virus che sfrutta le vulnerabilità del sistema operativo per diffondersi. La principale differenza tra worm e virus è la capacità dei worm di auto-replicarsi e diffondersi in modo indipendente mentre i virus dipendono dall'attività umana.

Sono stati messi a disposizione due dataset, a partire dai quali apportando opportunamente alcune modifiche si è arrivati ad ottenere la seguente divisione:

- *Train:* Contiene 300 Malware per ognuno dei tipi precedentemente analizzati mentre i Benigni sono 1988, rappresentando quindi il 37.6%

- *Test:* Ci sono 97 Malware per ognuno dei tipi precedentemente analizzati e i Benigni sono 800, rappresentando quindi il 42.8%

Il dataset ottenuto è volontariamente sbilanciato verso i malware. Tale scelta scaturisce dal voler garantire la possibilità di rilevare una quantità di falsi positivi maggiore rispetto a quella di falsi negativi; per il problema in questione, infatti, la rilevazione di un falso negativo ha impatto ben più grave della rilevazione di un falso positivo.

È stato necessario quindi identificare, estrarre e selezionare le features più significative per ogni elemento appartenente alla cartella di train.

Sono stati usati a questo scopo 2 diversi script, MC&IB_FeatureExtraction.py per la preparazione degli eseguibili relativamente agli approcci MalConv ed Image Based e RF_FeatureExtraction.py per l'estrazione delle features relativamente all'approccio Random Forest.

In particolare, al termine dell'esecuzione dei file sopracitati si ottiene quanto segue:

- ***Random Forest***: Non essendo possibile operare con una rappresentazione diretta byte-based dell'eseguibile, quindi agnostica rispetto alla struttura del file eseguibile, si è scelto di estrarre le features dei file. L'estrazione delle features si sviluppa in più fasi in modo da ottenere risultati più accurati.
  Inizialmente è stato necessario estrarre le features tramite EMBER ottenendo quindi un primo file .csv per ogni dataset, contenente più di 2000 features per eseguibile appartenenti alle seguenti classi:
  - ByteHistogram
  - ByteEntropyHistogram
  - StringsInfo
  - GeneralFileInfo
  - HeaderFileInfo
  - SectionInfo
  - ImportsInfo
  - ExportsInfo

  A fronte di una fase preliminare di train utilizzando diversi iper parametri e fissando il numero massimo di features al valore 'sqrt', in modo da utilizzare più features possibili, è stata valutata l'importanza delle features estratte e quindi è stato creato un secondo file .csv contenente la lista delle coppie feature-importanza. A questo punto sono state eliminate tutte le features con importanza pari a 0. Questa fase di eliminazione si è resa necessaria per poter ottenere a valle della fase di training delle prestazioni migliori, evitando di guardare a features fondamentalmente 'inutili'.

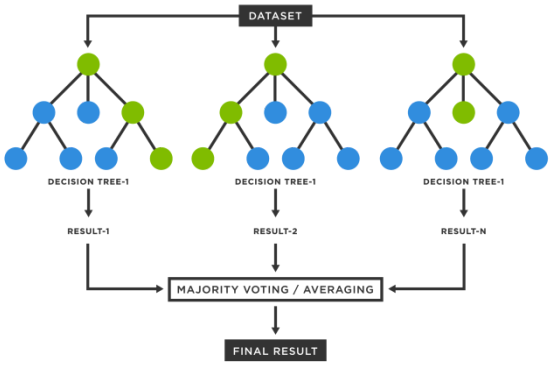
- **Image Based:** Preso in input il file binario, il suo dump esadecimale viene trasformato in un array di interi. L'array viene riorganizzato in una matrice e, infine, ciascun intero viene interpretato come un pixel in un'immagine in scala di grigi a 8 bit.
  Seguono poi operazioni di riscalamento.

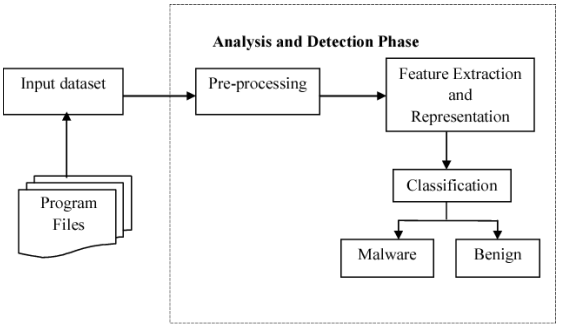
- **MalConv:** Si è scelto di operare con una rappresentazione diretta byte-based dell'eseguibile. In questo caso l'uso di tale rappresentazione consente semplicità, immediatezza e garantisce la possibilità di guardare all'intero malware senza effettuare preprocessing dei dati. Di contro la grandezza degli eseguibili può richiedere tempi di addestramento più lunghi ed un elevato utilizzo della memoria.
  Sulla base degli eseguibili a disposizione si è deciso di creare un file .csv costituito dai seguenti campi:

  - Nome dell'eseguibile
  - Label:
    - se l'eseguibile è maligno 1
    - se l'eseguibile è benigno 0

2. Approcci al problema

2.1 Random Forest





Per *Random Forest* si intende un algoritmo di machine learning supervisionato. Una random forest è un ensemble di alberi decisionali, generalmente addestrati tramite *sampling with replacement*, per i quali si può scegliere se usare o meno un unico algoritmo di addestramento. In fase di inferenza è necessario considerare le predizioni di tutti gli alberi e combinare le singole risposte. Il meccanismo di voting è basato su maggioranza e restituisce in output la predizione finale della Random Forest.
I modelli Random Forest presentano numerosi vantaggi rispetto ad altri algoritmi soprattutto grazie alla loro accuratezza, semplicità e flessibilità. Inoltre è doveroso considerare il fatto che, tale modello, possa essere usato per compiti di classificazione e regressione di natura non lineare, caratteristiche che di fatto lo rendono estremamente versatile. La teoria alla base del loro sviluppo è quella di utilizzare un insieme di classificatori non particolarmente specializzati sul problema da affrontare, ma che combinati insieme consentano di ottenere un'accuratezza elevata, il che si traduce nell'avere un gran numero di alberi non correlati capaci di creare previsioni più accurate di un singolo albero decisionale altamente specializzato.

```
SAMPLE_SPLIT = 2
MAX_FEATURES = "log2"
NUM_ESTIMATORS = 100
MAX_DEPTH = 3

# Data loading
train_set = load_my_dataset("/home/dario/AI4C/RF/Datasets/features_d_train.csv")
test_set = load_my_dataset("/home/dario/AI4C/RF/Datasets/features_d_test.csv")

X_train = train_set.data
X_test = test_set.data
y_train = train_set.target
y_test = test_set.target
feature_names = train_set.feature_names

# Creating a random forest
rf = RandomForestClassifier(n_estimators=NUM_ESTIMATORS, max_depth=MAX_DEPTH,
                            max_features=MAX_FEATURES, min_samples_split=SAMPLE_SPLIT)
```

Inizialmente sono stati definiti i parametri e la configurazione per il modello, è stato quindi istanziato il modello prendendo in considerazione l'implementazione della libreria scikit-learn in Python:

- ***n_estimators***: specifica il numero di alberi presenti nella foresta.

  È stato impostato a 100 per cercare il compromesso tra accuratezza e resistenza all'overfitting.

- ***max_depth***: specifica il numero di livelli massimo consentito in un albero decisionale. Se non specificato, l'albero decisionale continuerà a dividersi fino a raggiungere la purezza.

  Il valore dipende dal problema e dai dati, nel caso in esame si è scelta max_depth pari a 3.

- ***max_features***: specifica il numero di features da considerare per ogni split durante l'addestramento di ogni albero della foresta.

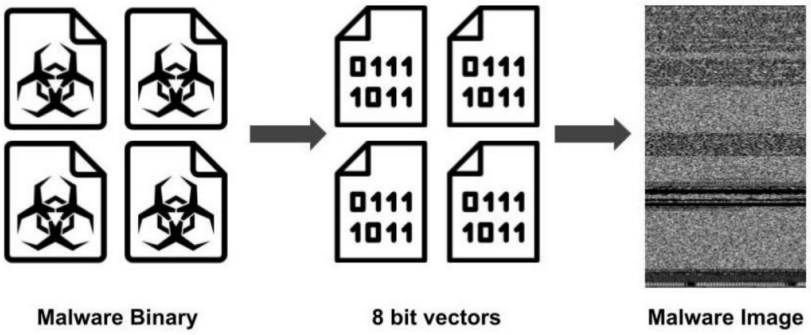
  Il valore per questo parametro varia in fase di training, poiché le feature con importanza pari zero sono state scartate in precedenza, resta solo un sottoinsieme più piccolo di feature rilevanti.
  Noto che il Random Forest lavora considerando parte delle features date in input, si è scelto di utilizzare l'operatore logaritmo, max_features=log2(n_features), che, garantendo l'uso di tutte le features più importanti, consente di aumentare la capacità di generalizzare diminuendo l'overfitting sulla rete, potendo quindi incrementare il numero di alberi usati.

- ***min_samples_split***: stabilisce il numero minimo di campioni richiesti per dividere un nodo interno.

  Lasciato al valore di default 2.

Il caricamento dei dati da dare in input al modello è operato tramite la funzione load_my_dataset implementata nel file train.py che legge i file .csv creati in fase di estrazione e li passa al modello in un formato accettato.

2.2 Image Based



La tecnica Image-based prevede di analizzare un eseguibile malware ed interpretare una sequenza di byte come sequenza di pixel in scala di grigi. Le immagini create sono passate ad una rete neurale che si occupa quindi di eseguire la classificazione vera e propria.

Partiamo dall'ipotesi che a diverse tipologie di malware corrispondano diverse tipologie di immagini. Usando molteplici dataset e provando a convertire famiglie di malware, si può notare che nella stessa famiglia c'è una sorta di omogeneità delle caratteristiche sull'immagine, questa può essere utile per distinguere più famiglie o un benigno da un malware.

Nonostante la scarsa correlazione che sussiste tra la rappresentazione binaria di un software e la sua rappresentazione per immagini, la rete riesce a identificare, con una buona accuratezza, la presenza di omogeneità usate per distinguere le caratteristiche nelle varie famiglie di malware.

```
LABELS = ['malware', 'benign']
input_path = "/Users/mario/OneDrive/Desktop/Magistrale 2° Anno - 1° Semestre/Progetto AI4C/immagini/"
VALIDAITON_SIZE = 0.1
BATCH_SIZE = 32
SEED = 42

data_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor()
])

train_ds = datasets.ImageFolder(input_path + 'train', data_transforms)
val_ds = datasets.ImageFolder(input_path + 'test', data_transforms)

dataloaders = {
    'train': torch.utils.data.DataLoader(train_ds,
                                         batch_size=BATCH_SIZE, shuffle=True),
    'validation': torch.utils.data.DataLoader(val_ds,
                                              batch_size=BATCH_SIZE, shuffle=True)
}

# Selecting the device on apple silicon
device = "mps" if torch.backends.mps.is_available() else "cpu"
print(f"Using device: {device}")

model = models.resnet50(weights=ResNet50_Weights.DEFAULT).to(device)

# Replacing the last fully-connected layer
model.fc = nn.Sequential(
    nn.Linear(2048, 128),
    nn.ReLU(inplace=True),
    nn.Linear(128, 1),
    nn.Sigmoid()).to(device)

criterion = nn.BCELoss()
optimizer = optim.Adam(model.fc.parameters())
```

Si tratta di un approccio piuttosto semplice: si trasforma il binario in un'immagine, si sceglie una rete addestrata, si effettua transfer learning e si usa la rete per classificare i malware oppure per distinguere tra benigni e malware.

La rete usata per il transfer learning è una CNN. Si è scelto di utilizzare come modello la rete ResNet-50, una rete neurale convoluzionale composta da 152 layer; la chiave per allenare un

così alto numero di livelli è l'uso di skip connection, il segnale (dati elaborati) trasferito al livello successivo è aggiunto anche all'output di quelli precedenti.

Dato che le *labels* sono 2: malware e benign, è stata creata una cartella "immagini" ed al suo interno sono state inserite due cartelle "malware" e "benign" le quali rispettivamente contengono le immagini dei malware e dei benigni del dataset 1.
Il modulo '*transforms*' fa parte della libreria Torchvision e fornisce una raccolta di operazioni predefinite per la trasformazione delle immagini. Nel caso in esame sono state usate le operazioni di ridimensionamento dell'immagine e di conversione a tensore della stessa.
Dopodiché, sono stati creati 2 DataLoaders: uno per il train set ed uno per il validation set.
L'API DataLoader consente di dividere i dati in mini-batch, mescolarli e applicare eventuali trasformazioni all'immagine. Il DataLoader utilizza la classe "*torch.utils.data.DataLoader*", che accetta come input un dataset e restituisce i dati in forma di batch.

Sono state apportate delle modifiche all'ultimo layer fully-connected, tale strato è stato infatti sostituito da un sotto-modello creato tramite il modulo Sequential.
L'input del sotto-modello è inizialmente passato al modulo `Linear(2048, 128)` che applica una trasformazione lineare ai dati in ingresso. L'uscita del modulo è passata poi come input al primo modulo `ReLU` così da applicare la funzione rectified linear unit in base all'elemento. L'output del primo modulo "ReLU" diventa l'input per il `Linear(128, 1)` i cui risultati sono infine l'input per il modulo `Sigmoid`.

Inizialmente sono stati definiti i parametri e la configurazione per il modello, è stato quindi istanziato il modello prendendo in considerazione l'implementazione della libreria Torchvision:

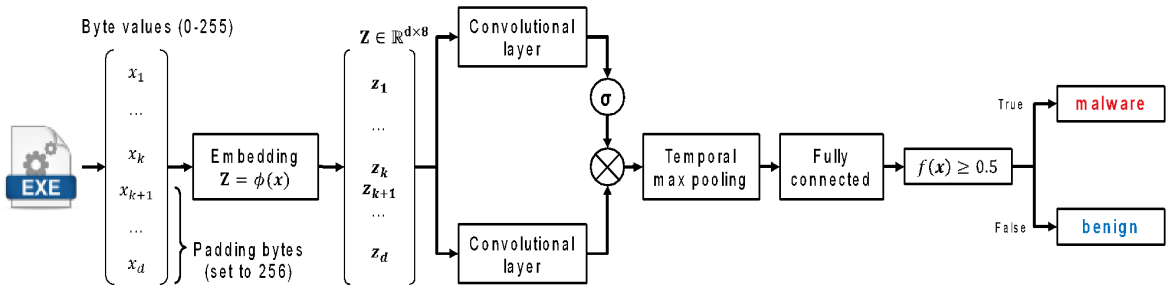
- ***batch_size:*** è un iper parametro che definisce il numero di campioni su cui lavorare prima di aggiornare i parametri del modello.

  Il valore è stato fissato a 32.

- ***weights:*** rappresenta i pesi pre allenati da usare per la rete, in particolare ResNet-50 usa i pesi seguenti.

acc@1 (on ImageNet-1K)	76.13
acc@5 (on ImageNet-1K)	92.862
min_size	height=1, width=1
categories	tench, goldfish, great white shark, … (997 omitted)
num_params	25557032
GFLOPS	4.09
File size	97.8 MB

2.3 MalConv



È stata la prima architettura proposta come rete convoluzionale per l'analisi diretta di eseguibili. Usa tecniche che analizzano l'eseguibile senza effettuare nessun tipo di estrazione delle feature ma direttamente come una sequenza di byte.

MalConv presenta la necessità di avere una input size fissa. Ipotizza infatti di lavorare con malware della dimensione massima di 2MB, questo significa che con file di dimensioni maggiori vengono analizzati solo i primi 2MB, mentre con file di dimensioni minori si aggiunge un padding tale da raggiungere la size desiderata.

Inizialmente la rete apprende un embedding, ad ogni byte corrisponde un vettore di 8 feature che rappresenta l'embedding del byte stesso all'interno di uno spazio vettoriale. La rappresentazione così ottenuta viene passata a due livelli convoluzionali monodimensionali e successivamente ad un livello di gating che presenta una funzione di attivazione ed una funzione di aggregazione. L'output del livello di gating viene passato ad un modulo di max pooling, e poi ad un livello fully connected. Infine si ha la funzione di classificazione finale che presenta una soglia fissa a 0.5, in questo modo se il valore in uscita dal layer fully connected è maggiore o uguale al valore soglia allora la rete classifica l'eseguibile come malware, altrimenti come benigno.

```
class MalConv(nn.Module):
    def __init__(self,input_length=2000000,window_size=500):
        super(MalConv, self).__init__()

        self.embed = nn.Embedding(257, 8, padding_idx=0)

        self.conv_1 = nn.Conv1d(4, 128, window_size, stride=window_size, bias=True)
        self.conv_2 = nn.Conv1d(4, 128, window_size, stride=window_size, bias=True)

        self.pooling = nn.MaxPool1d(int(input_length/window_size))

        self.fc_1 = nn.Linear(128,128)
        self.fc_2 = nn.Linear(128,1)

        self.sigmoid = nn.Sigmoid()
        #self.softmax = nn.Softmax()


    def forward(self,x):
        x = self.embed(x)
        # Channel first
        x = torch.transpose(x,-1,-2)

        cnn_value = self.conv_1(x.narrow(-2, 0, 4))
        gating_weight = self.sigmoid(self.conv_2(x.narrow(-2, 4, 4)))

        x = cnn_value * gating_weight
        x = self.pooling(x)

        x = x.view(-1,128)
        x = self.fc_1(x)
        x = self.fc_2(x)
        #x = self.sigmoid(x)

        return x
```

La cartella MalConv contiene tutti i file necessari alla definizione e addestramento del modello, in particolare:

- ***Cartella config:*** contiene il file example.yaml in cui vi sono le configurazioni dei parametri e degli iper parametri da usare per la costruzione del modello e l'addestramento della rete.

  I path e parametri usati sono i seguenti:

```yaml
### Data path
train_data_path: '/home/semanos/Scrivania/MALCONV/data/train/'                  # Training data
train_label_path: '/home/semanos/Scrivania/MALCONV/data/features_train.csv'     # Training label
valid_data_path: '/home/semanos/Scrivania/MALCONV/data/test/'                   # Validation Data
valid_label_path: '/home/semanos/Scrivania/MALCONV/data/features_test.csv'      # Validation Label

### output path
log_dir: '/home/semanos/Scrivania/MALCONV/log/'
pred_dir: '/home/semanos/Scrivania/MALCONV/pred/'
checkpoint_dir: '/home/semanos/Scrivania/MALCONV/checkpoint/'

### Parameter
use_gpu: False             #
use_cpu: 2                # Number of cores to use for data loader
display_step: 2           # Std output update rate during training
test_step: 10             # Test per n step
learning_rate: 0.01     #
max_step: 140          # Number of steps to train
batch_size: 20            #
first_n_byte: 2000000     # First N bytes of a PE file as the input of MalConv (defualt: 2 million)
window_size: 500          # Kernel size & stride for Malconv (defualt : 500)
sample_cnt: 1             # Number of data sampled for trianing (defualt 1 = all)
```

- ***Cartella data:*** presenta il dataset diviso in train e validation. Sono disponibili due file .csv, contenenti il nome di ciascun file all'interno delle cartelle train e validation con la label associata (0 benigno/ 1 maligno). La presenza di questi due file excel si rende necessaria poiché l'analisi del dataset, durante la fase di train, fa uso di un file .csv.

- ***Cartelle log, pred e checkpoint:*** le cartelle contengono i file di log, i risultati delle predizioni e i checkpoint del modello ottenuti a seguito della fase di train.

- ***Cartella src:*** nella cartella src è fornito il modello della rete e il file util.py, quest'ultimo estrae i sample dal dataset per trasformarli in tensori da poter utilizzare in fase di training.

- ***File train.py:*** analizza e manipola i file .csv precedentemente costruiti per i file appartenenti al dataset. Configura la rete a partire dai parametri specificati nel file example.yaml ed esegue poi la fase di training e validazione per il modello.

```python
malconv = MalConv(input_length=first_n_byte,window_size=window_size)
bce_loss = nn.BCEWithLogitsLoss()
model_dict = malconv.state_dict()
pretrained_dict = torch.load("/home/semanos/Scrivania/malconvGCT_nocat.checkpoint",
                             map_location=torch.device('cpu'))['model_state_dict']
#Modifica la dimensione di fc_2.weight
pretrained_dict['fc_2.weight'] = pretrained_dict['fc_2.weight'][:1,:]

# Modifica la dimensione di fc_2.bias
pretrained_dict['fc_2.bias'] = pretrained_dict['fc_2.bias'][:1]
pretrained_dict = {k: v for k, v in pretrained_dict.items() if k in model_dict}
model_dict.update(pretrained_dict)
malconv.load_state_dict(model_dict)
adam_optim = optim.Adam([{'params':malconv.parameters()}],lr=learning_rate)
sigmoid = nn.Sigmoid()
```

Per istanziare la rete è stata utilizzata una rete pre-addestrata, in modo che il modello MalConv creato partisse già con dei pesi pre-allenati.

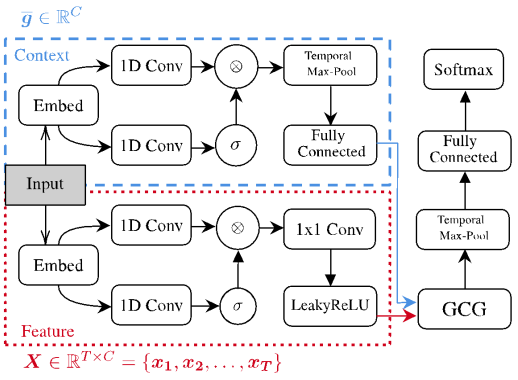
Per effettuare il caricamento del modello pre-allenato è stato necessario adattare i pesi del checkpoint poiché appartenenti ad un modello con una struttura diversa. È stato necessario far corrispondere la struttura "fc_2" del modello MalConv utilizzato a quella del modello checkpoint; sono state quindi modificate le dimensioni di fc_2.weight e fc_2.bias nel checkpoint per farle corrispondere alle dimensioni nell'architettura usata del modello.

I valori per i parametri usati sono tutti specificati all'interno del file di configurazione example.yaml precedentemente citato.

```
dataloader = DataLoader(ExeDataset(list(tr_table.index), train_data_path, list(tr_table.ground_truth), first_n_byte),
                        batch_size=batch_size, shuffle=True, num_workers=use_cpu)
validloader = DataLoader(ExeDataset(list(val_table.index), valid_data_path, list(val_table.ground_truth), first_n_byte),
                         batch_size=batch_size, shuffle=False, num_workers=use_cpu)
```

Tramite API DataLoader è stato completato il caricamento dei dati di train e test.

2.3.1 MalConv GCG (Global Channel Gating)



Rappresenta un tentativo di ottimizzare la rete, con l'obiettivo di estrarre due tipologie di informazioni, quindi di ampliare ancora di più il perceptive field e cercare di estrarre oltre ad una rappresentazione anche delle informazioni di contesto dei byte.

In questo caso, viene effettuata una duplicazione del modello MalConv. In figura, la sottorete blu, presenta la stessa struttura della rete MalConv originale e mira ad estrarre le informazioni di contesto; quella rossa, invece, viene utilizzata per estrarre un'ulteriore rappresentazione. In quest'ultima, è possibile notare la sostituzione del livello di temporal max-pool con un ulteriore livello convoluzionale il quale presenta una funzione di attivazione LeakyReLU. L'aggiunta di questo livello è un tentativo di andare a ridurre l'aggregazione effettuata dal livello di max-pool per ottenere un altro tipo di rappresentazione. L'output di queste due sottoreti va in ingresso ad un livello di gating chiamato Global Channel Gating (GCG)[1]. Dopodiché, si ha un altro livello di temporal max-pool e infine si ha la parte finale della struttura MalConv originale, quindi un layer fully connected e un layer con una activation function di tipo Softmax.

L'aggiunta del livello di gating GCG ha l'obiettivo di introdurre un *modulo di attenzione*. Lo scopo dei moduli di attenzione è quello di ottenere un'ulteriore serie di pesi che consentano alla rete di trovare le zone del file più rilevanti per classificarlo come malware o benigno.

[1] Raff, Edward and Fleshman, William and Zak, Richard and Anderson, Hyrum and Filar, Bobby and Mclean, Mark, The Thirty-Fifth AAAI Conference on Artificial Intelligence, Classifying Sequences of Extreme Length with Constant Memory Applied to Malware Detection, 2021,https://arxiv.org/abs/2012.09390

Di fatto questa struttura nasce con l'obiettivo di aggiungere anche l'attenzione alla rete MalConv ed inoltre, sulla base di alcune prove fatte, permette in teoria di aumentare la capacità della rete di generalizzare.

```
class MalConvGCT(LowMemConvBase):

    def __init__(self, out_size=2, channels=128, window_size=512, stride=512, layers=1,
                 embd_size=8, log_stride=None, low_mem=True):
        super(MalConvGCT, self).__init__()
        self.low_mem = low_mem
        self.embd = nn.Embedding(257, embd_size, padding_idx=0)
        if not log_stride is None:
            stride = 2**log_stride

        self.context_net = MalConvML(out_size=channels, channels=channels, window_size=window_size,
                                     stride=stride, layers=layers, embd_size=embd_size)
        self.convs = nn.ModuleList([nn.Conv1d(embd_size, channels*2, window_size, stride=stride, bias=True)] +
                                   [nn.Conv1d(channels, channels*2, window_size, stride=1, bias=True)
                                    for i in range(layers-1)])
        self.linear_atn = nn.ModuleList([nn.Linear(channels, channels) for i in range(layers)])

        # one-by-one cons to perform information sharing
        self.convs_share = nn.ModuleList([nn.Conv1d(channels, channels, 1, bias=True) for i in range(layers)])

        self.fc_1 = nn.Linear(channels, channels)
        self.fc_2 = nn.Linear(channels, out_size)
```

```
    # Over-write the determinRF call to use the base context_net to determinRF.
    # We should have the same total RF, and this will simplify logic significantly.
    def determinRF(self):
        return self.context_net.determinRF()

    def processRange(self, x, gct=None):
        if gct is None:
            raise Exception("No Global Context Given")

        x = self.embd(x)
        x = x.permute(0, 2, 1)

        for conv_glu, linear_cntx, conv_share in zip(self.convs, self.linear_atn, self.convs_share):
            x = F.glu(conv_glu(x), dim=1)
            x = F.leaky_relu(conv_share(x))
            x_len = x.shape[2]
            B = x.shape[0]
            C = x.shape[1]

            sqrt_dim = np.sqrt(x.shape[1])
            # we are going to need a version of GCT with a time dimension,
            # which we will adapt as needed to the right length
            ctnx = torch.tanh(linear_cntx(gct))
            # Size is (B, C), but we need (B, C, 1) to use as a 1d conv filter
            ctnx = torch.unsqueeze(ctnx, dim=2)
            # Roll the batches into the channels
            x_tmp = x.view(1, B * C, -1)
            # Now we can apply a conv with B groups,
            # so that each batch gets its own context applied only to what was needed
            x_tmp = F.conv1d(x_tmp, ctnx, groups=B)
            # x_tmp will have a shape of (1, B, L), now we just need to re-order the data back to (B, 1, L)
            x_gates = x_tmp.view(B, 1, -1)
            # Now we effectively apply σ(x_t^T tanh(W c))
            gates = torch.sigmoid(x_gates)
            x = x * gates
        return x
```

```
    def forward(self, x):
        if self.low_mem:
            global_context = CheckpointFunction.apply(self.context_net.seq2fix, 1, x)
        else:
            global_context = self.context_net.seq2fix(x)

        post_conv = x = self.seq2fix(x, pr_args={'gct': global_context})

        penult = x = F.leaky_relu(self.fc_1(x))
        x = self.fc_2(x)

        return x, penult, post_conv
```

La cartella MalConv_CGC contiene tutti i file necessari alla definizione e addestramento del modello ed è strutturata allo stesso modo della cartella MalConv.

- ***Cartella config:*** contiene il file example.yaml in cui vi sono le configurazioni dei parametri e degli iperparametri da usare per la costruzione del modello e l'addestramento della rete.

  I path e parametri usati sono i seguenti:

```yaml
exp_name: 'malconv_net'

### Data path
train_data_path: '/home/semanos/Scrivania/MALCONV FINALE/data/train/'                  # Training data
train_label_path: '/home/semanos/Scrivania/MALCONV FINALE/data/features_train.csv' # Training label
valid_data_path: '/home/semanos/Scrivania/MALCONV FINALE/data/test/'                   # Validation Data
valid_label_path: '/home/semanos/Scrivania/MALCONV FINALE/data/features_test.csv' # Validation Label

### output path
log_dir: '/home/semanos/Scrivania/MALCONV FINALE/log/'
pred_dir: '/home/semanos/Scrivania/MALCONV FINALE/pred/'
checkpoint_dir: '/home/semanos/Scrivania/MALCONV FINALE/checkpoint/'

### Parameter
use_gpu: False             #
use_cpu: 2               # Number of cores to use for data loader
display_step: 2          # Std output update rate during training
test_step: 20            # Test per n step
learning_rate: 0.01     #
max_step: 120        # Number of steps to train
batch_size: 20           #
first_n_byte: 2000000    # First N bytes of a PE file as the input of MalConv (defualt: 2 million)
window_size: 500         # Kernel size & stride for Malconv (defualt : 500)
sample_cnt: 1            # Number of data sampled for training (defualt 1 = all)
```

- ***Cartella data:*** presenta il dataset diviso in train e validation. Sono disponibili due file csv, contenenti il nome di ciascun file all'interno delle cartelle train e validation con la label associata (0 benigno/ 1 maligno). La presenza di questi due file excel si rende necessaria poiché l' analisi del dataset, durante la fase di train, fa uso di un file csv.

- ***Cartella log, pred e checkpoint:*** le cartelle contengono i file di log, i risultati delle predizioni e i checkpoint del modello ottenuti a seguito della fase di train.

- ***Cartella src:*** nella cartella src è fornito il modello della rete ed una serie di file di servizio:

  - *util.py*, estrae i sample dal dataset per trasformarli in tensori da poter utilizzare in fase di training.

  - *MalConvML.py*, contiene un approccio sperimentale alternativo all'addestramento con più livelli.

  - *checkpoint.py*, contiene il codice utilizzato per eseguire il checkpoint del gradiente per ridurre l'utilizzo della memoria.

  - *LowMemConv.py*, è la classe base che le implementazioni estendono per ottenere un pool di memoria fissa. Quest'ultimo è fornito dalla funzione seq2fix, che applica la convoluzione in blocchi, traccia i vincitori e quindi raggruppa le fette vincenti da eseguire con i calcoli del gradiente attivi. Per un uso adeguato si estende LowMemConvBase, implementando la funzione processRange, che applica qualsiasi strategia convoluzionale si desideri ad un intervallo di byte. La funzione determinRF viene utilizzata per determinare la dimensione del campo ricettivo testando in modo iterativo la dimensione di input più piccola che non genera errori, in modo da sapere come gestire le dimensioni dei blocchi in un secondo momento.

- ***File train.py:*** analizza e manipola i file .csv precedentemente costruiti per i file appartenenti al dataset. Configura la rete a partire dai parametri specificati nel file example.yaml ed esegue poi la fase di training e validazione per il modello.

```
malconv = MalConvGCT(out_size=1, channels=256, window_size=256, stride=64)
bce_loss = nn.BCEWithLogitsLoss()
model_dict = malconv.state_dict()
pretrained_dict = torch.load("/home/semanos/Scrivania/malconvGCT_nocat.checkpoint", map_location=torch.device('cpu'))[
    'model_state_dict']
# Modifica la dimensione di fc_2.weight
pretrained_dict['fc_2.weight'] = pretrained_dict['fc_2.weight'][:1, :]

# Modifica la dimensione di fc_2.bias
pretrained_dict['fc_2.bias'] = pretrained_dict['fc_2.bias'][:1]
pretrained_dict = {k: v for k, v in pretrained_dict.items() if k in model_dict}
model_dict.update(pretrained_dict)
malconv.load_state_dict(model_dict)
adam_optim = optim.Adam([{'params': malconv.parameters()}], lr=learning_rate, weight_decay=1e-5)
sigmoid = nn.Sigmoid()
```

La fase di preparazione del modello si svolge in maniera analoga a quella di MaCconv (paragrafo 2.3), in questo caso è importante notare che nella definizione della rete è stato usato il parametro *out_size* con valore 1, tale valore è necessario per far si che l'ultimo livello della rete restituisca in output la classificazione sulle due sole classi definite. Il modello MalConvCGC presenta per implementazione nativa out_size=2 che di fatto determina in output una multi classificazione. Si considera inoltre l'aggiunta dell'iper parametro "weight_decay", che permette di implementare una regolarizzazione di tipo L2, per contrastare il fenomeno dell'overfitting.

3. Addestramento e validazione dei tre metodi sul dataset 1 e valutazione dell'accuratezza

In questa sezione sono analizzate le configurazioni dei tre modelli scelti per la fase di training, soffermandosi sulle scelte fatte e sulle loro motivazioni.

3.1 Random Forest

```
X_train = train_set.data
X_test = test_set.data
y_train = train_set.target
y_test = test_set.target
feature_names = train_set.feature_names

# Creating a random forest
rf = RandomForestClassifier(n_estimators=NUM_ESTIMATORS, max_depth=MAX_DEPTH,
                            max_features=MAX_FEATURES, min_samples_split=SAMPLE_SPLIT)
# Training the forest
classifier = rf.fit(X_train, y_train)

# Testing
y_pred = classifier.predict(X_test)
```

Si istanzia il modello Random Forest tramite i parametri discussi nel paragrafo 2.1, si esegue il training della rete tramite la funzione fit, il modello restituito è infine usato per la fase di validazione.

A valle della fase di training sono stati ottenuti i seguenti valori:

	Precision	Recall	F1-score	Support
0	0,99	0,75	0,85	796
1	0,84	0,99	0,91	1067
accuracy			0,89	1863
macro avg	0,92	0,87	0,88	1863
weighted avg	0,9	0,89	0,89	1863

Accuracy score: 0,889

3.2 Image Based

```
def train_model(model, criterion, optimizer, num_epochs):
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch + 1, num_epochs))
        print('-' * 10)

        for phase in ['train', 'validation']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            count = 0
            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device).float()
                labels = torch.unsqueeze(labels, -1)
                outputs = model(inputs).float()
                loss = criterion(outputs, labels)

                if phase == 'train':
                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()

                preds = torch.round(outputs)
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
                count += 1
                torch.save(model.state_dict(), '/Users/mario/OneDrive/Desktop/resnet50.h5' + str(count) + ' ' + phase)

            epoch_loss = running_loss / (count * BATCH_SIZE)
            epoch_acc = running_corrects.float() / (count * BATCH_SIZE)

            print('{} loss: {:.4f}, acc: {:.4f}'.format(phase,
                                                        epoch_loss,
                                                        epoch_acc))
    return model
```

La funzione *train_model(model, criterion, optimizer, num_epochs)* si occupa di addestrare il modello scelto secondo i parametri in input
Poiché si sta analizzando un problema di classificazione binaria, si è scelta come criterion la BCELoss, che misura l'entropia incrociata binaria tra il target e l'output; si è scelto Adam come optimizer.

Per quanto riguarda il numero di epoche è stato scelto il valore 10 al fine di evitare che la rete andasse in overfitting. È stata inserita l'istruzione torch.save all'interno del processo di training in modo tale da salvare il checkpoint del modello per ogni epoca e poterlo utilizzare per operazioni successive.

Per ogni epoca sono riportati i valori di: Train loss, Train accuracy, Validation loss, Validation accuracy così da valutare le performance della rete. A valle della fase di training per il modello si sono raggiunti i seguenti risultati:

Epoch	Train loss	Accuracy	Validation loss	Accuracy
Epoch 1	0.2823	0.8727	0.2823	0.8591
Epoch 2	0.2649	0.8793	0.2693	0.8708
Epoch 3	0.2439	0.8904	0.2723	0.8702
Epoch 4	0.2154	0.9057	0.2761	0.8671
Epoch 5	0.2147	0.9076	0.2605	0.8750
Epoch 6	0.1948	0.9160	0.2556	0.8824
Epoch 7	0.1867	0.9175	0.2874	0.8724
Epoch 8	0.1780	0.9230	0.2548	0.8776
Epoch 9	0.1693	0.9292	0.2527	0.8755
Epoch 10	0.1638	0.9292	0.2439	0.8840

3.3 MalConv

```
valid_best_acc = 0.0
total_step = 0
step_cost_time = 0

while total_step < max_step:

    # Training
    malconv.train()
    for step, batch_data in enumerate(dataloader):
        start = time.time()

        adam_optim.zero_grad()
        cur_batch_size = batch_data[0].size(0)

        exe_input = batch_data[0].cuda() if use_gpu else batch_data[0]
        exe_input = Variable(exe_input.long(), requires_grad=False)
        label = batch_data[1].cuda() if use_gpu else batch_data[1]
        label = Variable(label.float(), requires_grad=False)

        pred = malconv(exe_input)
        loss = bce_loss(pred, label)
        loss.backward()
        adam_optim.step()

        history['tr_loss'].append(loss.cpu().data.numpy())
        history['tr_acc'].extend(
            list(label.cpu().data.numpy().astype(int) == (sigmoid(pred).cpu().data.numpy() + 0.5).astype(int)))

        step_cost_time = time.time() - start

        if (step + 1) % display_step == 0:
            print(step_msg.format(total_step, np.mean(history['tr_loss']),
                                  np.mean(history['tr_acc']), step_cost_time), end='\r', flush=True)
        total_step += 1

        # Interruzione per effettuare la validazione: ogni test_step epoche viene fatta una fase di validazione
        if total_step % test_step == 0:
            break
```

In fase di training sono presi in considerazione i valori di max_step e test_step che definiscono rispettivamente il numero di epoche per l'allenamento e il 'passo' per determinare l'inizio della fase di validazione.
Nel caso in esame max_step è stato fissato a 200 epoche e test_step a 20 epoche, ciò determina l'esecuzione di una fase di validazione ogni 20 epoche. Entrambi i valori sono definiti nel file example.yaml.

Sono stati definiti i parametri per la funzione di loss e l'optimizer, in particolare è stata scelta la loss function BCEWithLogitsLoss la quale combina uno strato Sigmoid e il BCELoss in un'unica classe. Questa versione è numericamente più stabile rispetto all'utilizzo di un semplice Sigmoid seguito da un BCELoss poiché, combinando le operazioni in un unico livello, si sfrutta il trucco log-sum-exp per la stabilità numerica; è stato scelto invece come optimizer Adam.

I dati sono caricati tramite DataLoader, le feature sono date quindi in input alla rete e a seguito della predizione viene calcolata l'accuracy, successivamente salvata in un file di log.

```
# Testing
history['val_loss'] = []
history['val_acc'] = []
history['val_pred'] = []
malconv.eval()
for _, val_batch_data in enumerate(validloader):
    cur_batch_size = val_batch_data[0].size(0)

    exe_input = val_batch_data[0].cuda() if use_gpu else val_batch_data[0]
    exe_input = Variable(exe_input.long(), requires_grad=False)

    label = val_batch_data[1].cuda() if use_gpu else val_batch_data[1]
    label = Variable(label.float(), requires_grad=False)

    pred = malconv(exe_input)
    loss = bce_loss(pred, label)

    history['val_loss'].append(loss.cpu().data.numpy())  # qui era
    # history['val_loss'].append(loss.cpu().data.numpy()[0])
    history['val_acc'].extend(
        list(label.cpu().data.numpy().astype(int) == (sigmoid(pred).cpu().data.numpy() + 0.5).astype(int)))
    history['val_pred'].append(list(sigmoid(pred).cpu().data.numpy()))

print(log_msg.format(total_step, np.mean(history['tr_loss']), np.mean(history['tr_acc']),
                     np.mean(history['val_loss']), np.mean(history['val_acc']), step_cost_time),
      file=log, flush=True)

print(valid_msg.format(total_step, np.mean(history['tr_loss']), np.mean(history['tr_acc']),
                       np.mean(history['val_loss']), np.mean(history['val_acc'])))
if valid_best_acc < np.mean(history['val_acc']):
    valid_best_acc = np.mean(history['val_acc'])
    torch.save(malconv, chkpt_acc_path)
    print('Checkpoint saved at', chkpt_acc_path)
    write_pred(history['val_pred'], valid_idx, pred_path)
    print('Prediction saved at', pred_path)

history['tr_loss'] = []
history['tr_acc'] = []
```

Allo stesso modo è gestita la fase di validazione passando in input alla rete le feature relative al dataset di test.

Al termine delle fasi di training e validazione, vengono salvati il modello e i dati, questi ultimi sono raccolti all'interno della cartella log e riportati di seguito:

Step	Tr_loss	Tr_acc	Val_loss	Val_acc
20	1.064878	0.6175	0.495178	0.7113
40	0.487672	0.7950	0.444577	0.7804
60	0.319985	0.8300	0.386018	0.7949
80	0.512803	0.8250	0.920236	0.6540
100	1.023090	0.7975	2.678043	0.7306
120	1.011024	0.7650	0.729588	0.7622
140	0.590195	0.8425	0.565578	0.7290
160	0.422493	0.8525	0.456699	0.8377
180	0.341454	0.9025	0.373487	0.8527
200	0.296896	0.8950	0.422154	0.8715

3.3.1 MalConv GCG (Global Channel Gating)

Anche in questo caso in fase di training sono presi in considerazione i valori di max_step e test_step, definiti in precedenza.
Per il modello MalConvGCG entrambi i valori sono ancora una volta definiti nel file example.yaml, in particolare il parametro max_step è stato fissato a 120 epoche e test_step a 20 epoche.

La fase di training e validazione è la medesima del modello MalConv definita al paragrafo 3.3.

Step	Train loss	Accuracy	Validation loss	Accuracy
20	2.965111	0.6025	2.050912	0.6460
40	0.775370	0.7250	0.681204	0.7418
60	0.919112	0.7375	0.737824	0.8141
80	0.665751	0.7225	0.429146	0.7815
100	0.399555	0.8100	0.332726	0.8490
120	0.284934	0.8825	0.334264	0.8602

4. Valutazione capacità di generalizzazione dei tre metodi sul dataset 2

Generalmente una rete addestrata sulla base dei campioni appartenenti al training set deve essere poi in grado di generalizzare, ossia di dare la risposta corretta in relazione a ingressi non considerati nell'insieme di addestramento.

In questa fase sono stati caricati i modelli ottenuti a valle di train e validation e a questi è stato sottoposto il dataset 2, composto da circa 1000 malware, in modo da poter valutare la capacità di generalizzazione delle reti.

La capacità di generalizzare, utilizzando un'analisi statica, si può valutare provando a capire se, dato un malware, si riesce a identificare su campioni nuovi una logica simile che conduca a un comportamento malevolo. Tale analisi risulta utile quando malware già presenti nel training set si evolvono nel tempo o sono riproposti in contesti diversi e quindi nuovi dataset presentano un malware noto ma con forme diverse.

Segue un'analisi in dettaglio dei risultati ottenuti per i singoli modelli.

4.1 Random Forest

```
#Test the generalization of the net on a dataset of validation

generalization_set = load_my_dataset("/home/dario/AI4C/RF/Datasets/features_d_validation.csv")
X_generalization = generalization_set.data
y_generalization = generalization_set.target

forest = load("/home/dario/AI4C/RF/Models/RF.joblib")

y_pred = forest.predict(X_generalization)
acc = accuracy_score(y_generalization, y_pred)
print("Generalization performance ->", acc)
```

La fase di generalizzazione conduce ad un risultato per l'accuracy pari a 0,797.

Tali risultati sono possibili grazie alla fase di preprocessing, analizzata nel sottoparagrafo 2.1, il modello Random Forest non ha la necessità di analizzare l'intero malware, e quindi l'intera logica, ma solo dati rilevanti riuscendo a raggiungere un'elevata capacità di generalizzazione.

4.2 Image Based

```
PIL.Image.MAX_IMAGE_PIXELS = 93312000000000
LABELS = ['malware']
input_path = "/Users/mario/OneDrive/Desktop/Magistrale 2° Anno - 1° Semestre/Progetto AI4C/dataset2/"
BATCH_SIZE = 32

data_transforms = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor()
    ])

test_ds = datasets.ImageFolder(input_path, data_transforms)

dataloaders = {
    'test': torch.utils.data.DataLoader(test_ds, batch_size=BATCH_SIZE, shuffle=False)}

device = "mps" if torch.backends.mps.is_available() else "cpu"
model = models.resnet50(weights=ResNet50_Weights.DEFAULT).to(device)
model.fc = nn.Sequential(
               nn.Linear(2048, 128),
               nn.ReLU(inplace=True),
               nn.Linear(128, 1),
               nn.Sigmoid()).to(device)
model.load_state_dict(torch.load('/Users/mario/OneDrive/Desktop/resnet50.h5',
                                 map_location=torch.device('cpu')), strict=False)
criterion = nn.BCELoss()
optimizer = optim.Adam(model.fc.parameters())

count = 0
running_loss = 0.0
running_corrects = 0
model.eval()

for inputs, labels in dataloaders['test']:
    inputs = inputs.to(device)
    labels = labels.to(device).float()
    labels = torch.unsqueeze(labels, -1)
    outputs = model(inputs).float()
    loss = criterion(outputs, labels)

preds = torch.round(outputs)
running_loss += loss.item() * inputs.size(0)
running_corrects += torch.sum(preds == labels.data)
count+=1

epoch_loss = running_loss / (count*BATCH_SIZE)
epoch_acc = running_corrects.float() / (count*BATCH_SIZE)

print('loss: {:.4f}, acc: {:.4f}'.format(epoch_loss, epoch_acc))
```

Loss: 0.5352	Accuracy: 0.6768

Otteniamo un livello di generalizzazione non ottimale.

I risultati potrebbero essere imputabili ad una fase di preprocessing che applica una resize abbastanza stringente.
La resize sulle immagini può, infatti, provocare perdita di informazioni. In tale situazione si va a penalizzare la fase di allenamento della rete, che, avendo meno feature su cui lavorare, riesce a specializzarsi solo su un determinato feature set risultando meno efficace nell'analisi di nuovi malware.

4.3 Malconv

```
history['val_loss'] = []
history['val_acc'] = []
history['val_pred'] = []
malconv.eval()
for _, val_batch_data in enumerate(validloader):
    cur_batch_size = val_batch_data[0].size(0)

    exe_input = val_batch_data[0].cuda() if use_gpu else val_batch_data[0]
    exe_input = Variable(exe_input.long(), requires_grad=False)

    label = val_batch_data[1].cuda() if use_gpu else val_batch_data[1]
    label = Variable(label.float(), requires_grad=False)

    pred = malconv(exe_input)
    loss = bce_loss(pred, label)

    history['val_loss'].append(loss.cpu().data.numpy())  # qui era
    # history['val_loss'].append(loss.cpu().data.numpy()[0])
    history['val_acc'].extend(
        list(label.cpu().data.numpy().astype(int) == (sigmoid(pred).cpu().data.numpy() + 0.5).astype(int)))
    history['val_pred'].append(list(sigmoid(pred).cpu().data.numpy()))

print(log_msg.format(total_step, np.mean(history['tr_loss']), np.mean(history['tr_acc']),
                     np.mean(history['val_loss']), np.mean(history['val_acc']), step_cost_time),
      file=log, flush=True)

print(valid_msg.format(total_step, np.mean(history['tr_loss']), np.mean(history['tr_acc']),
                       np.mean(history['val_loss']), np.mean(history['val_acc'])))
if valid_best_acc < np.mean(history['val_acc']):
    valid_best_acc = np.mean(history['val_acc'])
    torch.save(malconv, chkpt_acc_path)
    print('Checkpoint saved at', chkpt_acc_path)
    write_pred(history['val_pred'], valid_idx, pred_path)
    print('Prediction saved at', pred_path)

history['tr_loss'] = []
history['tr_acc'] = []
```

In fase di generalizzazione per il modello MalConv è stato possibile osservare valori di accuracy sul dataset 2 estremamente bassi.

Tale osservazione ci ha condotto a riconsiderare la fase di train, sono stati quindi definiti due casi possibili:

- *Caso 1:* si considera un numero di epoche pari a 200, il train in questo caso è quello presentato nel paragrafo 3.3.

  Ricordiamo che con questa configurazione l'accuratezza su training e validation set risultava essere rispettivamente di circa 0.89 e 0.87. In particolare in fase di generalizzazione sono stati riscontrati valori di accuratezza estremamente bassi come 0.30, segnale di un overfitting sulla rete.

- *Caso 2:* si considera un numero di epoche ridotto pari a 140, l'aggiunta di un iperparametro "weight_decay", per contrastare il fenomeno dell'overfitting e la diminuzione del parametro test_step a 10 epoche. In questo modo ogni 10 epoche di allenamento verrà effettuata una fase di validazione.

```
malconv = MalConv(input_length=first_n_byte,window_size=window_size)
bce_loss = nn.BCEWithLogitsLoss()
model_dict = malconv.state_dict()
pretrained_dict = torch.load("/home/semanos/Scrivania/malconvGCT_nocat.checkpoint",
                             map_location=torch.device('cpu'))['model_state_dict']
#Modifica la dimensione di fc_2.weight
pretrained_dict['fc_2.weight'] = pretrained_dict['fc_2.weight'][:1,:]

# Modifica la dimensione di fc_2.bias
pretrained_dict['fc_2.bias'] = pretrained_dict['fc_2.bias'][:1]
pretrained_dict = {k: v for k, v in pretrained_dict.items() if k in model_dict}
model_dict.update(pretrained_dict)
malconv.load_state_dict(model_dict)
adam_optim = optim.Adam([{'params':malconv.parameters()}],lr=learning_rate, weight_decay=1e-5)
sigmoid = nn.Sigmoid()
```

Allenando la stessa rete con la nuova configurazione l'accuratezza sul training e validation set risulta essere rispettivamente di 0.8850 e 0.8404.

Step	Train loss	Accuracy	Validation loss	Accuracy
10	1.764283	0.5850	0.569222	0.7311
20	0.581211	0.6800	0.531758	0.7483
30	0.484369	0.7700	0.467928	0.7783
40	0.545010	0.7850	0.480329	0.7916
50	0.506385	0.7750	0.507708	0.7467
60	0.485614	0.8200	0.417042	0.8168
70	0.460907	0.7850	0.427888	0.8120
80	0.456399	0.8400	0.647686	0.7943
90	0.604936	0.7750	0.735114	0.7633
100	0.563371	0.8450	0.684881	0.7804
110	0.655221	0.8550	0.600982	0.7434
120	0.397608	0.8800	0.593771	0.7922
130	0.402616	0.8400	0.492722	0.8382
140	0.312804	0.8850	0.412587	0.8404

In fase di generalizzazione si osserva un valore di accuratezza di 0.5590, migliore rispetto quello ottenuto in precedenza.

Tale valore ottenuto ha portato a stabilire che la rete non riesce a generalizzare sul dataset 2.

I risultati ottenuti in fase di generalizzazione, inferiori a quanto sperato, potrebbero essere determinati dalla composizione del dataset utilizzato. Il dataset 1 risulta di fatto essere esiguo e la rete potrebbe non essere pronta ad analizzare malware non precedentemente valutati.

4.3.1 MalConv GCG

La fase di generalizzazione sul dataset 2 per la rete MalConv GCG conduce ad un risultato per l'accuracy pari a 0.5720.

Il risultato ottenuto risulta essere superiore a quello del caso precedente ma rimane in ogni modo un risultato insoddisfacente. Nonostante i risultati di prove sperimentali documentino la forte capacità di generalizzare per la rete MalConv GCG, in questo caso specifico non vale questa assunzione.

I risultati ottenuti ci hanno portato a stabilire che il dataset utilizzato e la fase di allenamento eseguita hanno portato la rete a specializzarsi prettamente sul dataset 1, non riuscendo a sviluppare capacità di generalizzare su dataset mai visti.

5. Generazione di campioni offuscati

5.1 GAMMA (Genetic Adversarial Machine learning Malware Attack)

Si tratta di un attacco black-box basato su algoritmi genetici. Gli algoritmi genetici imitano l'evoluzione della specie per giungere ad un obiettivo; partendo da genitori casuali, essi genereranno figli che, tramite crossover e mutazioni, giungeranno a produrre l'oggetto target. L'oggetto target è definito tramite una funzione (*fitness function*) che valuta l'idoneità degli individui all'interno della popolazione creata e la bontà dell'insieme delle soluzioni rispetto al problema in esame. Il processo continua fino a quando non è più possibile migliorare la funzione obiettivo o si è raggiunto il massimo numero di iterazioni possibile.

Trasponendo il problema alla tematica dell'offuscamento dei malware ci sono una serie di manipolazioni che è possibile fare su un eseguibile in modo da garantire la proprietà di *functionality preserving*.

Focalizzando l'attenzione sulle alterazioni statiche, trascurando quindi le alterazioni comportamentali più complesse da modellare, è necessario scegliere un insieme di alterazioni da considerare come popolazione. È scelto il tipo di alterazioni da attuare e l'impatto che queste devono avere.

La *fitness function* non solo misura quanto una alterazione abbia effetto ma anche l'impatto in termini di modifiche sul file originale, scegliendo infine le alterazioni ad impatto minimo.

La fitness function nel caso dell' attacco GAMMA può essere definita come segue:

$$\begin{aligned} \underset{s \in \mathcal{S}}{\text{minimize}} \quad & F(s) = f(x \oplus s) + \lambda \cdot \mathcal{C}(s), \\ \text{subject to} \quad & q \leq T. \end{aligned}$$

- $x \in X \subset \{0, \ldots, 255\}^*$ rappresenta il programma in input di tipo malware descritto come una stringa di byte di lunghezza arbitraria
- Si definisce un set di k manipolazioni distinte che conservino la funzionalità da poter applicare al programma di input x come vettore $s \in S \subset [0,1]^k$
  - Ciascun elemento di s corrisponde a una diversa manipolazione che può essere applicata al programma di input.
  - Le manipolazioni sono parametrizzate in [0, 1], per indicare la misura in cui vengono applicate.
  - Se assumiamo che l'i-esimo elemento s_i sia associato all'iniezione di una determinata sezione, allora è possibile dire che rappresenterà una variabile binaria che denota se l'iniezione avviene (si = 1) o meno (si = 0).
- La funzione $\oplus : X \times S \rightarrow X$ applica le manipolazioni descritte da s al programma di input x, preservandone la funzionalità, e restituisce il programma manipolato.
- $f : X \rightarrow R$ per indicare l'output del modello di classificazione sul programma di input. In questo caso f rappresenta l'output del modello sulla classe dannosa, ossia più è alto il valore di f, più x è considerato dannoso.

Notiamo che F(s) è costituito da termini contrastanti:

- $f(x \oplus s)$: l'output della classificazione sul programma manipolato.

- *C(s)*: una funzione di penalizzazione che valuta il numero di byte iniettati nel malware di input.

- $\lambda > 0$ regola il trade-off tra i termini, promuove soluzioni con un numero minore di byte iniettati C(s) a scapito di ridurre la probabilità che il campione sia erroneamente classificato come benigno.

La cartella GAMMA contiene i seguenti elementi:

- **Adv:** campioni offuscati e utilizzati per valutare la trasferibilità.
- **File GAMMA_LOG_E_TRASF.xlsx:** Il file è composto di due fogli relativi al log dell'attacco GAMMA su rete MalConv di SecML e ad un log parziale per i risultati dell'attacco sui modelli selezionati nel Capitolo 2, non sono specificati i filtri relativi alla configurazione finale usata.
- ***File Gamma_logging:***

```
def gamma_parm(log,penalty_regularizer,seed,iteration,population,threshold, sections_to_extract,counter):
    """
    This function executes the GAMMA attack and saves the value used in a log with the
    effectivness of the attack on the model attacked

    @param log the path to the log file
    @param penalty_regularizer the penalty regularizer used for the GAMMA attack
    @param seed the seed used for the RNG of the attack
    @param iteration the number of iteration of the genetic algorithm
    @param population the population of the genetic algorithm
    @param threshold the threshold that the model should reach
    @param sections_to_extract the sections of the benign to extract ad use for the algorithm
    @param counter the counter of the test

    @return resultpath the path where the adversarials samples are stored
    """

    adv_folder = '/home/dario/AI4C/RF/data/mini/adv'
    malware_folder = '/home/dario/AI4C/RF/data/mini/malware/'
    bening_folder = '/home/dario/AI4C/RF/data/benign'


    X = []
    y = []
    file_names = []

    # List of malware samples to be obfuscated
    for i, f in enumerate(os.listdir(malware_folder)):
        path = os.path.join(malware_folder, f)

        with open(path, "rb") as file_handle:
            raw = file_handle.read()

        x = CArray(np.frombuffer(raw, dtype=np.uint8)).atleast_2d()
        _, confidence = net.predict(x, True)

        #if not predicted as benign discard the sample
        if confidence[0, 1].item() < 0.5:
            continue
```

In primo luogo, è stata creata la rete tramite una classe modello CClassifierEnd2EndMalware, che generalizza i modelli ML end-to-end di PyTorch. Poiché MalConv è già codificato all'interno del plugin, anche i pesi vengono memorizzati e possono essere recuperati con il metodo load_pretrained_model. Quindi, si usa un wrapper CEnd2EndWrapperPhi, ovvero un'interfaccia che astrae dalla fase di estrazione delle caratteristiche del modello. Questo è necessario per le impostazioni black-box dell'attacco. Si carica un dataset da *'malware samples/test folder'* contenente i malware per testare gli attacchi e si scartano tutti i campioni che non vengono visti dalla rete come benigni.

Si crea, poi, la popolazione della sezione dai file contenuti in una cartella specificata, segue la definizione dell'attacco GAMMA di iniezione della sezione con i seguenti parametri:

- ***section_population***: elenco contenente tutte le sezioni goodware da iniettare.
- ***model_wrapper:*** il modello di destinazione, avvolto all'interno di un CWrapperPhi, in questo caso il modello attaccato è 'net'.

- ***population_size:*** la dimensione della popolazione generata ad ogni round dall'algoritmo genetico.
- **penalty_regularizer:** il parametro di regolarizzazione utilizzato per il vincolo di dimensione.
- ***iterations:*** il numero totale di iterazioni.
- ***seed:*** specifica un seme di inizializzazione per il random generator.
- ***threshold:*** la soglia di rilevazione.

In un primo momento sono stati considerati come campioni maligni tutti quelli presenti nel dataset 2 e come campioni benigni, da cui estrarre le sezioni, 60 random sample.
I parametri per l'attacco sono stati fissati nel seguente modo:

- section_population = [.rdata, .data]
- population_size= 10
- penalty_regularizer = da 0.1 a 100
- iterations = 50
- seed = None
- threshold = 0.5

L'uso di tale configurazione ha condotto a risultati fondamentalmente privi di significato, in particolare, data la natura estremamente randomica dell'algoritmo c'è la necessità di un gran numero di esecuzioni per avere risultati potenzialmente coerenti.
Inoltre, dato che all'aumentare di population_size, iterations e numero di sample aumenta corrispondentemente il tempo di esecuzione dell'attacco, compiere un numero elevato di esecuzioni risultava incompatibile con i tempi concessi per lo svolgimento del progetto.
A valle delle considerazioni precedenti, si è scelto di ridurre l'insieme dei malware selezionando solo 49 elementi su cui la rete (MalConv di SecML) opera sicuramente una giusta classificazione come malware. In seguito si è deciso di far variare più configurazioni di parametri al fine di ottenere una quantità di risultati elevata che 'abbattesse' la randomicità dell'algoritmo e consentisse di identificare i valori rilevanti per i parametri in uso.

A questo punto sono stati eseguiti 3 diversi esperimenti[2]:

Esperimento A
numero malware considerati = 49
section_population = [.rdata, .data], [.data], [.text, .data]
population_size= 3, 5
penalty_regularizer = da 10^-1 a 100
iterations = 10, 25, 50
seed = None
threshold =0, 0.5

Questo primo esperimento ha portato ad individuare un uso errato del penalty regularizer dato che la media dei risultati al variare del parametro risulta fondamentalmente invariata.
Gli unici parametri che conducevano a variazione del risultato erano quelli relativi alle sezioni.

Esperimento B
numero malware considerati = 49
section_population = [.rdata, .data], [.data], [.text, .data]
population_size= 3, 5
penalty_regularizer = da 10^-8 a 10^-5
iterations = 10, 25, 50
seed = None
threshold = 0

Sulla base dell'esperienza acquisita dall'esperimento precedente, si è deciso di utilizzare valori per il "penalty_regularizer" molto più bassi, per mettere in risalto eventuali effetti dell'attacco.
Inoltre per velocizzare le esecuzioni, mantenendo una quantità di risultati elevata, è stato modificato il valore del threshold fissandolo a 0.

[2] Traccia degli esperimenti è disponibile a questo link.

In questa configurazione si è notato che i parametri relativi alle sezioni, per quanto conducessero a valori in media differenti, variavano con la stessa intensità.
La modifica del "penalty_regularizer" ha condotto ai risultati sperati, ossia ad una coerenza teorica con le formule che descrivono l'attacco.

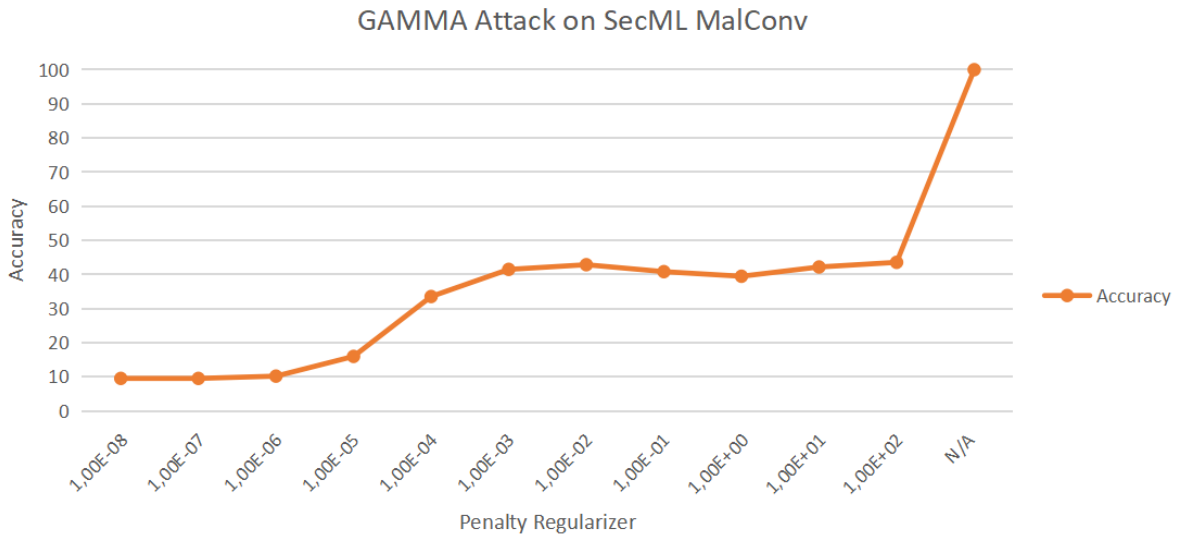
Esperimento C
numero malware considerati = 49
section_population = [.rdata, .data]
population_size= 3, 5
penalty_regularizer = da 10^-4 a 10^-2
iterations = 10, 25, 50
seed = None
threshold = 0

In ultima analisi, nota la correlazione fra le variazioni dei risultati sulle tre sezioni di popolazione, si è scelto di non considerare le sezioni [.data], [.text, .data] al fine di abbattere ulteriormente i tempi di esecuzione.
Infine, è stato modificato l'insieme dei valori da considerare per il "penalty_regularizer" in modo da coprire i valori nel range [10^-4, 10^-2], precedentemente non considerati.

I tre esperimenti hanno portato alla realizzazione di oltre 200 casi d'analisi e tutti i risultati ottenuti sono stati condensati in un unico file excel.
A partire da questo sono state calcolate le medie al variare del parametro "penalty_regularizer" mantenendo solo i risultati corrispondenti alla configurazione seguente:

numero malware considerati = 49
section_population = [.rdata, .data]
population_size= 3, 5
penalty_regularizer = da 10^-8 a 10^2
iterations = 10, 25, 50
seed = None
threshold = 0

In conclusione i risultati sono stati usati per tracciare una curva che mostri, sulla rete MalConv di SecML, l'andamento dell'accuracy al variare del penalty regularizer.



Possiamo notare come l'accuracy della rete migliori all'aumentare del penalty regularizer, portando l'attacco GAMMA ad essere sempre meno efficace. Tale risultato è in linea con la teoria poiché a valori bassi del penalty regularizer corrisponde una maggiore capacità di manipolare i file binari e di conseguenza un aumento della potenza dell'attacco.

La curva mostra un'irregolarità ovvero, una lieve crescita del valore dell'accuracy nell'intervallo [10^−3, 10^−2], imputabile al fatto che l'attacco potrebbe aver aggiunto un valore di rumore ai campioni tale da agevolare la rete conducendo ad una corretta classificazione dei malware.

L'ultimo valore della curva, identificato con N/A (Not Attacked), presenta accuracy del 100% poiché rappresenta l'accuracy sul dataset di partenza, indicando come la rete riesca a classificare correttamente come malware tutti i file nel dataset.

6. Valutazione della robustezza del sistema ai campioni offuscati

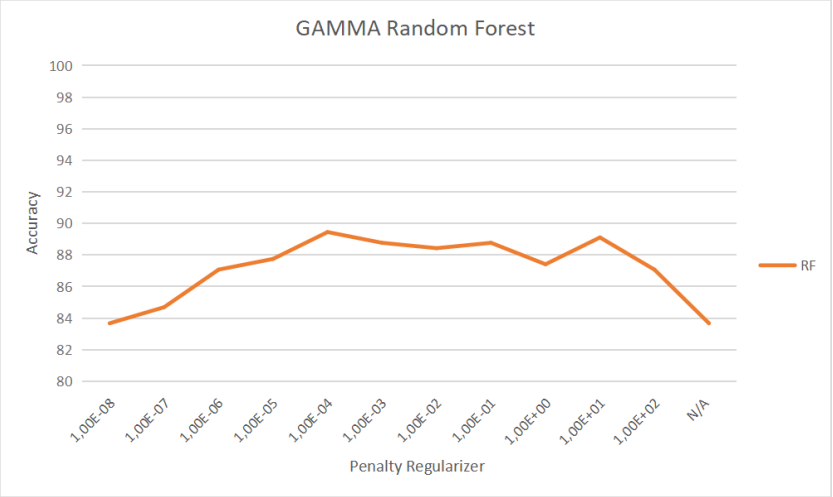
La trasferibilità degli attacchi è un concetto fondamentale nella sicurezza informatica e indica la capacità di un attacco, sferrato contro un modello di machine learning, di essere efficace contro un modello diverso, potenzialmente sconosciuto.

Questo concetto assume maggior importanza soprattutto nel caso zero-knowledge (ZK) o black-box attacks, dove l'attaccante anche senza avere conoscenza del sistema può riprodurne il comportamento e sfruttare quindi, la proprietà di trasferibilità degli attacchi.

Spesso un attaccante preferisce un attacco trasferibile su più classificatori ad uno efficace su un solo classificatore.

È stata analizzata la trasferibilità dell'attacco GAMMA, descritto nel capitolo 5, sui modelli presentati nel capitolo 2. Per ogni modello è stato creato un grafico che mostrasse l'andamento dell'accuracy al variare del penalty regularizer.

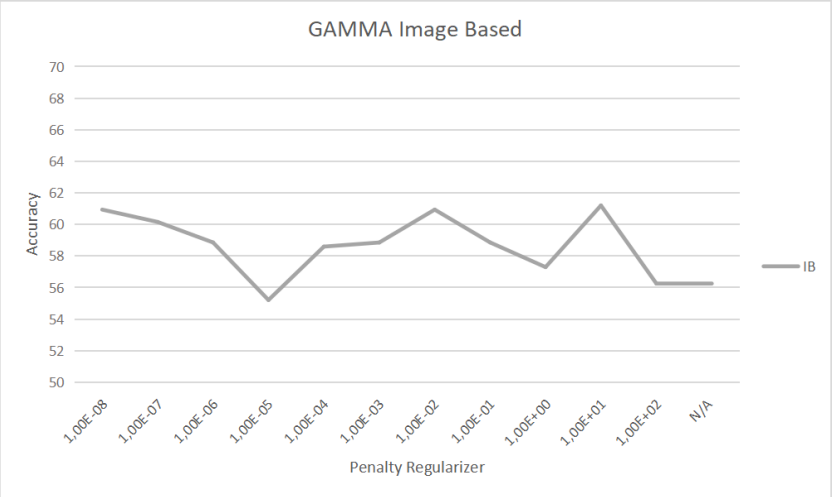
6.1 Random Forest



Come ci si aspettava l'attacco non è trasferibile su questo modello a causa delle grandi differenze fra il modello Random Forest e MalConv. Risulta rilevante la coincidenza dei punti di inizio e fine della curva, relativi al valore penalty regularizer 10^-8 e alla situazione Not Attacked.

Per valori bassi del penalty regularizer si ottiene una modifica dei binari elevata e consapevole, evitando la presenza di valori anomali e fuorviando la rete nella classificazione. La coincidenza, precedentemente evidenziata, potrebbe essere spiegata dal fatto che per valori di penalty regularizer elevati i binari subiscono variazioni piccole e casuali compromettendo l'estrazione delle features e causando la presenza di valori anomali, aiutando quindi la rete nella classificazione.

6.2 Image Based

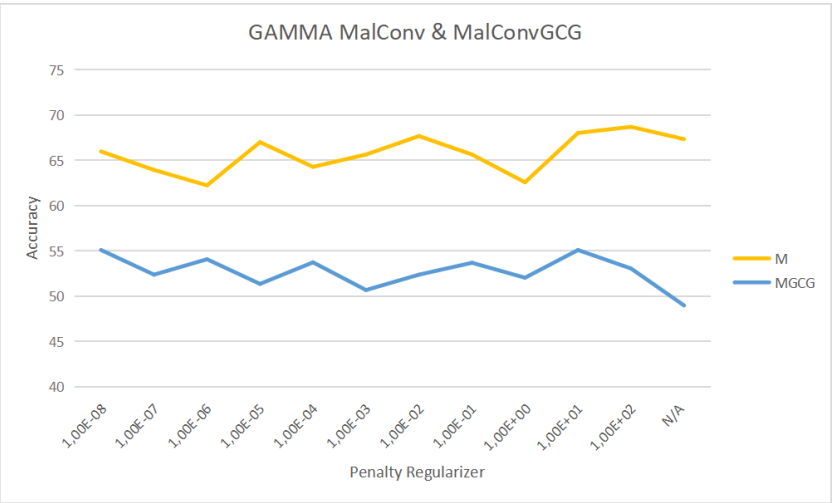


L'attacco non è trasferibile su questo modello.

Le variazioni dell'accuracy potrebbero essere determinate dalle variazioni all'interno dei file binari ma apparentemente risultano non correlate alla potenza dell'attacco.

Il risultato ottenuto è proprio quello atteso, dal momento in cui non ci sono similitudini tra la rete utilizzata per generare gli attacchi e l'implementazione Image Based.

6.3 MalConv & MalConv GCG



Per i modelli Malconv e MalconvGCG ci si aspettava, almeno in parte, la trasferibilità dell'attacco. Tuttavia, è importante sottolineare che tale proprietà non è garantita tra reti simili a causa di differenze intrinseche nella configurazione e nell'architettura che rendono la rete più sensibile ad attacchi specifici. Inoltre, il risultato potrebbe derivare dalle differenze nei dataset di addestramento usati dai sistemi.

Riferimenti

SecML Malware – https://secml-malware.readthedocs.io/en/docs/index.html

Scikit learn – sklearn.ensemble.RandomForestClassifier.html

Raff, Edward, et al. "Classifying sequences of extreme length with constant memory applied to malware detection." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 35. No. 11. 2021.

B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In 2018 26th European Signal Processing Conference (EUSIPCO), pages 533–537. IEEE, 2018.

Gibert, D., Mateu, C., & Planes, J. (2020). The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. Journal of Network and Computer Applications, 153, 102526.

Appunti dal corso di Artificial Intelligence for Cybersecurity – Anno Accademico 2022/2023