Seminars in Advanced Topics in Computer Science

Function Classification in Unstripped Binaries - University "La Sapienza"

Valerio Trenta, **1856471**

2020-10-09

1 Introduction and motivations

The goal of this paper is that of implementing some of the most well known machine learning techniques and neural networks to solve the **function classification** problem, that is detailed in paragraph two.
Classifying functions from unstripped or stripped binaries can be a hard but very useful task, and has several applications especially in the computer security field; indeed, whenever a security analyst is facing a binary sample to identify its features and functioning without having access to its source code, learning and understanding what exactly each of its functions does and how is usually a huge part of the strain, as well as, in some cases, one of the hardest parts. That's why recently, research has put quite the effort on the topic.
By classifying a function, we wish to understand its behavior - e.g. we could classify a function f as a cryptographic hash function or a simple getter depending on the operations it executes, and we would know depending on the output of the classification what these operations look like to our classifier.
We exploit the **UbuntuDataset** from the *In Nomine Function paper* [1] to extract two main features from each function in each binary of the dataset in paragraph three, and then proceed to cluster the functions in an unsupervised manner in paragraph four.
Having clustered the functions in several classes, we then exploit them to build several different neural networks to solve the **function classification** problem in the next three paragraphs.
Results and possible future improvements are reported in paragraph eight, as well as related works, papers and readings that either were cited applied to this work, or might be useful to improve the results and help solving the problem.

1.1 Problem definition

To provide a formal definition to our problem, keep in mind that it is a **classification problem**: given a set of k classes $\mathbf{C = \{c_1, ..., c_k\}}$ and a set of n functions in a programming language $\mathbf{X = \{x_1, ..., x_n\}}$, our goal is to learn a function $\mathbf{F: X \rightarrow C}$ so

that $F(x_i) = c_j$, for each i $\in$ {1...n} and j $\in$ {1...k}.

2 Dataset and features extraction

The **UbuntuDataset** is composed of thousands and thousands of compiled binaries, most of which stripped and some of them having their corresponding debug files, so we can *unstrip* them to recover their debug information - e.g. function names, variables names and much more - through the *eu-unstrip* utility.
We select a subset of these unstripped binaries, composed of 1080 of them, so that we can extract the two features we are mainly interested in: the **sequence of assembly instructions as simple text** and the **control flow graphs** - from now on, **CFGs** - of each function of each binary.
Detailing the extraction process and thus the dataset building process does not belong to the scope of this paper. However, these two features were extracted from each function by exploiting a Jython script that is run by the headless analyzer provided by **Ghidra** (see *extract.py*), which allows us to store in different JSON files the information about the two aforementioned main features of *120K different functions*.
Each file is composed of a several JSON strings representing the **networkx** object of the **CFG** of the function, with each node holding as data its corresponding **assembly instructions**. Describing the **CFG** is also out of the scope of this paper, but an accurate rendering of the possible content of one of these files is represented in **Figure 1**: this way, each function in our dataset is now clearly identified by its **CFG** and the instructions it carries.

Please, keep also in mind that the original **UbuntuDataset** is much bigger, and actually millions of functions are available. We are only exploiting a fraction of it, and the results will also reflect the poor size of the dataset. We should also point out that at this point the dataset provides no information about the class of each function, meaning the functions whose features were extracted are not labeled, thus we still cannot exploit a supervised approach to solve the problem.

3 Clustering

Since the labels for the functions in our dataset are lacking, by definition of the **function classification problem**, we cannot solve it. So, in order to tackle our problem, we need to define an *unsupervised approach*: we will exploit the *k-means++* algorithm to cluster the functions in **k** groups. The clusters we obtain will be later used again to solve the aforementioned problem, but we should stress the fact that these groups are *not to be considered* labels - at least, not yet: theoretically, the way we are building them grants us that each of these clusters contains functions that are somehow *similar* one with another, but in order to assign them a meaningful label and actually check whether this process was successful, each function in each group should be examined by a human expert.

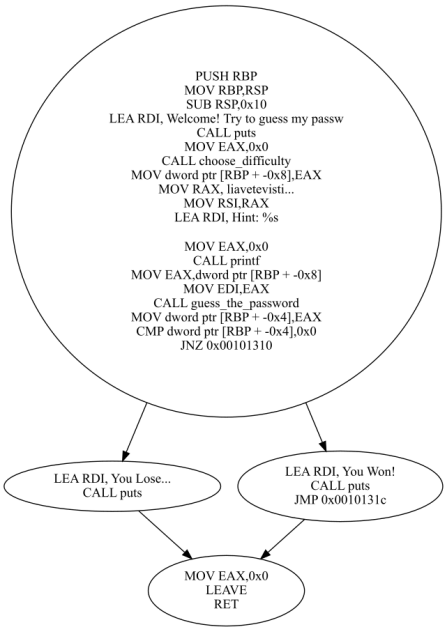

Figure 1: CFG with assembly instruction of a Main function.

This, and the fact that we are only exploiting a relatively small number of functions from our dataset, could negartively impact the results.

3.1 Function vectorization through Autoencoder

We need to turn each function into a single vector of dimension **d** in order to feed it to the **k-means++** algorithm. One can do this in several different ways, choosing one of the two features we have extracted. We choose to vectorize each function through the **assembly instructions** it contains - that is to say, *text*. We have tried two different and very well known vectorization (or encoding) approaches, *bag of words* and *tf-idf*, the latter being the most reliable and accurate one as expected.
We extract the list of assembly instructions from each function and apply the simplest form of preprocessing on them to build up a vocabulary by trimming each instruction to the first word - that is to say, the *x86 operation*, completely ignoring the remaining parts of the isntructions such as registers, offsets and other values. Please notice that this process surely leads to information loss: some of the instructions also contain strings or numeric values that may be useful and identify the function. For example, the instruction **MOV RAX, RBX** corresponds to **MOV** in our vocabulary and, as a matter of fact, the instruction **MOV RAX,** "**192.168.0.0**" will also correspond to the very same **MOV**.
Having a vocabulary made up of more than 400 terms, we now compute the *tf-idf* value for each instruction in the function, so that it is now represented as a vector of such values. Given our corpus, out of 120K vectorized functions, the one with the highest dimensionality is represented by a vector of length **l**, so we pad the remaining functions with zeros

to this value.
Now we apply a very simple and linear **autoencoder** to retrieve the *encodings* for each function, decreasing the length of the vectors to **d=100**. This is important, since the *k-means++* algorithm we are going to implement is known in literature to suffer from the *curse of dimensionality* and, the higher the dimension of the vectors, the harder it will be to cluster them (and the more the running time).
The autoencoder is composed obviously of two parts: an *encoder* abnd a *decoder*, both made of two *linear layers*. The input size of the encoder is the output size of the decoder and viceversa, since the output of tghe decoder should be the input itself. Each layer exploits a **ReLU** activation function and the model is trained with a **Mean Squared Error function loss** and an **Adam** optimizer - it's a pretty standard implementation, the **MSE** here is preferred since we are not classifying anything but rather we are interested in the encodings and how much they depart from the original vectors. As pictured in **Figure 2**, the loss is minimal, and thus we retrieve the vectors representing our functions with **d=100** and ready to be further processed.

```
AutoEncoder(
  (enc1): Linear(in_features=164, out_features=120, bias=True)
  (enc2): Linear(in_features=120, out_features=100, bias=True)
  (dec1): Linear(in_features=100, out_features=120, bias=True)
  (dec2): Linear(in_features=120, out_features=164, bias=True)
)
epoch : 1/20, recon loss = 0.01878214
epoch : 2/20, recon loss = 0.00382700
epoch : 3/20, recon loss = 0.00272125
epoch : 4/20, recon loss = 0.00270429
epoch : 5/20, recon loss = 0.00270030
epoch : 6/20, recon loss = 0.00269942
epoch : 7/20, recon loss = 0.00270018
epoch : 8/20, recon loss = 0.00269591
epoch : 9/20, recon loss = 0.00269616
epoch : 10/20, recon loss = 0.00269645
epoch : 11/20, recon loss = 0.00269973
epoch : 12/20, recon loss = 0.00269316
epoch : 13/20, recon loss = 0.00269485
epoch : 14/20, recon loss = 0.00269397
epoch : 15/20, recon loss = 0.00269603
epoch : 16/20, recon loss = 0.00269290
epoch : 17/20, recon loss = 0.00269493
epoch : 18/20, recon loss = 0.00269851
epoch : 19/20, recon loss = 0.00269294
epoch : 20/20, recon loss = 0.00269264
```

Figure 2: Loss during the Autoencoder training step.

3.2 Clustering through k-means++

Now that we have the encodings, we can apply *k-means++* to perform an unsupervised learning and clustering of the functions. We have, so far, a dataset containing functions represented as *"name of function"* $\rightarrow$ *"100-dimensional vector"*.
We provide this dataset to the algorithm as a matrix of **n*d** dimension, where **n** = number of functions in our dataset (that is, 120K) and **d** = dimension of the vectors representing the fucntions (that is, 100). This way, each row in the matrix corresponds to a different function, and each column of a specific row a value in the encoding of the function.
We don't and absolutely can't know a priori the value of **k**, that is to say exactly how many clusters we are going to retrieve. We try three different values: k = 5, k = 7 and

k = 10, whose reesults are pictured in **Table 1**, and we keep the results we obtain from **k=10**.

Table 1: k-means++ results

cluster	k=5	k=7	k=10
1	12337	43168	12334
2	26475	9423	7123
3	35891	10162	5883
4	38858	18500	1193
5	7278	12334	10162
6	//	26059	9423
7	//	1193	18500
8	//	//	14390
9	//	//	15774
10	//	//	26057

From the table, we can clearly see how the clustering process is carried out when **k** is increased: some clusters remain clearly the same between the three steps (e.g. clusters 1 in k = 5 and k = 10 and cluster 5 in k = 7, or cluster 3 and 4 in k = 7 and 5 and 7 in k = 10), while others are clearly split from one iteration to another.
We have now clustered the functions in our dataset in 10 different groups, each of them containing a certain number of functions that, given the assembly instructions they are made of, are similar one to another - that is to say, their vectors are close and thus assigned to the corresponding centroid in a 100-dimensional space.
This first result could make it possible for a human expert to actually assign a label to each of these clusters, classifying these functions - e.g., the third cluster could be almost entirely made of functions that perform cryptographic operations, while the last one could contain a fair amount of functions that perform network operations, and so on. Assigning such labels is, at the moment, out of the scope of this paper: what we want to do now is balancing this dataset so that we can train a neural network over it and make it possible to the network itself to assign a given function to one of the clusters we have retrieved.

4 Convolutional Neural Network over Adjacency Matrices

We have yet to exploit the second main feature we have extracted from the functions: the **CFGs**. In order to classify each function, given the clusters we have retrieved in the previous step, we need to balance our dataset first.
That is also why we chose to keep the results obtained with **k = 10** in the previous paragraph: these clusters are indeed easier to balance. In our case, we take clusters *0, 6, 7, 8* first and clusters *0, 7, 8, 9* after, and take only the first *12280* elements from them in both cases, so we have a training dataset of *43K* functions and a testing one of *5K* functions.

Now, we are ready to implement our neural network.

4.1 CFGs as adjacency matrices

We represent each function through its corresponding **Control Flow Graph**, and we need to *encode* it in such a way that its features and properties are correctly exhibited.
One of the most well known and simplest way to encode a graph, is to represent it through its **adjacency matrix**.
Given a graph **G** = (V, E) with |V| = **n**, its *adjacency matrix* **A** is an **nxn** matrix whose element $a_{i,j} = 1$ if and only if ∃ edge (i, j) ∈ E, otherwise $a_{i,j} = 0$. Given this property, adjacency matrices are usually sparse.
Representing our functions in the dataset in this way might be useful to handle the classification task through a **Convolutional Neural Network**, since it is known in literature that these neural networks can handle inputs such as matrices. We also encode another property of our graph, that is the *topological order* of its nodes: given **G** = (V, E) acyclic and directed graph, we perform a linear ordering of its nodes so that for every edge (i, j) ∈ E, node **i** comes before node **j** in the ordering. Given the fact that the *topological order* can be obtained only from graphs that are acyclic - and directed, but the **CFGs** we retrieved are already directed - we first traverse them and remove any directed cycle - loop - that we encounter, making them *directed acyclic graphs*. Once the graphs have been topologically ordered, by computing their adjacency matrices following this ordering we will retrieve such matrices having a composition that is very similar to the one of a diagonal matrix, having the non-zero elements located mostly on the first half of the columns for the very first rows, and on the second half as the matrix approaches the end of the rows, as pictured in **Figure 3**. We also notice that, usually, the last or last two rows of such matrices will be composed of all zeros - and this is likely due to the fact that they represent nodes in the graphs with no outgoing edges, because they are the nodes corresponding to the last bunch of instructions of each function.

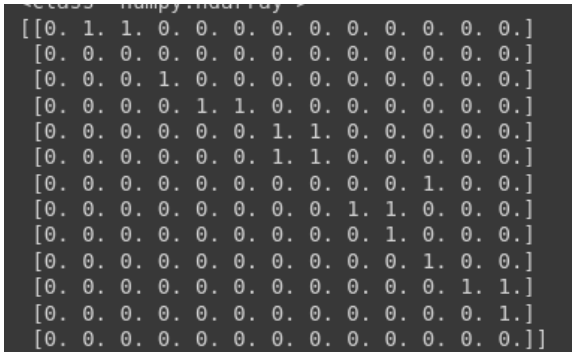

Figure 3: An adjacency matrix retrieved after performing a topological ordering process, and yet to be normalized.

4.2 Embedding the topologically ordered adjacency matrices

One particular point should be stressed out the most: the adjacency matrices come from CFGs of different size in terms of nodes and edges. This means that, in our dataset, we retrieve matrices ranging from dimension 1x1 (a single-element matrix) to 1536x1536.
In order to feed them as input to a CNN, we need to normalize these matrices - meaning, we need to have all the matrices of same dimension. Unfortunately, this means either padding all of them to the maximum dimension - which is impossible for computational reasons, since storing more than 120K matrices of 1536x1536 is infeasible on our machine - or choosing a dimension **dxd** $<$ 1536x1536 and pad some matrices and litterally cut the others.
To avoid losing too much information, we can perform a *principal components analysis* on the matrices exceeding the **dxd** limit, but we still have to pad with zeros the other ones. We are basically performing *normalization* over the adjacency matrices we retrieved.
We set $\mathbf{d = 100}$ once again - and we find, on our colab machine, this to be the maximum value we could set without crashing the machine itself. Performing **PCA** and padding in this case is likely to lead to a huge loss in terms of accuracy, because we are losing a lot of information when reducing the size of the matrices and introducing noise when padding them to the desired size.

4.3 Convolutional Neural Network

Our convolutional neural network is simply composed of two consecutive *convolutional layers* with **128** and **256** filters each and a both having a unit input channel. *Kernel size* of both convolutional layers is **(5, 5)**.
The output of the second layer is then given as input to a pooling layer, and then flattened be fed to a *linear fully connected layer* whose output is of size **k**, where **k** is the number of classes of our problem - in this case, k = 4 since we chose in both cases only four clusters. This output will consist of four different values which correspond to how likely is each of the four class to be the one the sample belongs to.
Softmax is then applied as activation function of the final layer to output the single class with the highest probability of being the correct one, that is the network's prediction.
Being a classification problem, **cross-entropy** is exploited as loss function, and once again **adam** as optimizer. The results are shown in **Table 2** in terms of evaluation metrics and in **Table 3, 4** in terms of confusion matrix.

Table 2: CNN evaluation metrics

Metrics (avg.)	Results (0, 6, 7, 8)	Results (0, 7, 8, 9)
accuracy	0.49	**0.53**
precision	0.50	**0.51**
recall	0.49	**0.53**
f1-score	0.47	**0.49**

Table 3: CNN confusion matrix (0, 7, 8, 9)				
classes	0	7	8	9
0	**1098**	77	22	76
7	235	**410**	**435**	113
8	74	187	**895**	81
9	**552**	98	348	211

Table 4: CNN confusion matrix (0, 6, 7, 8)				
classes	0	6	7	8
0	**956**	50	245	22
6	169	**651**	268	149
7	391	244	**501**	73
8	**451**	212	230	300

4.4 Preliminar analysis of results

We can draw some conclusions from this first attempt. The overall accuracy and all the evaluation metrics seem no better than a simple random choice classification. We can clearly see from the confusion matrix computed from the evaluation in the case of clusters (0, 6, 7, 8), that the neural network is able to classify the elements in classes 0 and 6 better. This may be due to the fact that the adjacency matrices of the elements in these four clusters are not so different, and especially when comparing their first 100 rows and columns. Or at least, that these elements can be fairly distinguished in clusters 0 and 6, but when it comes to the remaining clusters, the neural network can see no differences among those two and the others. This is also true for the case of clusters (0, 7, 8, 9), but in this case the network is able to distinguish elements of clusters 0 and 8, but the pattern of the matrix is almost the samed - cluster 9 takes cluster 8's place, basically.
We can point out at least three ways to improve these results:

- try and pick higher values for **d** so that we lose less information in the adjacency matrices when performing a PCA or cut, or try to input the adjacency matrices with their original size - providing to the CNN an input of variable size;
- in general, try to augment the original dataset from the 120K functions and try different clusters combination for the classification problem, and also improve the **CNN** by trying different kernel sizes and number of filters to apply;
- find a better way to encode the graphs, not only exploiting the adjacency matrices.

5 Random walks and embeddings

There are several other ways we could encode our functions by exploiting theie **CFGs**. One way, for instance, would be to traverse the graphs by performing a *random walk* over each of them and then encoding each function through the order we meet each node in its **CFG**. This would allow us also to exploit another kind of neural network - namely, a **Recurrent Neural Network** and, in particular, an **LSTM**, since the random walks would provide us with encodings that are *sequences*.
Random walks can be implemented with different strategies, one of them being described by the pseudocode below in **Algorithm 1**.

Algorithm 1: Random Walk over graph G

Result: A list of random-walk-ordered nodes **L**.
G = (V, E);
let v be the entrypoint of G;
L = ∅;
k = 0;
p = T[v];
while *k<100* **do**
    let A be the topologically ordered adjacency matrix of G;
    let T be the topologically ordered transition matrix of A;
    max = argmax{p};
    L = L ∪ {max};
    p = T[max];
end

The transition matrix **T** is computed from the adjacency matrix **A** by computing, for each row, the sum of the non-zero elements and then dividing each element of the row by that sum, so to obtain for each node the transition probability p to traverse the edges in the corresponding row of the adjacency matrix. In practice, we compute **T** for each graph by exploiting the *google_matrix* method from *networkx* library, which is computed following the Pagerank algorithm, so we also set parameter $\alpha = 0$ with 1-α being the well known teleportation probability in the Pagerank algorithm, since there is no such thing as teleportation in functions. Also, instead of choosing the highest probability among the row of **T**, since each row of this matrix is likely to actually have more than one element with same non-zero and highest value, we put these values into a list and then randomly choose one of them to be the traversed edge. Both **A** and **T** are topologically ordered as described in the previous paragraph. Finally, we choose to set k < 100 so that we have a random walk of length **100** for each graph.
This way, we encode each **CFG** as a sequence of 100 nodes, and each element of this sequence is the position of the node itself in the node list of the graph. One may think this would be enough for a **LSTM** to actually learn to classify the graphs and thus the functions, but the results we get are similar to the ones obtained in the previous **CNN** implementation. This may be due to the fact that, apart from learning to traverse the graphs through the sequences, we need a way to actually distinguish between them by assigning a meaningful value to its nodes in the random walk, one that is carrying much more information than its plain index in the nodes list.

5.1 Doc2Vec embeddings for graph's nodes

For each **CFG**, given our dataset, for each node in the graph we also have a *data* field which stores, once again, the *assembly instructions* as plain text that are contained in each node of the graph. By applying the same trimming process we performed for the inputs of our **autoencoder** on paragraph 3, we can now treat each node as a **text document**

containing the instructions of the node, and thus apply **Doc2Vec** [3] to infer the embedding for each node of each graph that is traversed during the random walk, encoding it as a 100-dimensional vector.
This way, we finally have for each **CFG** a sequence of 100 indexes, each of them representing a node traversed during the random walk on the graph and, at the same time, in the lookup table, a 100-dimensional vector that is the encoding we retrieve from the assembly instructions contained in the node.

6 Classification over random walks on CFGs

With the random walks on the **CFG** and the embeddings we retrieved for the nodes of each graph, we now train two different models with embeddings: a **CNN** and a **Bi-LSTM**.

6.1 CNN architecture

The **CNN** is similar to the previous architecture, but it exploits a very well known design for CNN text classification reported in [2].
The embedding layer is the aforementioned one, but the vectors are then fed to a two-dimensional *convolutional layer* which has a unit input channel composed of 100 filters and a kernel of length **(5, 100)**, meaning each filter takes into account 5 adjacent vectors in the sequence at a time. This is du to the fact that the **Doc2Vec** model was trained with parameter window size **5**, meaning the documents take into account the adjacent documents in a range of five to the left and to the right in the sequence, and we do the same in the CNN. The output of size 100 is then given to the *linear fully connected layer* after applying a one-dimensional *max pooling*. The linear output is of size 4, and once again a **softmax** activation gives as output the class to which the sample is most likely to belong to.

6.2 Bidirectional LSTM architecture

The **LSTM** is composed of an embedding layer which is basically a lookup table: it receives as input the sequence of length 100 of nodes traversed during the random walk, where each element corresponds to an index representing a node of the traversed graph, pointing to a 100-dimensional vector which encodes the assembly instructions contained in the node itself. The vector is then fed to a **Bidirectional LSTM layer** with a hidden output (cells) of size 100. Finally, the hidden output (of size 200) is given as input to a *linear fully connected layer* which outputs 4 values, and the final activation function - a **sigmoid** - outputs the most likely class to which the sample belongs to.

6.3 Results

We train the two neural networks over several combinations of the dataset of the retrived clusters. We find out the **CNN** outperforms every other architecture we tried with *43K* training functions and almost *5K* evaluation functions, with four clusters (0, 7, 8, 9) cut

Table 6: CNN confusion matrix (0, 7, 8, 9)				
classes	0	6	7	8
0	**795**	2	438	38
7	10	**500**	3	**680**
8	184	2	**1035**	16
9	11	332	12	**854**

Table 7: Bi-LSTM confusion matrix (0, 6, 7, 8)				
classes	0	6	7	8
0	**926**	131	196	20
6	2	**1070**	2	119
7	223	238	**771**	5
8	1	**992**	4	211

to the same length of 12280. The **LSTM** outperforms the **CNN** and reaches its best results when exploiting the same training and evaluation dataset configuration, but with clusters (0, 6, 7, 8). All the other combination of clusters tried for both architectures reach lower results in terms of evaluation metrics. Both neural networks were trained with **adam** optimizer at same learning rate (0.001) and same loss function - **multiclass Cross-Entropy**.
Tables 5, 6 and **7** (confusion matrices and evaluation metrics) show the results obtained with both architectures.

Table 5: LSTM/CNN metrics		
Metric (avg.)	LSTM (0, 6, 7, 8)	CNN (0, 7, 8, 9)
accuracy	0.61	**0.65**
precision	0.66	**0.66**
recall	0.61	**0.65**
f1-score	0.58	**0.64**

7 Final remarks and future improvements

Every result that is reported in this paper, should be framed taking into account the fact that the original dataset provides no exact *labels* for the functions: when performing the classification task, each sample that is correctly predicted by the neural network as belonging to the expected class, is to be intended as a sample that has been linked twice, first to the same cluster and then to the same class representing that cluster. The clusters we have retrieved through the *k-means++* implementation could, indeed, be very far from a ground truth labeling process that might be carried out by a human expert - e.g., sample **x** that was clustered in cluster **0**, might not share many common features with the rest of the samples in the very same cluster, misleading the neural network trying to predict its label: cluster **0** itself might be composed of several smaller clusters that were not identified by the algorithm in the first place.
The results reported are indeed the best ones that were obtained among several runs: depending on how the dataset is split into training and test sets at each run, and on which clusters we choose as labels, they may heavily change due to the ambiguous nature of the labels. An analysis on the retrieved clusters and a subsequent, exact labeling procedure

performed by a human expert should indeed be the first future improvement.

About the first implementation of the **CNN** exploiting adjacency matrices, we have already discussed how 100x100 matrices are probably not enough to actually capture all the features of the **CFGs**. A possible way to overcome this, might be to try and color the matrices - as they were images - augmenting the channel input size, or simply increasing their dimension.

Random walks with corresponding embeddings seem to improve the results, and should be further explored. The embeddings that were exploited, obtyained through **Doc2Vec**, are mainly due to text data: it would be interesting to try different embeddings, more related to the geometrical features of the graphs - one could be **Node2Vec** [4]. Traversing the graphs with a random walk might lead us to encode the nodes and thus the graphs in terms of nodes degree or its flow. Also, a different and possible type of embedding could be retrieved from several subsequent random walks on the same graph as showed in **Deepwalk** [5].

We should also stress another point: when performing the functions vectorization through the assembly instructions as text both through tfidf and bag of words, as well as the **Doc2Vec** embeddings in the last two neural networks implementations, we are only taking into account the plain instructions by truncating them and erasing every other string such as registers, offsets and possible numeric or string values. By actually considering all these values that we are cutting off, the results in terms of functions and nodes representations might improve. For instance, in [1] offsets exceeding a certain boundary are replaced by the keyword **IMM**, while the others are retrieved as they are. Also, as we point out in the third paragraph, the Ghidra script retrieves from unstripped binaries - and this works also from stripped ones - strings that are statically defined in the binary themselves, which may be very useful to represent the functions and their nodes; some of them might, indeed, contain words that may identify the operations carried out by the functions - for instance, a string representing an IP address might suggest the function is carrying out network operations. Defining a more accurate preprocessing phase might improve the results we obtained, and this should be investigated further.

References

[1] *In Nomine Function: Naming Functions in Stripped Binaries with Neural Networks, Fiorella Artuso, Giuseppe Antonio Di Luna, Luca Massarelli, Leonardo Querzoni*

[2] *Convolutional Neural Networks for Sentence Classification, Yoon Kim*

[3] *Distributed Representations of Sentences and Documents, Quoc Le, Tomas Mikolov*

[4] *node2vec: Scalable Feature Learning for Networks, Aditya Grover, Jure Leskovec*

[5] *DeepWalk: Online Learning of Social Representations, Bryan Perozzi, Rami Al-Rfou, Steven Skiena*