Binary Functions Similarity: An Exploration of Machine Learning Algorithms in Similarity Match

Asmita
Electrical and Computer Engineering
UC Davis
aasmita@ucdavis.edu

Ning Miao
Electrical and Computer Engineering
UC Davis
nmiao@ucdavis.edu

Jiayin Ye
Department of Statistics
UC Davis
jypye@ucdavis.edu

***Abstract*—In this report, we investigate the application of machine learning algorithms in similarity matching for binary functions. We propose several machine learning algorithms and try to use features extracted from instructions to determine whether these two binary functions are similar. Based on the result we obtain, we evaluate which algorithm is more accurate and try our best to explain what makes the differences among these learning algorithms.**

I. Introduction

In recent decades, due to the increasing popularity of IoT devices and embedded systems, various attack methods have been proposed targeting embedded software. A common approach is to exploit existing software vulnerabilities. Most embedded systems are built based on open-source and third-party libraries to avoid reinventing the wheels and reduce development costs. Such a strategy would cause a severe security concern. As a 0-day vulnerability is exploited, attackers can rapidly utilize such vulnerability and deploy a huge-size attack to any systems that contain the same libraries. Therefore, to minimize such threats, as soon as a new vulnerability is exploited, security analysts need to identify similar functions that are affected.

Static analysis and component identification are widely used to identify software vulnerabilities. In embedded and IoT ecosystems, security analysts usually do not have access to the source code. Therefore, reverse engineering techniques are utilized to analyze such systems. However, even for the same library, the reverse-engineered result is affected by the compiler version, optimization level, and system architecture. Functions similarity is then proposed to solve this problem. Analysts do not have to repeatedly compile the same function in different environments, instead, analysts would compare the target with already identified vulnerabilities. The technique not only reduces the time cost for detecting functions that contain vulnerabilities but also helps analysts better understand system structure during static analysis on binary codes.

In this project, we aim to explore the performance of different machine learning techniques on function similarity identification tasks using mnemonic features extracted from reverse-engineered function binary cfg (control flow graph) as input. The reason why we choose such features is that static analysis usually depends on binary codes and these features can be obtained without access to the source code. Moreover, if two functions are similar, the structures of their instructions should preserve the similarity, and so do the corresponding embedding of instructions. As a result, we can identify the similarity between a pair of functions by feeding the embedding.

The remainder of the report is arranged as follows. The dataset is introduced in Section II. The methodology and the implementation are explained in Section III. The evaluation of different machine learning algorithms' performances on the binary functions similarity task is reported in Section IV. Our findings and limitations are discussed in Section V.

II. Background

A. Dataset

The dataset used in this project originated from the OpenSSL Dataset provided by [2]. This database contains a total amount of 95535 function graphs generated from two versions of OpenSSL binaries. To obtain these binaries, researchers in [2] compiled OpenSSL (v1_0_1f and v1_0_1u) for both x86 and ARM platforms with 4 optimization levels of the compiler (gcc-5.4.0). The complete database contains various sub-tables. In this project, we use lstm_cfg primarily.

The complete database also includes pre-divided train and test sets, which only contain id. Each pre-divided set contains two kinds of function pairs: similar (true) pairs, which are two CFGs from the same source code, and dissimilar (false) pairs, which are CFGs from different source codes.

The detailed applicable train and test sets are generated based on the lstm_cfg sub-table. Due to the limits of computation resources, we randomly select 10k test pairs and 60k training pairs for evaluation, the ratio of similar pairs to dissimilar pairs is 50%:50%.

To generate the detailed applicable train and test sets, we first need to extract the required sub-datasets from the complete database file into csv files. Then for each id-based pair in the provided pre-divided sets, accordingly mnemonic features of each function are extracted from the corresponding sub-table, lstm_cfg. Moreover, since the provided mnemonic features are separated by block level, we had to rebuild a function-level dataset based on data provided in the complete database. The scripts for all back-end dataset creation are provided in our submission.

B. Related Work

Feng et al. [1] proposed a categorization approach utilizing the clustering technique, Genius. They first extract attributed CFG from the original binary codes and then, from extracted CFG, unsupervised learning is used to generate a codebook, which contains similarity metric computation and clustering. Unlike traditional clustering algorithms, instead of using numeral input, they use kernel matrix as input, which is generated from the similarity computed previously. Using this codebook, raw features are transformed into higher-level numerical vectors which are associated with the target functions. Their results show that they can reach a fairly high positive rate ($\sim 80\%$) with a trade of negligible false positive rate ($\sim 3\%$) and still maintain a high processing rate.

On the other hand, Xu et al. [3] claimed their approach, namely Gemini, outperforms [1]. In this research, the authors used deep neural network to produce function embeddings. Authors claimed that the training speed, compared with [1] is boosted from more than 1 week to 30 min to 10 hours and still maintains a higher true positive rate ($\sim 85\%$) with a trade of a similar false positive rate ($\sim 3\%$).

III. Method and Implementation

Before introducing the learning algorithms we evaluate, the input data, both from test set and train set will be explained and discussed. The first thing we do is to slice or extend a designated length of instructions from the target function, in our case, the size is 100 instructions. Thus we would either slice the first 100 instructions or if the target function consists of less than 100 instructions, we use *empty instructions* to fill up to 100 instructions. The next step is to transfer these instructions to pre-trained embeddings using the skip-gram method introduced in word2vec [2]. Each instruction has a specified id, in our case, *empty instructions* would use 0. These instruction-id maps can be found in '**word2id.json**'. Now, we can use the pre-trained embeddings code book '**embedding_matrix.npy**' to convert id to vectorized embeddings. Each set of vectorized embeddings consists 100 dimensions. The corresponding vectorized embeddings for *empty instructions* simply contains 100 0s. For now, we have completed pre-processing for a single function and the result is a matrix with size of $[100 * 100]$.

Then, the vectorized embeddings from the two target functions are concatenated together to form a matrix with the size of $[2*100*100]$. This is the input we desire. Such input will be passed to machine learning algorithms like a 2-channel image, each channel presents a single function and both channels have the same size.

So far, we have finished building applicable train and test sets. The following subsections will explore and evaluate how different algorithms perform for binary similarity. The model scripts are provided in our submission. The experimental environment is our lab's Rubichest server. The algorithms we evaluate in this project only cover those introduced in this course, including K-nn, MLP, CNN, and SVM. The overall process is presented in Figure 1.

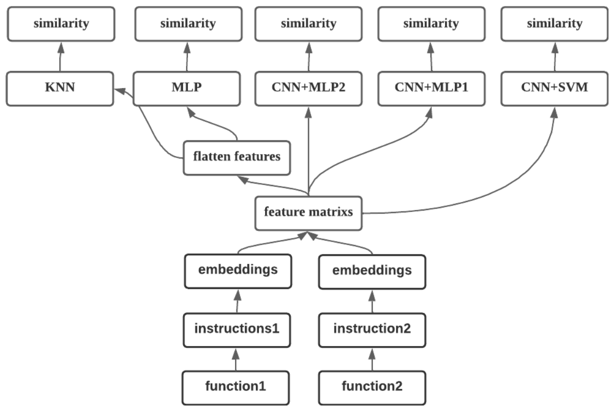

Fig. 1. Implementation process

A. Base Line Model: K-nearest neighbors

The baseline model for this study is K-nearest neighbors with cosine similarity as the distance measurement. K-nearest neighbors is one of the most popular machine learning algorithms applied in similarity match. This algorithm calculates the distances between the test data point and each data point in the training dataset and classifies the test data point based on the labels of its k nearest neighbors in the training dataset. The cosine similarity of two vectors is simply the cosine of the angle between them. We flatten the feature matrix for both channels and use the sum and difference between these channels as input. For better efficiency and lower performance overhead, we use the square root of the sample size of the training dataset as K. We choose this algorithm as our baseline model because it is widely applied in text mining to measure similarity.

B. Multilayer Perceptron (MLP)

The MLP model evaluated has an input layer, two hidden layers, and an output layer. Compare with the K-nearest neighbors model, the input takes a simple concatenation of the flattened feature vectors of two functions. For both hidden layers, the ReLU activation function is used for non-linearity. The learning rate selected for this model is 0.001. The batch size is 100 for 30 and 100 epochs.

1) Cross Entropy loss: Cross Entropy Loss is used for the MLP model. The final output layer has 2 neurons, which correspond to the probabilities of the two classes: similar and dissimilar, respectively.

2) Binary Cross Entropy loss: Binary Cross Entropy Loss with a sigmoid activation function is used for another MLP model. The final output layer has 1 neuron, which is the probability of the classes similar.

C. Convolutional neural network (CNN)

In this model, the input we use is the one discussed at the beginning of this section, which contains 2 channels and each one stands for 1 function.

1) CNN+MLP2: This model consists of 2 convolutional layers with max-pooling layers, 2 hidden layers with ReLU activation function, and the final output layer with cross-entropy loss, similar to what we did for our 3rd assignment.

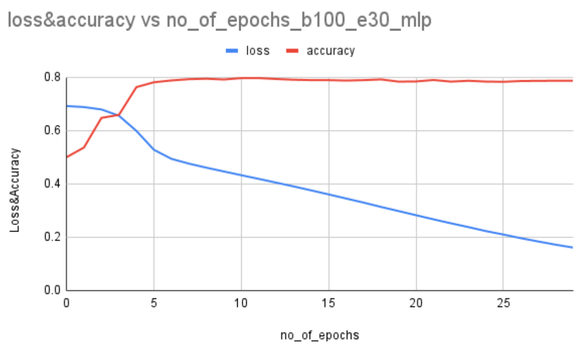

Fig. 2. Loss & Test accuracy vs number of epochs (for 30 epochs and 100 batch size (MLP-CE).

Due to time constraints, we trained it for 30 epochs with the batch size of 100.

2) CNN+SVM: This model consists of 2 convolutional layers with max-pooling layers, a hidden layer, 2 dropout layers, and the final output layer with hinge loss, in which l2 penalty is 0.01. ReLU activation function is again used for non-linearity. Adam optimization is used with a learning rate of 0.0001. We trained it for 60 epochs and a batch size of 100. The hyperparameters are not dynamically tuned by Bayesian Optimization with 3-fold cross-validation. We didn't try enough parameter settings due to resource constraints.

3) CNN+MLP1: This model is similar to the above structure but instead of using the hinge loss, it uses binary cross-entropy loss.

IV. Evaluation

To evaluate the performances of our models, we only looked at the test accuracy since the dataset is balanced.

The loss and test accuracy vs epochs for each method (except K-nearest neighbors) are shown in the following figures, in which the blue lines represent loss and the red lines represent the test accuracy.

A. Base Line Model: K-nearest neighbors

We've tried different Ks, such as 51, 101, 151, etc., but the test accuracy is always 57.65%. It is possible that the dimension of features is too high (2*100*100) that merely using the cosine similarity can not capture important information for classification.

B. Multilayer Perceptron (MLP)

1) Cross Entropy Loss: For batch size 100, number of epochs 30, refer Fig. 2

With 100 epochs, the output was: refer fig. 3

The loss decays exponentially and almost becomes 0 after 80 epochs. The test accuracy increases sharply, reaches around 80%, and quickly flatlines within 10 epochs.

2) Binary Cross Entropy loss: Refer fig. 4

With the same hyperparameters as the above model, the loss and accuracy of MLP with binary cross entropy loss and sigmoid activation function almost don't change until the last 5 epochs, which is probably because of the large learning rate.

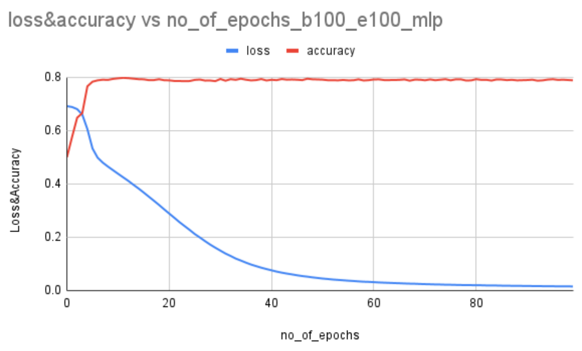

Fig. 3. Loss& Test accuracy vs number of epochs (for 100 epochs and 100 batch size (MLP-CE).

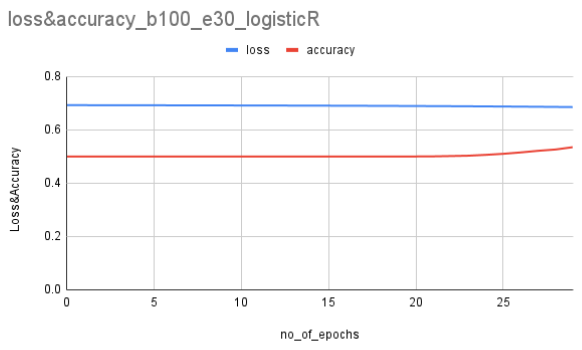

Fig. 4. Test accuracy vs number of epochs (for 30 epochs and 100 batch size (MLP-BCE).

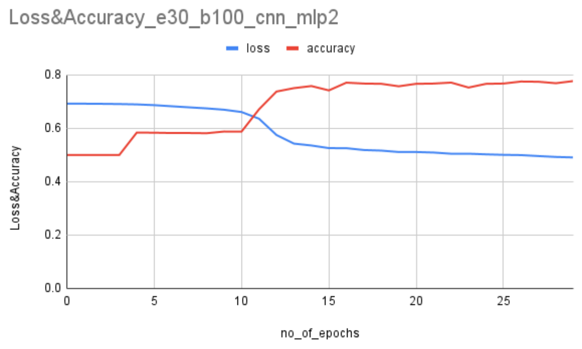

Fig. 5. Loss & Test Accuracy vs number of epochs (for 30 epochs and 100 batch size (CNN-MLP2).

C. Convolutional neural network (CNN)

1) CNN+MLP2: Refer fig. 5.

It has two hidden layers with 120 and 84 neurons respectively (chosen randomly). Moreover, its accuracy is just slightly less than MLP. But the training time is much longer than MLP.

2) CNN+MLP1: Refer fig. 6

3) CNN+SVM: Refer fig. 7

CNN+MLP1 and CNN+SVM have similar patterns in loss and test accuracy. The loss decreases for the first 20 epochs and then flatlines. The test accuracy achieves around 75% after 10 epochs and then fluctuates a little.

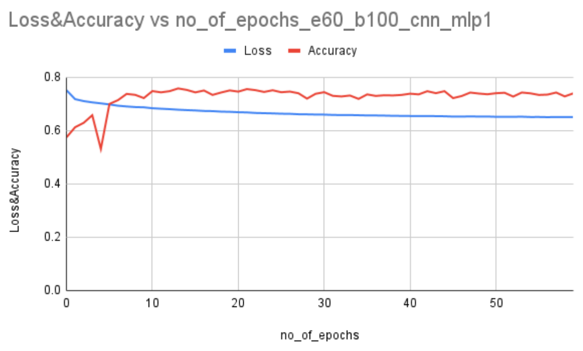

Fig. 6. Loss & Test Accuracy vs number of epochs (for 60 epochs and 100 batch size (CNN-MLP1).

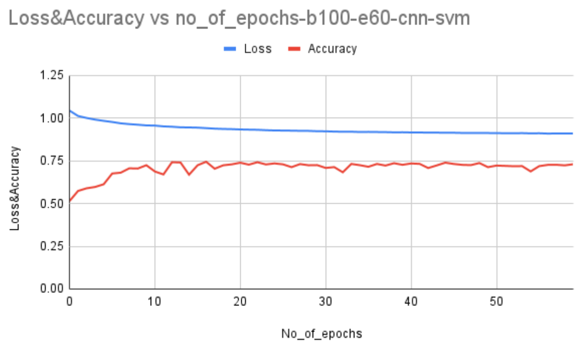

Fig. 7. Loss & Test accuracy vs number of epochs (for 60 epochs and 100 batch size (CNN-SVM).

D. Model Comparison

Table I and Figure 8 show the final results and a comparison between all models.

Among all methods we applied, MLP with Cross Entropy Loss (the red line) obviously outperforms the baseline model (the black line) and other methods with a test accuracy of 79% and a faster convergence rate.

Moreover, the hybrid models CNN+MLP2 (the green line), CNN+MLP1 (the orange line), and CNN+SVM (the grey line), although have more complicated structures, actually underperform the simple MLP model with test accuracies of 78%, 74%, and 73%. But it took much longer time for training than MLP. We believe the CNN layers are not well-tuned and lose some information or the training dataset doesn't have enough data to train a large number of parameters. And the test accuracy of the CNN+SVM model fluctuates a lot compared to other models. There is a possibility that the cross entropy loss is more suitable than the hinge loss for this task.

The test accuracies of other methods are all below 60%.

V. Other Attampts

Besides the lstm_acfg is utilized, we also try to train models by the numerical acfg features. This sub-table only provides 6 attributes: number of string constants, number of numeric constants, number of transfer instructions, number of calls, number of instructions and number of arithmetic instructions. Therefore, as discussed previously, to compose the train and test sets, each instance would include 12 features, 6 for each function, and labels. To exam whether numerical features can be used to determine whether two functions are similar, we evaluate 4 models: K-nn, Random Forest, MLP with SGD optimizer and MLP with Adam optimizer. The result is show in table II. The loss & accuracy curve for 2 MLP model shows in fig 9 (with SGD as optimizer) and fig 10 (with Adam optimizer). From these 2 figures we can see that when using Adam optimizer, the curve is more smooth. Both figures shows a flat accuracy curve and starts from around 80% of the accuracy. This proves that the numeric acfg features can also be used to help determine whether 2 functions are similar. Such features do not need lots of time for the training purpose and therefore suitable to be used as rapid classification and provide a fair accuracy.

Algorithms	Loss Function	Epochs	Test accuracy
KNN			57.65%
MLP	Cross Entropy	100	**78.85%**
	Binary Cross Entropy	30	53.51%
CNN+MLP2	Cross Entropy	30	77.68 %
CNN+MLP1	Binary Cross Entropy	60	73.97%
CNN+SVM	Hinge	60	73.01%

TABLE I
TEST ACCURACY FOR MNEMONIC LSTM_CFG DATASET

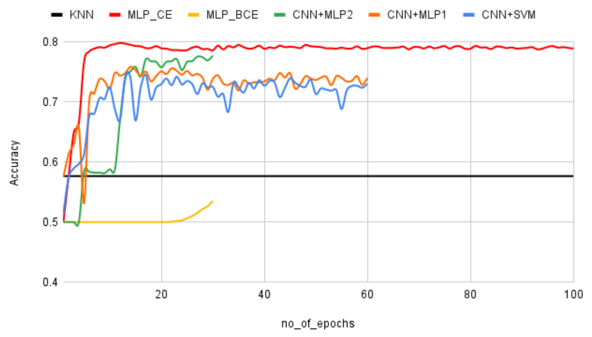

Fig. 8. model comparison.

Algorithms	Loss Function	Epochs	Test accuracy
KNN			79.32%
Random Forest			71.24%
MLP with SGD	Cross Entropy Loss	50	80.26%
MLP with Adam	Cross Entropy Loss	50	84.34%

TABLE II
TEST ACCURACY FOR NUMERIC ACFG DATASET

VI. Discussion

In this project, we explored various algorithms. The MLP model outperforms others by higher final test accuracy. There is a lot more scope to improve this project which we will further take forward as part of our Ph.D. research. Due to

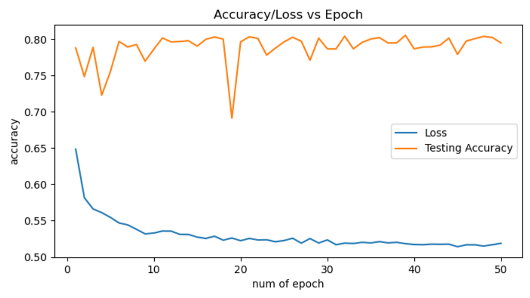

Fig. 9. Loss & Test accuracy vs number of epochs (MLP with SGD).

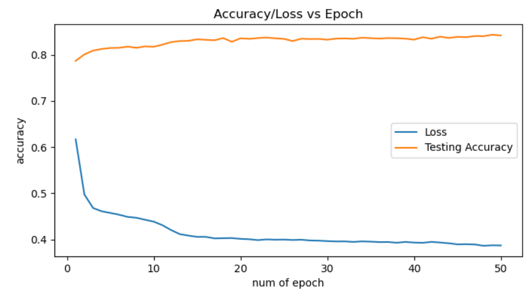

Fig. 10. Loss & Test accuracy vs number of epochs (MLP with Adam).

time and resource constraints, we didn't explore further by tuning different hyper-parameters. Moreover, there is a scope to improve the performance by tuning the dataset features, for example, logically selecting the 100 unique instructions, instead of fetching the first 100. On the other hand, exploring *bag of words* and *graph embedding neural networks* would be more promising. This is the initial stage of our research where it tried to identify whether machine learning is suitable for binary-level similarity detection without access to the source code. As part of our future research, apart from working on the above limitations, we aim to make this binary similarity possible for embedded firmware binaries, which are widely used in IoT devices. These devices use a lot of third-party software components, which our research would help in the identification of vulnerabilities at. As these third-party APIs and SDKs are used in several products, this work would fasten the static analysis process by identifying the similar libraries used in different products and linking the vulnerabilities associated with same libraries.

VII. Role of each team member

A. Asmita

I worked on the back-end dataset generation, feature extraction from binaries cfgs, and feature vectorization to get embedding vector corresponding to each feature of respective functions. After this, I worked on applying these MLP, and CNN algorithms on these datasets, and then finally wrote a part of the report (we worked together on the report and presentation).

B. Jiayin

I worked on data processing by transforming the raw data into input for different algorithms. After this, I applied K-nearest neighbors, CNN+SVM, and CNN+MLP on the datasets, and wrote part of the report and presentation.

C. Ning

I worked on the data extraction (acfg numerical features), and utilized this dataset in various algorithms. I helped wrote part of the report and presentation.

References

[1] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 480–491. [Online]. Available: https://doi.org/10.1145/2976749.2978370

[2] L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," *Proceedings 2019 Workshop on Binary Analysis Research*, 2019.

[3] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 363–376. [Online]. Available: https://doi.org/10.1145/3133956.3134018