Malware detection using machine learning based on word2vec embeddings of machine code instructions

Igor Popov
Siberian State University of Telecommunications and Information Sciences
Novosibirsk, Russian Federation
gaurnar@gmail.com

Abstract— **Applying machine learning for automatic malware detection is a perspective field of scientific research. One of popular methods in static analysis of executable files is observing machine code instructions that they contain. This paper proposes applying word2vec technique for extracting vector embeddings of machine code instructions and evaluates convolutional neural network-based classifier that uses extracted vectors for malware detection.**

Keywords—machine learning, malware detection, machine code instructions, word2vec, convolutional neural networks

I. INTRODUCTION

Malicious software (also known as malware) is a major threat not only for ordinary Internet users, but also for big corporations and even government organizations. Inability to response to the threat of malware often has grave consequences. Complexity and diversity of malware rises each year, forcing antivirus product vendors to come up with new methods of their detection.

Traditional approach employed by antivirus software is a signature method. It involves creating so called signatures (bits of machine code, file hashes etc.) for various malware samples. However, such approach is becoming less and less effective [1]. It is caused by increasing diversity of malicious software, as well as proliferation of its polymorphic variants. It is also obvious that signature methods are ineffective at detecting new types of threats.

One of possible solutions is applying machine learning. It involves building special models that are capable of "learning" from input data and later using them for processing new (previously) unseen data. Input data come in the form of features describing analyzed objects. Features can be integer or floating point numbers, Boolean ("true" or "false") and categorical values. In application to malware detection, features may include various characteristics of software modules (executable and library files) and detection itself is a classification problem, involving two labels: "malicious" and "benign".

Applicability of various machine learning models depends not only on the task at hand, but also on the number and characteristics of features. Thus, selection and processing of features is vital for building any machine learning system. For malware detection domain, all feature selection methods can be roughly divided into two types: dynamic and static analysis.

Dynamic analysis involves executing software samples and collecting data regarding their behavior and changes in execution environment. It is obvious that such approach allows to get the most reliable (and thus valuable for classification) data. However, it also requires complex infrastructure and is costly, both in required resources and in time. Latter is caused by necessity to run each sample for some time (enough for malware to manifest itself), which can significantly slow down the research process for large datasets.

Static analysis, on the contrary, implies examining software modules without directly executing them. Such approach involves analyzing metadata of executable modules (for instance, contents of Portable Executable format headers for Windows operation systems), machine code instructions (extracted using disassembler), and binary data (including resources: images, icons, character strings etc.). This approach stands out by its speed, lower infrastructure requirements and thus availability to a wider audience of researchers. Moreover, practice shows that static analysis is often as effective as more costly methods (like dynamic analysis). One of its major disadvantages is vulnerability to obfuscation, which involves altering structure and contents of executable modules in an attempt to obstruct their analysis by antivirus products.

This paper covers only one of two executable modules analysis methods mentioned above – static analysis, as being the most straightforward to set up.

One of promising methods for extracting features from analyzed software samples is aforementioned processing of machine code instructions. It is self-evident that machine codes that are contained in executable modules define their behavior. However, they cannot be directly transformed to machine learning features (e.g., like it can be done for metadata). One of possible solutions for extracting features from machine code is to apply methods that are widely used in natural language processing. What justifies such approach is that there is some resemblance between words in natural language and machine code instructions, as well as between natural language sentences and machine code blocks.

Applying methods that has proven their effectiveness in natural language processing domain was already studied in malware detection context, including their usage for extracting features from machine code instructions. One of such examples is N-gram extraction method applied for machine code instructions presented as a sequence of codes. Here features

may be presence of N-gram in question in analyzed sample (Boolean feature), frequency of N-gram etc. It is worth noting that practically always it is required to take only most frequent N-grams for feature extraction, because the space of all possible N-grams is too big. Using such approach, Yuxin et al. [2] as well as Zolotukhin and Hamalainen [1] managed to reach high accuracy scores: 94% and 97.09% correspondingly. However, such feature extraction method does not consider some significant characteristics of machine code instructions, namely their context and order (unless we take N to be big enough, which is not plausible because of explosive growth of space of possible N-grams).

Approach proposed in this paper involves using word2vec technique [3], which recently gained popularity in analysis of natural language texts. Word2vec names a group of related models, which are used for generating embeddings for words from texts in natural languages. Such models are two-layered "shallow" (as opposed to "deep") neural networks. In one of approaches neural networks learns to predict neighbor words by given word (so-called "skip-gram" approach), in another one – to predict current word by given neighbors ("Continuous Bag Of Words", CBOW). After learning is complete, we can get word embeddings (vector representations) for each input word. Such vectors are shown to have an interesting property: vectors of semantically close words have small distance between them.

Usage of aforementioned embeddings for classification of natural language sentences was studied by Kim [4]. His model is based on convolutional neural networks, which learn from word2vec vector representations of words from a window of chosen size. Convolutional neural networks are artificial neural networks of a particular architecture, which were first proposed by LeCun [5]. Initially used for computer vision domain, they proved their effectiveness for other domains of machine learning, including natural language processing.

This paper reviews building a proof-of-concept classifier based on convolutional neural network with architecture resembling the one described in [4]. Conducted evaluation of said classifier on test set of samples shows satisfactory results, which encourages continued research in this area.

Proof-of-concept model was developed using *Anaconda* – scientific distribution of Python programming language. Author also used *TensorFlow* library by Google (Python bindings) for building neural networks and word2vec implementation from *gensim* library. Prototyping and visualization was done using *Jupyter Notebooks* web-based environment. Finally, machine code instructions were extracted using *Capstone* disassembler (Python interface).

II. Machine code instruction embeddings

As previously noted, for generating embeddings of machine code instructions we will use word2vec technique. Word2vec algorithm expects as its input a set of sentences in natural language. For our case, we will treat machine code instructions as words. For "sentences", we will take a sequence of instructions that is limited on both of its sides by either an end of analyzed portion of executable module or non-code binary data.

For building word2vec model we will use machine code instructions from the first kilobyte (chosen arbitrarily) of executable module data, starting from entry point. Entry point was chosen as a start offset for extracting machine code mainly because we can be sure that the data at this offset is definitely machine code (and not some arbitrary data, which can also reside in code section of executable). To identify machine code instructions we will use already mentioned above *Capstone* disassembler, taking only instruction opcode and omitting instruction arguments. The latter is done to lower the variability of "words" to account for relatively small dataset.

In order to construct feature vectors for analyzed samples we identify first n instructions (possibly with gaps filled with non-code data between them), again starting from entry point of executable. Then we concatenate embeddings of found instructions (taken from previously trained word2vec model) to form a feature vector of size $m \cdot n$, where m is the size of instruction embedding (m, n and other similar symbols here and later in the paper are hyper-parameters of the model, which were analyzed during conducted experiments). If embedding for some instruction is not known to the model, we use zero vector instead. Similarly, if some sample does not have n instructions in the first 10 kilobytes (taken arbitrarily based on personal judgement) from its entry point, we pad the feature vector with zero vectors.

III. Proof-of-concept model

As already stated, the model proposed in this paper resembles the model proposed by Kim in [4]. Specifically, it contains one convolutional layer and one fully connected one. Let us review model architecture in more detail.

A. Convolution layer

For applying convolution, feature vector is transformed in matrix with m and height n (i.e. i-th row in the matrix corresponds to i-th machine code instruction taken from analyzed sample). The convolution filter has width m (i.e. the whole row), height h and number of output channels k (h and k are another two of model hyper parameters). Convolution filter is moved only vertically, the stride being one row at a time (which is equivalent to so-called "valid" padding with stride of one in both dimensions).

For convolution layer of our model we will use *rectifier* activation function (neuron that has such activation function is called *rectifier linear unit*, or ReLU), which is popular in deep learning field. Function itself is very simple and can be formulated as $y(x) = max(0, x)$.

B. Fully-connected layer and softmax

Next in our model's architecture comes fully connected layer of neurons. For its input, we concatenate all convolution output vectors. Here we also apply technique frequently used to prevent overfitting in deep neural networks – *dropout* [6]. It essentially means zeroing vector components with some probability p_d. Finally, output of fully connected layer has two components, which can be treated as probabilities of two classes ("malicious" or "benign") after applying *softmax* function (also called *normalized exponential function*). The

latter essentially "squashes" a vector of arbitrary real values to a same-dimensional vector of real values in the range (0, 1) that add up to 1.

C. Training

Training of the described model is done using backward propagation of error. Specifically, here we use Adam optimizer [7] (featuring, among other things, learning rate decay) to minimize cross-entropy loss function calculated from model output (after applying *softmax* function). To further prevent overfitting we also add to loss function L2 regularization terms for model weights.

IV. Model evaluation

Model was evaluated on 1200 malicious and 1200 benign PE executable samples. Malicious samples were taken from VirusShare.com, open repository of malware samples (https://virusshare.com). Benign samples were collected from local workstations running Windows operation system from the following folders: “Program Files”, “Windows” and “Downloads”. Benign samples were additionally scanned with antivirus.

Experiments were conducted to manually find semi-optimal hyper-parameters of the proposed model and find the best achieved F1 score. The following parameters were analyzed:

- n – number of machine code instructions to form feature vector;
- m – size of word2vec embedding;
- h – height of convolution filter;
- k – number of convolution output channels;
- p_d – probability of dropout;
- C_{L2} – L2 regularization coefficient.

Tables I – V show observed dependencies of F1 score (our main metric) on various hyper-parameters of the model.

Table I shows dependency on hyper-parameter n. Other parameters were fixed with values: $m = 500$, $k = 16$, $C_{L2} = 0.01$, $p_d = 0.5$. It is also worth noting that the maximum tested value of n is 200 because of the constraints of proof-of-concept model implementation (namely memory consumption inefficiency). Continued search for optimal n value is subject to further research.

Table II shows dependency on hyper-parameter m. Other parameter values: $n = 100$, $h = 10$, $k = 16$, $C_{L2} = 0.01$, $p_d = 0.5$.

Table III shows dependency on hyper-parameter k. Other parameter values: $n = 100$, $m = 500$, $h = 10$, $C_{L2} = 0.01$, $p_d = 0.5$.

Table IV shows dependency on hyper-parameter p_d. Other parameter values: $n = 100$, $m = 500$, $h = 10$, $k = 16$, $C_{L2} = 0.01$.

Finally, Table V shows dependency on hyper-parameter C_{L2}. Other parameters were fixed with values: $n = 100$, $m = 500$, $h = 10$, $k = 16$, $p_d = 0.5$.

Analyzing experiment results, we can deduce semi-optimal hyper-parameter values: $n = 200$, $m = 500$, $h = 10$, $k = 16$, $p_d = 0.5$, $C_{L2} = 0.01$. Using such parameters, we achieve F1 score of 97.2%. Such high scores may be attributed to unintentional dataset deficiencies, but they nonetheless show plausibility of proposed approach.

TABLE I. Dependency of F1 score on number of instructions and height of convolution filter

n	**h**				
	5	***10***	***12***	***15***	***20***
50	65.9	65.9	65.9	66.3	66.3
100	94.7	96.2	95.2	68.9	68.9
200	95.9	**97.2**	**97.2**	96.3	96.4

TABLE II. Dependency of F1 score on size of word2vec embedding

m	**F1 score**
250	95.4
500	**96.2**
750	93.0

TABLE III. Dependency of F1 score on number of convolution output channels

k	**F1 score**
1	93.5
4	94.3
8	94.4
16	**96.2**
24	95.2
32	95.1

TABLE IV. Dependency of F1 score on probability of dropout

p_d	**F1 score**
0.25	94.4
0.5	**96.2**
0.75	94.7

TABLE V. Dependency of F1 score on L2 regularization coefficient

C_{L2}	**F1 score**
0.1	93.4
0.01	**96.2**
0.001	95.2

V. Conclusion

In this paper, we have described usage of machine code instruction embeddings, extracted using word2vec technique, for malware detection. We have evaluated proposed approach using proof-of-concept classifier featuring one convolutional and one fully connected neural network layers. Conducted experiments show satisfactory results, allowing us to treat continued research in this direction as perspective. Continued research includes improving dataset, experimenting with machine code extraction techniques (such as control flow graphs) and model architecture, optimal hyper-parameters search.

References

[1] M. Zolotukhin and T. Hamalainen, "Detection of Zero-day Malware Based on the Analysis of Op-code Sequences" in *2014 IEEE 11th Consumer Communications and Networking Conf.*, Las Vegas, USA, 2014, pp. 386-391.

[2] D. Yuxin, D. Wei, Z. Yibin and X. Chenglong, "Malicious Code Detection Using Opcode Running Tree Representation" in *2014 9th Int. Conf. on P2P, Parallel, Grid, Cloud and Internet Computing*, Guangdong, China, 2014, pp. 616-621.

[3] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space", arXiv:1301.3781 [cs.CL], Jan. 2013.

[4] Y. Kim, "Convolutional Neural Networks for Sentence Classification", arXiv:1408.5882 [cs.CL], Aug. 2014.

[5] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition", *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[6] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", *J. Machine Learning Research*, vol. 15, no. 1, pp. 1929-1958, 2014.

[7] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", arXiv:1412.6980 [cs.LG], Dec. 2014.