Malware detection using Recurrent Neural Networks

Antoine Cajot[1]
[1] *antoine.cajot@student.uliege.be (s125729)*

I. INTRODUCTION

Malware, a portmanteau for malicious software, can be found everywhere and can take an infinite number of forms : from Trojan horses to the recent outburst of ransomware, without forgetting rootkits, spyware, worms, botnets and so many others. Because of the havoc they can wreak, detecting them as early as possible is critical — preferably before they have a chance of executing their malicious payload — and can be tackled in various ways.

On the one hand, machine learning algorithms and, in particular, deep neural networks have nowadays become one of the focal points of research on malware detection [1] [2] [3] [4] [5] [6] because, especially, they are very potent algorithms for recognizing patterns, an asset which is incredibly useful in malware detection. On the other hand, relying on deep neural networks for malware detection and classification makes these tasks vulnerable to adversarial attacks that are plaguing (deep) neural networks (see [7] and [8], among others).

II. SOME RECENT WORKS ON THE TOPIC

In [9], the authors relied on RNNs or echo state networks (ESNs) to extract features from binaries and then processed those features using a standard classifier. The main interest of relying on RNNs or ESNs to extract features is to make the features more robust to obfuscation methods, such as reorderings, occurring in dynamic malware by modelling the malware language model. What is innovative in their approach is the use of max-pooling with RNNs and ESNs — max-pooling was previously generally only used for CNNs. They obtained a true positive rate improved by a factor of 3 compared to bag-of-events based methods, and by a factor of 2 compared to bag-of-trigrams based methods. For most of their experiments, ESNs also performed better than RNNs but altogether, both ESNs and RNNs with max-pooling presented better results than the other eight more traditional methods tested.

A similar approach is undertaken in [10] where the malware language is modelled using either LSTM cells or a gated recurrent unit (GRU) instead of a RNN or an ESN. In addition to the max-pooling employed in [9], an attention-based mechanism is also tested as replacement : an attention score is first determined for each time step and, in the end, the temporal average of all hidden vectors is worked out. For the classification part, a logistic regression (LR) is tested against a multi-layer perceptron (MLP) with a single hidden layer and ReLU activation. Finally, a new one-step model (going directly from the file event stream to file prediction) is also introduced relying on sequences of 1014 events (padded if shorter) as input features to a CNN classifier, more akin to the usual malware classifiers than the two-step model-classification approach described previously. Of all the combinations they tested, the overall best performing model was the LR-LSTM architecture combined with max-pooling even though the MLP-LSTM architecture combined with max-pooling was not far behind, nor was their one-step CNN classifier.

Instead of relying on RNNs to extract features, the authors of [11] use them to process a flow of system calls. The flow of system calls (after being preprocessed to remove redundancies) is fed to a convolutional neural network before going through a layer of LSTM cells. By combining CNNs and RNNs, they are able to obtain overall significant improvements over simpler network architectures or more standard methods such as Hidden Markov Models (HMV) or Support Vector Machines (SVM).

Another important result is obtained in [12] where it is found out that an ensemble of RNNs is able to determine the maliciousness of an executable within only 5 seconds of its execution with an accuracy of 94%. The authors claim it is the first time general types of malware can be detected in early stages of execution rather than relying on post-execution activity logs. The input features are a combination of ten machine activity data metrics : a snapshot of these is taken every second of execution. Machine activity features were preferred over system calls features as the latter are more easily manipulated, making adversarial attacks easier. In addition to robustness to obfuscation, machine activity features are also continuous whereas the system calls are categorical : regression between values is therefore possible and the model could possibly infer information from unseen input values. The last advantage of activity features lies in their short size (a small vector of size 10). The metrics were the following : system and user CPU usage, packets sent and received, bytes and received, memory and swap use, the total number of processes currently running and the maximum process ID. The architecture was a RNN based on GRUs as they are supposedly faster to train. The authors also noticed that even when the model had not been trained on a specific malware did it still succeed in detecting it with a very decent accuracy.

Eventually, even though it did not cover RNNs, a notable mention is the work of J. Saxe and K. Berlin [1], where a deep feedforward neural network of four layers is leveraged. It is worth mentioning, in particular, for the specificity of the input features, a set of four complementary features extracted separately from the binaries : byte/entropy histogram features to help with context detection, portable executable (PE) import features (based on DLL import tables), string appearance features, and PE metadata features extracted from numerical fields of the PE's packaging. Their system achieved a 95% detection rate at 0.1% false positive rate.

III. METHODOLOGY

A. Gathering data

The model is to be trained on successive snapshots of system metrics while the subject software is run inside a virtualized and isolated environment. The virtualized environment chosen is Windows 7 (Ultra x64) as Windows (and Windows 7 particularly, as remaining one of the most deployed OS worldwide [13] and being a few years old already) is targeted by much more malware than the other operating systems.

The first step is therefore to collect samples of malware and benignware to be run inside that environment. The malware part has been made easier by the huge contributing efforts made in VirusShare (see `https://virusshare.com/`). As for the benignware, a good way to find representative samples is to crawl on online directories such as Softonic, PortableApps or PortableFreeware. In this study, it was preferred to stick with portable executable as they are more easily transferred and executed. It should be noted, however, that not including other types of executable is a first important limitation of this study as it diminishes the expressiveness of the data set.

Once the software has been collected, the next step is to run it and to collect the metrics. A proper way to deal with this is to work with the Cuckoo sandbox which offers a great framework for automating tests between the host operating system and the guest virtual machine. The chosen machinery was VirtualBox. A custom script to collect system metrics every second relying on cuckoo was written, relying on the psutil module. Samples were run by Cuckoo for at least 60 seconds in the virtualized environment and all collected data was stored in a SQLite database to allow small access times and easier filtering. In total, 576 benign and 522 malign execution snapshot sequences were collected and the shortest execution sequence was 47 seconds long. The input data set was thus restricted to 522 samples of each and to a maximum sequence size of 47. Also, some of the metrics were evinced, only were kept the following 13 features : cpu user, system, interrupt and dpc usage (see `https://docs.microsoft.com/en-us/azure/monitoring/infrastructure-health/vmhealth-windows/winserver-processor-cpudpctime` for the latter), virtual and swap memory usage, disk i/o read and write count and written and read bytes, bytes sent and received over the network as well as the number of packets and, eventually, the highest process id assigned.

B. Data preprocessing and partitioning

The data set $\boldsymbol{X}$ set was standardized metric-wise $\frac{x-\bar{x}}{\sigma_x}$ before being shuffled and partitioned into the training, validation and testing subsets with respective proportions 0.64, 0.16 and 0.2.

C. Base Model

Because Gated Recurrent Units (GRU) are known to train faster than LSTM cells, they were preferred over the latter. The input layer as well as the hidden layer(s) are bidirectional GRU layers and the output layer is a fully-connected layer with sigmoid activation.

D. Exploring the hyperparameter space

To avoid the tediousness of manual or grid search, a random search in the hyperparameter space was conducted over 52 possible hyperparameter combinations. The hyperparameters to be fine-tuned and the considered part of their domain are summarized in the table below.

Hyperparameter	Considered domain
Epoch count	$[1, 75] \in \mathbb{N}$
Batch size	$\{16, 32, 64, 128\}$
Hidden layers count	$\{1, 2, 3\}$
Optimizer	$\{SGD, Adam,$ $Adadelta, RMSprops\}$
Learning rate	$\{0.001, 0.0015, 0.002\}$
Units per layer	$[1, 200] \in \mathbb{N}$
Dropout rate	$\{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$
Recurrent dropout rate	$\{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$
L1 reg factor (bias)	$\{0, 0.01, 0.02\}$
L2 reg factor (bias)	$\{0, 0.01, 0.02\}$
L1 reg factor (recurrent)	$\{0, 0.01, 0.02\}$
L2 reg factor (recurrent)	$\{0, 0.01, 0.02\}$
Metric sequence size	$[5, 47] \in \mathbb{N}$

Naturally, only the validation set was used to evaluate the performance of these 52 networks.

E. Testing the top configurations

Once the best performing combination of this random search was frozen, it could eventually be evaluated relying on the test set.

IV. RESULTS

The best performing configuration of hyperparameters as far as accuracy on the validation set is concerned is detailed in the table below. The evaluation accuracy achieved was 85.71%.

Hyperparameter	Value
Epoch count	9
Batch size	16
Hidden layers count	2
Optimizer	*Adam*
Learning rate	0.001
Units per layer	115
Dropout rate	0.5
Recurrent dropout rate	0
L1 reg factor (bias)	0.01
L2 reg factor (bias)	0.02
L1 reg factor (recurrent)	0.01
L2 reg factor (recurrent)	0
Metric sequence size	23

After evaluating on the test set, the accuracy of this model drops a little to 82.61%, which is a decent result but could be improved a lot nevertheless (see limitations).

Of course, there are other interesting configuration that are of interest and the evaluation accuracy is not the only comparison metric. For instance, one of them also succeeded in securing a bit more than 80% of accuracy but using only 6 sequences of metrics, which, as mentioned in [9], seems to imply that malware could be identified in early stages of execution using only system metrics.

V. LIMITATIONS OF THIS STUDY AND FUTURE WORKS

Some bottleneck limitations of this study are related to the data set. They should be reviewed in their chronological order of appearance.

First, before execution, comes the fact that all of the benignware collected are portable executable and do not reflect the diversity of shades of software that can be found on and offline. Also, the software collected is too ludicrously small in number to be statistically representative of the population (it could be at least multiplied by four to five).

Second, during their execution in the virtualized environment, it should be obvious that a lot of applications require proper user interaction to actually perform useful work. Therefore, metrics collected while Cuckoo emulates random pointer and keyboard gestures with the app running will never be actually representative of an "actual" thread of execution. Collecting those metrics on actual running systems with actual users might be a far more suitable approach.

A great deal of other limitations are related to the methodology. On the one hand, time did not allow to test for good old LSTM cells instead of GRU, nor to compare this base architecture to others such as Echo State Networks. The hyperparameter random search was also very weak because it involved only 52 random combinations of hyperparameters but could have easily been tripled in number.

Finally, there are a few implementation hiccups. For instance, it was noticed after the collection of system metrics that one of the metrics had not been correctly extracted or ingested into the database, the number of running processes. It was therefore impossible to include this in the input features and had to be evicted, while it would probably have helped a lot with the prediction accuracy of the network. And this is just one hiccup among others.

VI. IMPLEMENTATION

The full implementation of this study as well as the extracted metrics database and some of the results can be found on
`https://github.com/PixelWeaver/MalwareClassification`

[1] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. *CoRR*, abs/1508.03096, 2015.

[2] Wenyi Huang and Jack W. Stokes. Mtnet: A multi-task neural network for dynamic malware classification. In *DIMVA*, 2016.

[3] Wei Wang, Mengqiang Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. Malware traffic classification using convolutional neural network for representation learning. *2017 International Conference on Information Networking (ICOIN)*, pages 712–717, 2017.

[4] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference*, pages 141–147, Aug 2012.

[5] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1):343–357, Jan 2016.

[6] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye. Significant permission identification for machine-learning-based android malware detection. *IEEE Transactions on Industrial Informatics*, 14(7):3216–3225, July 2018.

[7] 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, Sep. 2018.

[8] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[9] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920, April 2015.

[10] B. Athiwaratkun and J. W. Stokes. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2482–2486, March 2017.

[11] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In Byeong Ho Kang and Quan Bai, editors, *AI 2016: Advances in Artificial Intelligence*, pages 137–149, Cham, 2016. Springer International Publishing.

[12] Matilda Rhode, Pete Burnap, and Kevin Jones. Early-stage malware prediction using recurrent neural networks. *Computers & Security*, 77:578 – 594, 2018.

[13] Browser & platform market share, july 2019. `https://www.w3counter.com/globalstats.php?year=2019&month=7`.