2019 IEEE Security and Privacy Workshops (SPW)

MaxNet: Neural network architecture for continuous detection of malicious activity

Petr Gronát
Avast Software
Prague, Czech Republic
gronat@avast.com

Javier Aldana-Iuit
Avast Software
Prague, Czech Republic
aldana@avast.com

Martin Bálek
Avast Software
Prague, Czech Republic
balek@avast.com

***Abstract*—This paper addresses the detection of malware activity in a running application on the Android system. The detection is based on dynamic analysis and is formulated as a weakly supervised problem. We design an RNN sequential architecture able to continuously detect malicious activity using the proposed max-loss objective. The experiments were performed on a large industrial dataset consisting of 361,265 samples. The results demonstrate the performance of 96.2% true positive rate at 1.6% false positive rate which is superior to the state-of-the-art results. As part of this work, we release the dataset to the public.**

***Index Terms*—Android; malware detection; dynamic analysis; recurrent network**

I. Introduction

Machine learning and especially deep learning has become an extremely useful and interesting topic in cybersecurity in the last few years. In this context, malware detection has received a significant amount of attention in the past years[1, 2, 3]. This paper addresses the problem of detecting Android malware activity on a system in the time domain by looking at behavioral sequences on a large amount of industrial data.

When malware application is being executed on a system, its behavior consists of a number of different activities placed along the time axis, and there is just a subsequence of actions which results in malicious activity. Very often, the malware application behaves as a goodware and at some point of execution, the malicious activity is formed. Hence, the challenging goal is to identify such a subsequence within the whole sequence of events.

Being equipped with this paradigm, we develop a behavioral model that analyses a *dynamic behavior* of the application in the system during execution. We use a sequence of API/function calls generated by the application at the runtime as input and design a recurrent neural network (RNN) architecture enabled to detect malicious activity. Without a loss of generality, this specific work focuses on malware activity detection on Android device, however, it can be applied to other systems. The model has been trained and tested on a large portion of *industrial data* consisting of *361,256 samples* generated on an emulator farm.

Many mobile phone vendors pursue on-device hardware acceleration to provide with better support to these AI frameworks. Therefore we consider deploying an RNN based model directly to a device as one of the security layers to be a viable solution.

Contribution. The contribution of this work is three-fold; (i) we formulate malware detection as a weakly supervised problem, (ii) we propose RNN architecture for identifying the malicious subsequence within a sequence, and (iii) provide a large scale dataset consisting of both malware and benign samples and release it to the public for research purposes. To the best of our knowledge 'maxNet' is the first anti-malware tool that utilizes devoted RNN architecture for detection of malicious activity within a sequence of events.

Paper organization. The rest of the paper is organized as follows; Section IV presents a method overview, a brief description of the features being used in this work can be found in Section V, Section VI describes proposed neural network architecture for malware detection, the training, and evaluation phase, and Section VII details architecture and dataset description, implementation details, and experimental results.

II. Motivation

Increasing usage of smartphones within the last decade comes with the growing prevalence of mobile malware. For instance, according to the G DATA report[4] a new instance of Android malware emerges nearly every ten seconds.

Malware authors use many techniques to evade the detection such as code obfuscation, encryption, including permissions which are not needed by the application, requesting for unwanted hardware, download or update attack in which a benign application updates itself, some of them may bypass offline security checks, e.g. by relying on the so-called droppers, that load the malicious payload after being activated. All such techniques often make it difficult to identify the malware using the static analysis (e.g. exploiting permissions, intent filters, API usage, static call graphs).

It is thus favorable to add another layer of detection based on continuous monitoring of the running system and using modern machine learning tools to detect malicious activity directly on the device. We address this problem by developing a behavioral model amenable of reliable training on a large amount of industrial data. We learn the behavior of an application on runtime where the behavior can be viewed as a

DOI 10.1109/SPW.2019.00018

sequence of actions. A certain subsequence of the actions in a specific order can be responsible for the malicious activity, e.g. accessing SMS code and then logging to your banking account, while the same subsequence in a different order can be benign.

Challenges: There are several scientific challenges to address; (i) how to get behavioral features and introduce the temporally of the running application process, (ii) how to design a sequential model that can process these features, (iii) how to train such a model, and finally, (iv) how to get sufficient amount of labeled data to train the model? In order to address these challenges, we implement an *observer engine* in the Android kernel that produces a sequence of events from the running application process. We develop a recurrent neural network architecture (RNN) called 'maxNet' able to process such a stream of events. The model can observe the activity of the executed application and eventually can block the execution of a malicious activity before harming the user. We cast the malicious activity detection as a weakly supervised task and propose training procedure for the recurrent model. Finally, we collect a dataset of $361,265$ samples and release to the public for research purposes.

III. Related work

A. Malware detection techniques

Malware detection techniques are generally divided into two categories: static and dynamic. In case of static detection, one extracts information from a binary without having the application being executed. Mentioning just a few relevant for Android systems, one can start from creating a unique signatures[5], analyze requested permissions as in [6], do a more detailed analysis of used API calls as in DroidAPIMiner[7], or even model the sequence of API calls as in MaMaDroid[8].

Anomaly-based detection model [9] continuously monitors the different features of the device state such as battery level, CPU usage, network traffic, etc. Measurements are taken during running and are then supplied to an algorithm that classifies them accordingly. CrowDroid[10] and AntiMalDroid[11] are two different anomaly-based tools used for malware detection on Android devices. The first depends on analyzing system calls logs while the latter analyzes the behavior of an application and then generates signatures for malware behavior.

In this paper, we focus on a dynamic analysis where an application is examined by its behavior during execution. Dealing with dynamic analysis always means to solve the problem that not all events in one record can be attributed with malicious behavior. Authors of [12] transform all sequences to feature vectors via calculating relative frequencies of n-long subsequences, reducing the dimensionality by custom feature selection and classify via Support Vector Machines (SVM), while [13] first cluster subsequences based on additional information on CPU and memory usage and then train a Random Forest classifier on each cluster (using the same feature n-gram based feature selection procedure).

B. Sentiment analysis

Sentiment analysis or opinion mining is the computational study of user opinions, sentiments, emotions, appraisals, and other attitudes. The malware detection technique presented in this work has an analogy into the sentiment analysis of a text. Our framework lays in the field of document-level sentiment classification, where the documents are the Android event sequences, under the assumption that documents are opinionated, i.e. are labeled by an expert as positive or negative. Neither subjectivity and sentence-level classification approaches are suitable for our purposes since the basic information unit is the full sequence of events of single application execution without local labeling of sub-sequences.

Existing research has produced numerous techniques for various tasks of sentiment analysis, which include both supervised and unsupervised methods. In the supervised setting, early papers used SVM, Maximum Entropy, Naive Bayes, etc.[14] with different kinds of feature representations like the bag of unigrams and feature combinations. Unsupervised methods include various methods that exploit sentiment lexicons, grammatical analysis, and syntactic patterns, i.e., fixed syntactic phrases[15]. A decade ago deep learning has emerged as a powerful machine learning technique[16] and produced state-of-the-art results in many application domains, ranging from computer vision[17, 18, 19, 20] recognition[21, 22, 23] to NLP[24, 25]. Application of deep learning to sentiment analysis has also become very popular[25, 26] among researchers.

During the training, we are given a sequence of events collected on the emulator and a label of this sequence (malware or benign) transferred from our database of Android applications. In the context of sentiment analysis, each event can be viewed as a word, hence the whole sequence forms a 'text'. The label is analogous to the sentiment of a given text, negative or positive. Each event (or word) can be turned into the one-hot encoded sparse vector and, the one-hot sparse vector is embedded into a low-dimensional dense vector space represented by an embedding matrix. This embedding is trainable and represents a meaningful space where two semantically related events are kept close to each other.

C. Recurrent neural networks

Recurrent Neural Network (RNN)[27] is a class of neural networks whose connections between neurons form a directed cycle. Unlike feed-forward neural networks, RNN can use its internal 'memory' to process a sequence of inputs, which makes it important for processing sequential information. RNN performs the same task for every element of a sequence with each output being dependent on all previous computations, which is like 'remembering' information about what has been processed so far. Theoretically, RNN can make use of the information in arbitrarily long sequences, but in practice, the standard RNN is limited to looking back only a few steps due to the vanishing gradient or exploding gradient problem[28]. Researchers have developed more sophisticated types of RNN to deal with the shortcomings of the standard RNN model, for instance, Bidirectional RNN[29], Deep Bidirectional RNN or Long Short Term Memory (LSTM)[30].

LSTM is a special type of RNN, which is capable of learning long-term dependencies. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for 'remembering' values over arbitrary time intervals. Gated Recurrent Unit (GRU) can be viewed as a simplified version of the LSTM unit. It was introduced by [31], where the authors achieved semantically and syntactically meaningful representation of linguistic phrases. It has been shown that GRUs exhibit better performance on smaller datasets[32]. It is worth mentioning that in the context of Windows malware, [33] uses RNN architecture but only for embedding, not for final classification. Due to the sequential nature of the task, in the experiments, we used both the GRU and the LSTM units.

IV. Method overview

When a software process is labeled as malicious, we typically assign a single malicious label to it. But not everything that the process does is malicious. The process can be represented as a sequence or a stream of events $\vec{e} = (e_t)_{t=0}^{\infty}$. The malicious activity is hidden within a sequence of many benign events. The goal is to train a prediction function $f(\cdot)$ and to find threshold Θ such that

$$f(\vec{e_\tau}) \begin{cases} > \Theta & \text{if } e_\tau \text{ belongs to the malicious subsequence} \\ \leq \Theta & \text{otherwise,} \end{cases} \tag{1}$$

where $\vec{e_\tau} = (e_0, e_1, ..., e_\tau)$ is a sequence of events from the beginning of the process up to time step τ. The inference function (1) is represented by a recurrent neural network (RNN) model that is fed by a stream of events from the sequence $\vec{e}$ one by one, and at each time step the function (1) provides the inference about the current event e_τ belonging to the malicious activity. If this can be done accurately, an interesting corollary is that the malware may be detected at the beginning of the malicious activity.

Since for each stream generated by the process, we only have a binary label (malware or benign) for the entire stream we deal with a weakly-supervised problem as we do not know which portion of the stream belongs to malicious activity.

Our task can be treated as sentiment analysis of a text (for instance user reviews) from the NLP domain. One of the models being utilized in the sentiment analysis is a many-to-one RNN, where the text is fed word by word into the model. The details will be discussed in Section VI-A. However, this approach is not well suited for our problem, hence in Section VI-B, we modify the objective function such that backpropagation is not tied to the last output node, which allows the model to focus its attention to the malicious subsequences.

Finally, it is necessary to find a threshold Θ to achieve the optimal performance of the model. The threshold Θ will be established dynamically during the training procedure accordingly to the target (desired) false positive rate performance. The procedure for establishing the threshold Θ will be described in Section VI-C.

V. Features

To obtain the features from the running process, the *observer engine* has been implemented in the Android kernel. From the running application, the engine gets the behavioral features (events) which are used by the model to distinguish between malicious and benign applications.

The observer engine modified the C++ implementation of Android's Process Management system to intercept Binder transactions in binary format prior to each transaction passing through to the Android kernel binder driver. By reconstructing the fields of the binary payload and extracting the unique integer ID referencing an API (or function call), the pattern and/or frequency of every inter-process communication (IPC) call sent system-wide was counted and stored in a memory structure for processing and analysis. While nearly 50,000 unique binder calls are possible, for optimal performance the observer engine only tracked a subset of 1383 IDs most relevant to the running application picked based on our expert knowledge.

Hence, in total, we gather 1383 distinct API/functional calls, and a stream of integers from 0 to 1382 forms a sequence of events $\vec{e}$ captured within the first 60 seconds of execution of the application. A detailed description of the observer engine can be found in [34].

VI. Neural network architecture

In this section, we first review existing sentiment analysis architecture from the NLP domain, Section VI-A. Second, in Section VI-B, we discuss its modification for malware activity detection by introducing the max-loss objective function. The problem of establishing the optimal threshold Θ is tackled in Section VI-C, and finally, the usage of the model during the evaluation is described in Section VI-D.

A. Sentiment analysis approach

Sentiment analysis problem can be solved by using a many-to-one RNN architecture. The RNN unit has its internal state responsible for 'remembering' information provided by previous inputs. As the network receives one input at the time, at each time step the internal state is updated and can be modified. The prediction is then inferred on top of the RNN output after receiving the last input. The situation is schematically shown in figure 1. The right part illustrates an unrolled representation of the graph depicted in the left, from the time step τ back to time step 0. At each time step, the recurrent block A receives current input event e_t and updates its both hidden state h_t, output $\hat{y}_t$, and passes the hidden state h_t to the next time step. The recurrent block A contains RNN unit (GRU or LSTM) and shares the parameters. More details on the recurrent block A will be provided later in the section VII-B.

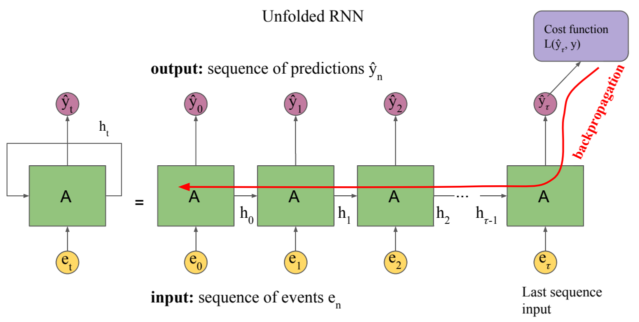


Figure 1: An illustration of an unrolled many-to-one architecture and backpropagation path with a standard objective function as defined by (3).

The way the state evolves with the input depends on the parameters of the RNN unit and the initial state of the unit. More formally it can be written in a form of recurrent function

$$\vec{h_t} = f_\theta(h_{t-1}, e_t) \tag{2}$$

where θ are parameters we wish to learn. As we modify the parameters, the recurrent unit will perform a different behavior. The parameters θ are being trained via backpropagation through time algorithm. For sentiment analysis, the backpropagation is performed after receiving the last input e_τ, hence the objective function can be written as follows

$$L(\hat{y}, y) = \frac{1}{2}||\hat{y}(e_\tau) - y||_{L2} \tag{3}$$

where $\hat{y}$ is the network prediction and y is the target. As shown in figure 1, the backpropagation always starts at the last output node after receiving the last input e_τ.

B. Training phase with max-loss

The approach just described can be applied to malicious activity detection. However, the problem is that the objective function (3) is not well suited for the goal of identifying malicious activity as soon as it occurs. It is rather designed for the case when the inference step is performed after receiving the last output. In fact, we aim at one-to-one architecture, where for each input the network outputs the probability that current event e_t belongs to the malicious activity. To train such a network it is necessary to have a target label for each input event e_t. Such a label would tell us if the current input is a part of a malicious subsequence. However, in practice, it is not feasible to collect such labels. For each sequence, we can only have a weak label telling us if a malicious subsequence (or more subsequences) is present in the whole sequence or not.

It is important to note that at the evaluation phase on a system, for each input in turn, the model outputs prediction, and if the prediction exceeds the threshold Θ, the whole sequence is labeled as malware regardless the future inputs. This weakly supervised problem can be tackled by relaxing the loss function (3) as follows

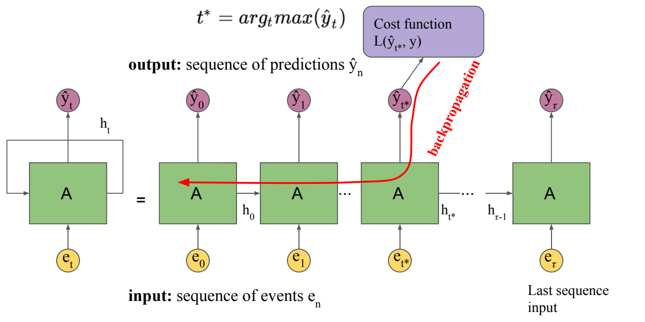

$t^* = arg_t max(\hat{y}_t)$
Cost function L(ŷ_{t*}, y)
output: sequence of predictions ŷ_n
backpropagation
h_t
A
=
h_0
h_{t*}
h_{τ-1}
input: sequence of events e_n
Last sequence input

Figure 2: An illustration the backpropagation path with a propsed objective function as defined by (4). The backpropagation starts at the output node with the time stamp t^*.

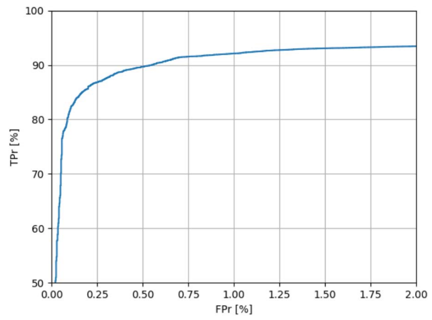


Figure 3: ROC curve using our model on AV-test data[35] from January 2018.

$$L(\hat{y}, y) = \frac{1}{2}||\max_t \hat{y}(e_t) - y||_{L2} \tag{4}$$

where the term $\max_t \hat{y}(e_t)$ represents the maximal output of the recurrent unit for the whole sequence along the training.

The consequence of the relaxation (4) is that during the training, the backpropagation starts at the node where the maximal output of the recurrent unit happened. The situation is depicted in figure 2. For a particular sequence, the network parameters θ are being modified during the training. Notice that in each epoch, the maximum appears on a different node. The intuitive corollary of the objective (4) is that the network is forced to pay attention to the malicious subsequences as soon as it appears.

C. Establishing threshold during training

The last problem to address is how to establish the threshold Θ. For a malware detection engine in industrial applications, it is critical to achieve very low false positive detection rate (FPr) while achieving a sufficiently high true positive detection rate (TPr). The tradeoff between FPr and TPr at various threshold settings Θ can be expressed by the receiver operating characteristic (ROC) curve. For illustration, figure 3 shows the ROC curve on AV-test data[35] from January 2018.

The challenge is how to pick the threshold Θ such that for unseen data in the wild the FPr is close to a target

FPr. We define the target FPr as desired FPr performance on unseen data. The distribution of the unseen data, however, is different from the data seen by the model during the training. Hence its ROC curve differs from the one on training data or validation data. If the ROC curve on unseen data was known, one could pick a threshold Θ corresponding to the target FPr performance. Unfortunately, this is never the case in practice. To address the issue, we propose establishing the threshold Θ during training from the test data for every training epoch. The procedure is captured in Algorithm 1.

Algorithm 1 Establishing threshold Θ

```
    Input: Trained model and Validation batches
    Output: Optimal threshold $\Theta^*$ for given model
1: procedure ESTABLISHING THRESHOLD $\Theta^*$
2:     targetFPr ← target false positive performance, e.g. 0.01
3:     TPrs ← empty list
4:     $\Theta s$ ← empty list
5:     for each validation batch do
6:         construct the ROC curve
7:         $\Theta'$ ← threshold corresponding to the targetFPr
8:         TPr' ← TPr performance at given $\Theta'$
9:         interpolate $\Theta$' and $TPr'$ if necessary
10:        $\Theta s.add(\Theta')$
11:        $TPrs.add(TPr')$
12:    $\Theta_{mean} \leftarrow mean(\Theta s)$
13:    $\Theta_{std} \leftarrow std(\Theta s)$
14:    $TPr_{mean} \leftarrow mean(TPrs)$
15:    $TPr_{std} \leftarrow std(TPrs)$
16:    $\Theta^* = \Theta_{mean} + k \cdot \Theta_{std}$
```

Consider a single epoch during the training. After updating the model, for each batch of test data, we construct the ROC and compute a threshold Θ such that the FPr upon using this threshold corresponds to the target FPr.

The Θs array (record of all Θ values) is assumed to be generated by a normal distribution.

After evaluating all test data batches, we compute the mean and standard deviation of the recorded values in Θs array, and the optimal threshold for the current epoch is computed as

$$\Theta^* = \mu_\Theta + k * \sqrt{\sigma_\Theta} \tag{5}$$

where μ_Θ and $\sqrt{\sigma_\Theta}$ denote mean and standard deviation of values in the list Θs and k is a hyperparameter. In our scenario, we set $k = 2$ to cover 93% of the threshold population. Finally, we also keep track of the mean and variance of the corresponding TPr in order to improve the early stopping policy discussed later in Section VII-B.

D. Evaluation phase

During the evaluation, the trained recurrent model along with the corresponding threshold Θ discussed above is deployed on a device or a backend. When the unknown application is run, the model is fed by events produced by the observer engine introduced in Section V. After receiving an input event, the model outputs an inference and if it is greater than Θ the process is labeled as malware regardless of the following events.

VII. Experiments

A. Dataset description

We took $361,265$ Android applications, and for each, we generated a behavioral sequence on our emulator farm. The sequences were assigned with the label and timestamp of the application from our industrial database. The dataset covers the time period from January 2012 up to January 2018.

Hence, each application in the dataset is represented by (i) an integer sequence $\vec{e}$ representing events acquired by the observer engine described in Section V, (ii) a label transferred from the database and (iii) Unix timestamp representing when the application appeared for the first time. In total, the dataset contains $100,595$ malware and $260,670$ benign samples and there exist 1383 distinct events within all sequences.

B. Architecture and training details

We experimented with different setups. We used different embedding matrix sizes, single or double layers of LSTM or GRU units followed by one to three fully-connected (FC) layers and tried different regularization (L2, L1, batch-norm, dropout) functions and optimizers (ADAM, RMSprop, Nestorov momentum). The best performing and least complex model is described next, the architecture is shown in figure 4.

a) Hyper parameters: Each input event e_t is encoded into a one-hot vector, which is passed to the embedding matrix of size 1384×16, since in the dataset there exist 1383 distinct features and one extra token is used for padding. Each event is embedded into 16-dimensional dense vector which is passed into the GRU layer of size 256 which is followed by the FC layer of size 128 followed by dropout and combined into a single neuron. The output is finally passed to a sigmoid function to produce the prediction $\hat{y}$.

During training, we use a *keep probability* of 0.7 for the dropout layer and ReLU nonlinearities. We observed that stacking GRU units or adding optional FC layers did not improve the results. The batch size of 512 was used throughout the experiment and the best results were achieved with the RMSProp optimizer with the learning rate of $2.3 \cdot 10^{-3}$. The learning rate is decayed by a factor of 0.6 if there is no improvement in the training loss for two epochs.

The early stopping policy is set such that the training is terminated if the estimated TPr (Section VI-C) on the validation set is not improving for 8 epochs. The convergence to the best model was typically achieved in 30 epochs. The constant k in formula (5) was set to 2.0 for the model threshold estimation.

b) Data split: The standard practice is to split the dataset in training-test sets by selecting the samples randomly. For the case of malware, our industrial experience has proved that this is not a good strategy. It is likely that a randomly selected training set could contain modified copies of the same malware re-packed as in the testing set. This would lead to artificially

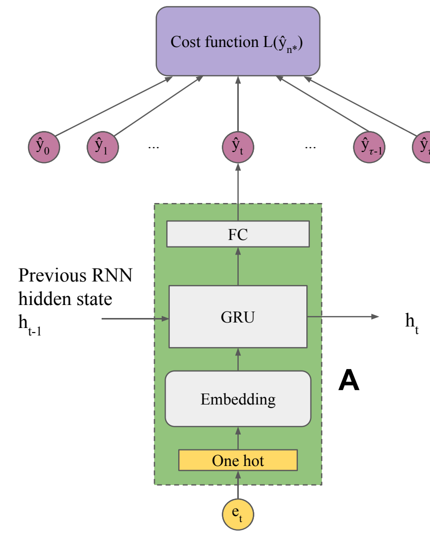


Figure 4: The max-loss RNN, each input e is one-hot encoded and is passed to trainable embedding layer. Its output is passed to the RNN unit and then to the FC layer with a single prediction node. The objective function L identifies the node producing the maximal output and the loss is computed.

good performance, but the classifier would fail to generalize on new malware samples.

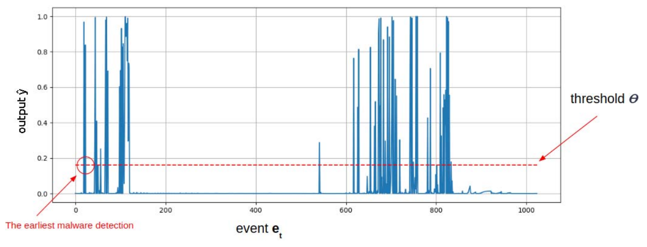
To prevent this problem, we split our data timewise, such that the most recent samples are in the testing set, but not in the training set. Our dataset was split to training, validation and testing data as follows: all the events sequences were ranked w.r.t. the timestamps and split in a ratio of 90% / 10%. The latter is being used as a test set. The bigger chunk is further randomly shuffled and 10% of it is used for validation and the rest for training.

To summarize, $36.3k$ newest samples belongs to the test set, $293.7k$ samples are used for training the model, and $32.6k$ samples are used for validation. For chosen batch size and GPU memory constraints, each sequence is padded or cropped to the length of 1024 if necessary.

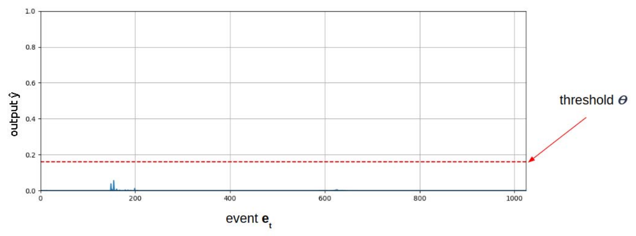
C. Results

A comparison of the proposed method with several state-of-the-art results is provided by table I where we compare with both dynamic and static methods. The maxNet architecture is superior to most of the competitor's results. It significantly outperforms the other techniques in terms of the FPr (1.6%) which is critical for industrial applications. Moreover, the TPr performance is much better compared to with performances of others and is marginal to Amos et al.[36] who experimented with a smaller dataset and reported TPr of 97.3% but obtained FPr of 31.0%. As shown in table I, our method achieved the highest F1-score. A similar score was achieved by Morales-Ortega et al.[37] who also reported 0.96.

It is worth noticing that our dataset is significantly bigger than datasets of our competitors shown in table I (for some by a factor of 100) and and consists of dozens of families [34].



(a) malware



(b) benign

Figure 5: Examples of the network output at each time step. Output from malware sample (top) exceeds threshold Θ during malicious activity. The output from a benign sample (bottom) never exceeds threshold Θ.

Method	TPr	FPr	#malware	#benign	F1 score	type
maxNet (ours)	96.2	**1.6**	**100,959**	**260,670**	**0.96**	D
Canfora et al. [12]	95.9	4.2	1000	1000	0.95	D
MaMaDroid [8]	95.5	6.4	2974	2568	0.95	D
Ferrante et al. [13]	70.6	38.0	1523	1709	0.66	D
Amos et al. [36]	**97.3**	31.0	1,330	408	0.94	S
Morales-Ortega et al. [37]	96.2	3.7	1,377	1,377	0.96	S
Kurniawan et al. [38]	85.3	14.7	200	200	0.85	S
Ahmadi et al. [39]	72.0	7.5	9,664	10,058	0.80	S

Table I: Comparison of maxNet with state-of-the-art results. Type: S-static, D-dynamic

The train and validation set came from a period from January 2012 to September 2017 while the test set contains samples that appear within three months after, hence the test set contains the unseen type of malware. The results show that the proposed method is robust and capable of catching a *zero-day* malware.

We also tested our model *in-the-wild* on the challenging industrial AV-test benchmark[35] with unseen data. As shown in figure 3 the model achieved 94% TPr at 1.6% FPr. Despite the TPr is slightly lower compared to table I, it demonstrates that our model generalizes reasonably well.

Finally, the figure 5 shows a prediction of our model for malware and the benign application input sequence of the length of 1024 events. In the subfigure 5a the output of the network exceeds the threshold Θ at the 20-th input event. The malware detection engine can block the malicious process execution at this point. On the other hand, as shown in subfigure 5b, our model does not trigger with a benign sample, whose threshold never exceeds Θ.

VIII. Conclusion

We formulate malware detection as a weakly supervised problem and design a sequential RNN model capable of detecting malicious activity within the process as soon as it happens. It was shown how to train this model by a relaxation of the objective function from many-to-one RNN architecture. As part of this work, we release a public dataset consisting of $361k$ samples[34]. The experiments on this dataset demonstrate the performance of 96.2% true positive rate at 1.6% false positive rate which is superior to the state-of-the-art results.

Acknowledgment

We thank Rajarshi Gupta for driving and supporting the project, to all team members who did the engineering work on feature acquisition: Pablo Sole, Hiram Lew, Vladislav Iliushin, Filip Havlicek.

References

[1] B. Amos, H. A. Turner, and J. White, "Applying machine learning classifiers to dynamic android malware detection at scale.," in *IWCMC*, 2013.

[2] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of android malware in your pocket.," in *NDSS*, 2014.

[3] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. A. Porras, "DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *ESORICS*, 2014.

[4] C. Lueg, "8,400 new Android malware samples every day." https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day, 2017. (Accessed on 04/09/2018).

[5] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *SIGSOFT*, 2014.

[6] A. P. Felt, D. Song, D. Wagner, and S. Hanna, "Android permissions demystified," in *CCS*, 2012.

[7] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in android.," in *SecureComm*, 2013.

[8] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting android malware by building markov chains of behavioral models," *arXiv:1612.04433*, 2016.

[9] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""Andromaly": A behavioral malware detection framework for android devices," *JIIS*, 2012.

[10] "Strace." https://sourceforge.net/projects/strace/.

[11] M. Zhao, F. Ge, T. Zhang, and Z. Yuan, "AntiMalDroid: An efficient SVM-based malware detection framework for android.," in *ICICA*, 2011.

[12] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Detecting android malware using sequences of system calls," in *DeMobile*, 2015.

[13] A. Ferrante, E. Medvet, F. Mercaldo, J. Milosevic, and C. A. Visaggio, "Spotting the malicious moment: Characterizing malware behavior using dynamic features," in *ARES*, 2016.

[14] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up?: Sentiment classification using machine learning techniques," in *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*, EMNLP '02, (Stroudsburg, PA, USA), pp. 79–86, Association for Computational Linguistics, 2002.

[15] P. D. Turney, "Thumbs up or thumbs down?: Semantic orientation applied to unsupervised classification of reviews," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, (Stroudsburg, PA, USA), pp. 417–424, Association for Computational Linguistics, 2002.

[16] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *AIStat*, 2011.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.

[19] R. Zhang, J.-Y. Zhu, P. Isola, X. Geng, A. S. Lin, T. Yu, and A. A. Efros, "Real-time user-guided image colorization with learned deep priors," *TOG*, 2017.

[20] R. Arandjelovic, P. Gronat, A. Torii, T. Pajdla, and J. Sivic, "NetVLAD: CNN architecture for weakly supervised place recognition," *PAMI*, 2017.

[21] J. K. Chorowski, D. Bahdanau, D. Serdyuk, K. Cho, and Y. Bengio, "Attention-based models for speech recognition," in *NIPS*, 2015.

[22] S. Upadhyay, M. Faruqui, G. Tur, D. Hakkani-Tur, and L. Heck, "(almost) zero-shot cross-lingual spoken language understanding," in *ICASSP*, 2018.

[23] H. Sak, A. W. Senior, K. Rao, and F. Beaufays, "Fast and accurate recurrent neural network acoustic models for speech recognition," *CoRR*, 2015.

[24] J. Li, W. Monroe, T. Shi, S. Jean, A. Ritter, and D. Jurafsky, "Adversarial learning for neural dialogue generation," in *EMNLP*, 2017.

[25] M.-T. Luong, Q. V. Le, I. Sutskever, O. Vinyals, and L. Kaiser, "Multi-task sequence to sequence learning," in *ICLR*, 2016.

[26] L. Dong, F. Wei, C. Tan, D. Tang, M. Zhou, and K. Xu, "Adaptive recursive neural network for target-dependent twitter sentiment classification," in *ACL*, 2014.

[27] J. L. Elman, "Finding structure in time," *COGNITIVE SCIENCE*, 1990.

[28] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE NN*, 1994.

[29] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE SP*, 1997.

[30] S. Hochreither and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, 1997.

[31] K. Cho, B. van Merrienboer, . Glehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translation.,” in *EMNLP*, 2014.

[32] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” 2014.

[33] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, “Malware detection with deep neural network using process behavior,” in *COMPSAC*, 2016.

[34] Avast, “Avast research page.” http://public.avast.com/research/. (Accessed on 12/21/2018).

[35] “Av-test — antivirus & security software & antimalware reviews.” https://www.av-test.org/en/. (Accessed on 04/24/2018).

[36] B. Amos, H. A. Turner, and J. White, “Applying machine learning classifiers to dynamic android malware detection at scale.,” in *IWCMC*, 2013.

[37] S. Morales-Ortega, P. J. Escamilla-Ambrosio, A. Rodriguez-Mota, and L. D. Coronado-De-Alba, “Native malware detection in smartphones with android os using static analysis, feature selection and ensemble classifiers.,” in *MALWARE*, 2016.

[38] H. Kurniawan, Y. Rosmansyah, and B. Dabarsyah, “Android anomaly detection system using machine learning classification,” in *ICELTICs*, 2015.

[39] M. Ahmadi, A. Sotgiu, and G. Giacinto, “IntelliAV: Toward the Feasibility of Building Intelligent Anti-malware on Android Devices,” in *CD-MAKE*, 2017.