Generic Black-Box End-to-End Attack against RNNs and Other API Calls Based Malware Classifiers

Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici
Software and Information Systems Engineering Department, Ben Gurion University of Israel
Email: ishairos@post.bgu.ac.il

***Abstract*—Deep neural networks are being used to solve complex classification problems, in which other machine learning classifiers, such as SVM, fall short. Recurrent Neural Networks (RNNs) have been used for tasks that involves sequential inputs, like speech to text. In the cyber security domain, RNNs based on API calls have been able to classify unsigned malware better than other classifiers. In this paper we present a black-box attack against RNNs, focusing on finding adversarial API call sequences that would be misclassified by a RNN without affecting the malware functionality. We also show that the this attack is effective against many classifiers, due-to the transferability principle between RNN variants, feed-forward DNNs and state-of-the-art traditional machine learning classifiers. Finally, we introduce the *transferability by transitivity* principle, causing an attack against generalized classifier like RNN variants to be transferable to less generalized classifiers like feed-forward DNNs. We conclude by discussing possible defense mechanisms.**

I. Introduction

In recent years, deep neural networks (DNNs) have outperformed other machine learning algorithms in many complex problem domains ([23]), such as image recognition, etc.

When the input is a sequence, Recurrent Neural Networks (RNNs) and their variants (LSTM, GRU, etc.) outperforms. This is true for both classification tasks, such as video classification and machine translation, and for generative models, such as handwriting text generation. This success caused cyber security experts to use RNNs for malware classification tasks. Application Programming Interface (API) calls, often used to characterize the behavior of a program, seem like an obvious choice. Since the sequence of API calls gives them the context and proper meaning, RNNs are the natural choice - and indeed gives state of the art performance ([33]).

As reviewed in [27], machine learning classifiers and algorithms are vulnerable to different kinds of attacks, aiming to undermine the system's integrity, availability, etc. Adversarial examples are originally correctly classified input, which are perturbed to be assigned a different label. In this paper, the RNN is a binary classifier between malicious and benign API call sequences. The adversarial example would therefore be a malicious API call sequence, originally correctly classified by the RNN, but after the perturbation (which doesn't affect the malware functionality) - the sequence is classified by the RNN as benign (a form of evasion attack).

Our contributions in this paper are:

1) We implement a novel end-to-end method to generate adversarial examples for a RNN malware classifier. Unlike previous papers, we focus on the cyber security domain and implement a method that would preserve the functionality of the malware, after the perturbation, using *mimicry attack*.
2) We implement a black-box attack, that is, the adversary don't know the attacked RNN model architecture and hyper-parameters, a more realistic attack scenario than a white-box attack. We test our implementation against various types of machine learning malware classifiers trained on a dataset of 500,000 malware and benign samples. This, again, shows that the proposed attack is feasible and an immediate threat.
3) We focus on the principle of transferability in RNN variants. While this subject was covered in the past in the context of DNNs, to our knowledge, this is the first time it is being evaluated in the context of RNNs, proving that the proposed attack is generic not just against specific RNN variant, but also against other RNN variants (like LSTM, GRU, etc.), against feed-forward DNNs (including CNNs) and against traditional machine learning classifiers such as SVM and random forest.
4) We show that the proposed attack is effective against the largest number of classifiers ever reviewed in a single paper to the best of our knowledge, including architectures that were not compared before for API calls based malware classification: RNN, LSTM, GRU, their bidirectional and deep variants, feed-forward DNN, 1D CNN, SVM, random forest, logistic regression and decision tree classifiers, making the proposed attack the first effective one against all state-of-the-art API calls based classifiers.

The rest of the paper is structured as follows: Section II contains the relevant related work. Section III specifies our attack methodology, including surrogate model training and adversarial examples generation. Section IV contains our experimental results, including our attack performance. Section V contains possible defense mechanisms, our conclusions and future work.

II. Background and Related Work

A. Deep Neural Networks (DNNs)

Neural Networks are a class of machine learning models made up of layers of neurons—elementary computing units.

A neuron takes an n-dimensional feature vector $\mathbf{x} = [x_1, x_2...x_n]$ from the input or the lower-level neuron, and outputs a numerical output $\mathbf{y} = [y_1, y_2...y_m]$, such that

$$y_j = \phi(\sum_{i=1}^{n} w_{ji}x_i + b_j) \tag{1}$$

to the neurons in higher layers or the output layer. For the neuron j, y_j is the output, b_j is the bias term, while w_{ji} are the elements of a layer's weight matrix. The function ϕ is the non-linear activation function, such as $sigmoid()$, which determines the neuron's output. The activation function introduces non-linearities to the neural network model. Otherwise, the network remains a linear transformation of its input signals. Part of the success of DNNs is attributed to those multi-layers of non-linear correlations between features, which aren't available in popular traditional machine-learning classifiers, such as SVM, which has at most a single such layer, using the kernel trick.

A group of m neurons forms a hidden layer which outputs a feature vector $\mathbf{y}$. Each hidden layer takes the previous layer's output vector as the input feature vector and calculates a new feature vector for the layer above it:

$$\boldsymbol{y}_l = \phi(\boldsymbol{W}_l\boldsymbol{y}_{l-1} + \boldsymbol{b}_l) \tag{2}$$

, where $\boldsymbol{y}_l$, $\boldsymbol{W}_l$ and $\boldsymbol{b}_l$ are the output feature vector, the weight matrix, and the bias of the l-th layer. Proceeding from the input layer, each subsequent higher hidden layer automatically learns a more complex and abstract feature representation which captures higher-level structure. Part of the success of DNNs in complex domains, such as computer vision, is often attributed to the ability to use raw input as features and use this characteristic, termed *representation-learning* , which replace manual feature engineering.

1) Convolutional Neural Networks (CNNs): Let $\boldsymbol{x_i} \in \mathbb{R}^{\mathrm{k}}$ be the k-dimensional vector corresponding to the i-th element in the sequence. A sequence of length n (padded where necessary) is represented as: $\boldsymbol{x}[0 : n-1] = \boldsymbol{x}[0] \perp \boldsymbol{x}[1] \perp \boldsymbol{x}[n-1]$, where $\perp$ is is the concatenation operator. In general, let $\boldsymbol{x}[i : i+j]$ refer to the concatenation of words $\boldsymbol{x}[i], \boldsymbol{x}[i+1], ..., \boldsymbol{x}[i+j]$. A convolution operation involves a filter $\boldsymbol{w} \in \mathbb{R}^{\mathrm{hk}}$, which is applied to a window of h elements to produce a new feature. For example, a feature c_i is generated from a window of words $\boldsymbol{x}[i : i+h-1]$ by:

$$c_i = \phi(\boldsymbol{W}\boldsymbol{x}[i : i+h] + b) \tag{3}$$

Here $b \in \mathbb{R}$ is the bias term and ϕ is the activation function. This filter is applied to each possible window of elements in the sequence $\{\boldsymbol{x}[0 : h-1], \boldsymbol{x}[1 : h], ..., \boldsymbol{x}[n-h : n-1]\}$ to produce a *feature map*: $\boldsymbol{c} = [c_0, c_1, ..., c_{n-h}]$, with $\boldsymbol{c} \in \mathbb{R}^{\mathrm{n\text{-}h+1}}$. We then apply a max over time pooling operation over the feature map and take the maximum value: $\hat{c} = \max(\boldsymbol{c})$ as the feature corresponding to this particular filter. The idea is to capture the most important feature - one with the highest value - for each feature map.

We have described the process by which one feature is extracted from one filter. The model uses multiple filters (with varying window sizes) to obtain multiple features. These features form the penultimate layer and are passed to a fully connected soft-max layer whose output is the probability distribution over labels.

CNNs have two main differences, in comparison to fully-connected DNNs: 1) CNNs exploit spatial locality by enforcing a local connectivity pattern between neurons of adjacent layers. The architecture thus ensures that the learnt "filters" produce the strongest response to a spatially local input pattern. Stacking many such layers leads to non-linear "filters" that become increasingly "global". This allows the network to first create representations of small parts of the input, then from them assemble representations of larger areas. 2) In CNNs, each filter is replicated across the entire input. These replicated units share the same parameterization (weight vector and bias) and form a feature map. This means that all the neurons in a given convolutional layer respond to the same feature (within their specific response field). Replicating units in this way allows for features to be detected regardless of their position in the input, thus constituting the property of translation invariance. This property is important in both vision problems and with sequence input, such as API calls trace.

B. Recurrent Neural Networks (RNNs)

A limitation of Neural Networks is that they accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g. probabilities of different classes). Recurrent neural networks allow us to operate over sequences of vectors in the input, the output, or both. In-order to do that, the RNN keeps a hidden state vector, the context of the sequence, which is combined with the current input to generate the RNN's output.

Given an input sequence $[\boldsymbol{x}_1, \boldsymbol{x}_2...\boldsymbol{x}_T]$, RNN computes the hidden vector sequence$[\boldsymbol{h}_1, \boldsymbol{h}_2...\boldsymbol{h}_T]$ and the output vector sequence $[\boldsymbol{y}_1, \boldsymbol{y}_2...\boldsymbol{y}_T]$ by iterating the following equations from t = 1 to T:

$$\boldsymbol{h}_t = \phi(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{x}_{t-1} + \boldsymbol{b}_h) \tag{4}$$

$$\boldsymbol{y}_t = \boldsymbol{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_o \tag{5}$$

, where the $\mathbf{W}$ terms denote weight matrices (e.g. $\boldsymbol{W}_{\mathbf{ih}}$ is the input-hidden weight matrix), the $\boldsymbol{b}$ terms denote bias vectors (e.g. $\boldsymbol{b}_h$ is hidden bias vector) and ϕ is usually an element-wise application of an activation function. DNNs without a hidden state, as specified in Eq. (4), reduces Eq. (5) to the private case of Eq. (2), known as *feed-forward networks*.

1) Long Short-Term Memory (LSTM): Standard RNNs suffer from both exploding and vanishing gradients. Both problems are caused by the RNN's iterative nature, whose gradient is essentially equal to the recurrent weight matrix raised to a high power. These iterated matrix powers cause the gradient to grow or to shrink at a rate that is exponential in the number of time steps. The vanishing gradient problem does not necessarily cause the gradient itself to be small; the gradient's component in directions that correspond to long-term dependencies might be small, while the gradient's component in directions that correspond to short-term dependencies is large. As a result, RNNs can easily learn the short-term but not the long-term dependencies. That is, a conventional RNN might have problems predicting the last word in: "I grew up in France... I speak fluent French." if the space between the sentences is large.

The Long Short-Term Memory (LSTM) architecture ([16]), which uses purpose-built memory cells to store information, is better at finding and exploiting long range context than conventional RNNs. The LSTM's main idea is that, instead of computing $\boldsymbol{h}_t$ from $\boldsymbol{h}_{t-1}$ directly with a matrix-vector product followed by a nonlinearity, the LSTM directly computes $\triangle \boldsymbol{h}_t$, which is then added to $\boldsymbol{h}_{t-1}$ to obtain $\boldsymbol{h}_t$.This implies that the gradient of the long-term dependencies cannot vanish.

2) Gated Recurrent Unit (GRU): [4] introduced the Gated Recurrent Unit (GRU), which is an architecture that is similar to the LSTM, but reduces the gating signals from three (in the LSTM model: input, forget and output) to two (and the associated parameters). The two gates are called an update gate and a reset gate.

Some papers show that that GRU RNN is comparable to, or even outperforms, the LSTM in many cases ([5]), while using less training time.

3) Bidirectional Recurrent Neural Networks (BRNNs): One shortcoming of conventional RNNs is that they are only able to make use of previous context. It is often the case that for malware events the most informative part of a sequence occurs at the beginning of the sequence and may be forgotten by standard recurrent models. Bidirectional RNNs ([37]) overcome this issue by processing the data in both directions with two separate hidden layers, which are then fed forwards to the same output layer. A BRNN computes the forward hidden sequence $\overrightarrow{\boldsymbol{h}}_t$, the backward hidden sequence $\overleftarrow{\boldsymbol{h}}_t$ and the output sequence $\boldsymbol{y}_t$ by iterating the backward layer from $t = T$ to 1, the forward layer from $t = 1$ to T and then updating the output layer:

$$\overrightarrow{\boldsymbol{h}}_t = \phi(\boldsymbol{W}_{x\overrightarrow{h}}\boldsymbol{x}_t + \boldsymbol{W}_{\overrightarrow{h}\overrightarrow{h}}\overrightarrow{\boldsymbol{h}}_{t-1} + \boldsymbol{b}_{\overrightarrow{h}}) \tag{6}$$

$$\overleftarrow{\boldsymbol{h}}_t = \phi(\boldsymbol{W}_{x\overleftarrow{h}}\boldsymbol{x}_t + \boldsymbol{W}_{\overleftarrow{h}\overleftarrow{h}}\overleftarrow{\boldsymbol{h}}_{t+1} + \boldsymbol{b}_{\overleftarrow{h}}) \tag{7}$$

$$\boldsymbol{y}_t = \boldsymbol{W}_{\overrightarrow{h}y}\overrightarrow{\boldsymbol{h}}_t + \boldsymbol{W}_{\overleftarrow{h}y}\overleftarrow{\boldsymbol{h}}_t + \boldsymbol{b}_o \tag{8}$$

Combing BRNNs with LSTM gives bidirectional LSTM ([12]), which can access long-range context in both input directions.

C. Dynamic Analysis Machine Learning Malware Classifiers

Malware (malicious software) numbers are consistently on the rise. Machine learning malware classifiers, where the model is trained on features extracted from the analyzed file, have two main benefits: 1) The fact that the classifier is trained automatically on the malware samples saves costs and time compare to manually signing new malware variants. 2) They are based on features and not on a fingerprint of a specific and exact file (e.g. file's hash), so they can better handle new threads that use the same features (assuming we are using features indicative of maliciousness).

Malware classifiers can use either static features, gathered without running the code (e.g. byte-sequence (n-grams), strings or structural features of the inspected code) or dynamic features (e.g. CPU usage), collected during the inspected code run. An up-to-date review of the work done in those domains can be found in [21].

While using static analysis has a performance advantage, it has a main disadvantage: Since the code isn't being run, it might not reveal its "true features". For example, if one looks for specific strings in the file, one might not be able to catch polymorphic malware, in which those features are either encrypted or packed and decrypted only during run-time, by a specific bootstrap code. Other limitations of static analysis and techniques to counter it appear in [26]. The most prominent dynamic features are the sequences of API calls and especially those made to the OS, termed system calls. Those are harder to obfuscate during run-time without harming the functionality of the code.

1) Traditional Machine Learning Malware Classifiers: The use of system calls to detect abnormal software behavior was shown in [9]. System call pairs (n-grams of size 2) from test traces were compared against those in the normal profile. Any system call pair not present in the normal profile is called a mismatch. If the number of system calls with mismatches within their window in any given time frame exceeded a certain threshold, an intrusion was reported.

Extensive work has been done on this domain, such as the usage of code-flow graphs and data taint graphs with various classifiers types. A survey of the different features (e.g. monitoring API calls, arguments and information flow) and tools used for dynamic analysis assisted malware classification was conducted in [7].

2) Deep Neural Networks Malware Classifiers: DNNs were used as malware classifiers using either static features, or dynamic ones, outperforming state-of-the-art traditional machine learning classifiers such as SVM ([15], [22]). The rationale for this was explained in section II-A.

In [36], the static features used by a DNN classifier were bins of byte value-entropy (to account for packed files), hash values of the PE imports strings, numerical values from the PE header and strings in file hash values. In [15], API calls were used as static features for a stacked auto-encoder classifier, by extracting them from the file using a PE parser.

In [33], both RNNs and Echo State Networks were used to classify files using API call sequences as dynamic features.

Since directly classify the files is not efficient (there is only a single bit of information for every sequence), a recurrent model was trained to predict next API call, and the hidden state of the model (that en- codes the history of past events) as the fixed-length feature vector that is given to a separate classifier (logistic regression or multilayer perceptron). [2] Compared RNN, LSTM and GRU architectures and found LSTM to outperformed other architectures. A comparison was also made to a convolutional neural network (CNN), proven that the sequence based RNNs performs better. However, the authors used a 2D CNN, regarding each API call as a single character in a sentence. In contrast, we used a 1D CNN, thus omitting artificial spatial correlations between distant API calls, making the proposed architecture more suited to the task and perform better.

Static and dynamic features can be used by the same classifier. For instance, null-terminated patterns observed in the process' memory, tri-grams of system API calls, and distinct combinations of a single system API call and one input parameter were used by a DNN classifier in [20]. The same features were used with feed-forward networks in [6].

Feed-Forward DNNs and RNNs are sometimes combined to leverage the benefit of both. In [22], a CNN was used as a preprocessing phase to an LSTM based on system calls, for feature extraction leveraging the spatially local correlation in the system calls sequences.

D. Evasion Attacks

Attacks against machine learning classifiers, refereed as adversarial machine learning, come in two main phases: 1) During the training of the model. For instance, a *poisoning attack* can be performed by inserting adversarial crafted samples to the training set as part of the baseline training phase of an anomaly detection classifier. 2) During the prediction phase. For example, *evasion attack* involves modifying the analyzed sample's features to evade the model. We would focus only on the latter - especially on the ability to perturbed inputs to be mis-classified by the model. Such inputs are called *adversarial examples* ([40]). An extensive review of other aspects can be found in [27].

Attacks also varies by the knowledge of the adversary about the classifier. *Black-box attack* ([39]) requires no knowledge about the model beyond the ability to query it as a black-box, that-is, inserting an input vector and get the output classification - the *oracle model*. In a *white-box attack*, the adversary has various degree of knowledge about the model architecture and hyper-parameters. Such knowledge can be gained either through internal knowledge or using a staged attack to reverse engineer the model beforehand ([41]). While white-box attacks tend to be more efficient than black-box attacks ([34]), they rely on assumption that are usually not practical.

1) The Transferability Property: Most black-box attacks rely on the concept of *adversarial example transferability*, presented in [40]: adversarial examples crafted to be mis-classified by one model are likely to be misclassified by a different model, either. This transferability property holds even when models are trained on different datasets. This means that the adversary can train a *surrogate model*, which has similar decision boundaries as the original model and perform a white-box attack on it. An adversarial examples that successfully fool the surrogate model would most likely fool the original model, either ([29]). [32] studied the transferability between DNNs and other models such as decision tree and SVM. [24] conducted a study of the transferability over large models and a large scale dataset, showing that while transferable non-targeted adversarial examples are easy to find, targeted adversarial examples rarely transfer with their target labels.

To the best of our knowledge, this paper is the first to explore transferability in RNN models, including conventional RNN, LSTM, GRU and their bidirectional variants, and between them and feed-forward networks and traditional machine learning classifiers, and the first one to explain why attacks should be made against the most generalized classifier, to leverage *transferability by transitivity* (Definition 1).

2) Mimicry Attacks and Other Evasion Attacks in Traditional Machine Learning: [42] deals with *mimicry attacks*, where an attacker is able to code a malicious exploit that mimics the system calls trace of benign code, thus evading detection. Several methods were presented: 1) Make benign system calls generate malicious behavior by modifying the system calls parameters. This works since most IDSs ignore the system call parameters. 2) Adding semantic *no-ops* - system calls with no effect, or whose effect is irrelevant, e.g.: opening a non-existent file. The authors showed that almost every system call can be no-op-ed and thus the attacker can add any needed no-op system call to achieve a benign system call sequence. 3) *Equivalent attacks* – Using a different system call sequence to achieve the same (malicious) effect.

This article has not implemented a generic end-to-end attack and focused on a single classifier type.

[39] implemented a white-box evasion attack for PDFRate, a random forest classifier for static analysis of malicious PDF files, by using either a mimicry attack of adding features to the malicious PDF to make it "feature-wise similar" to a benign sample, or by creating a SVM representation of the classifier and subvert it using the same method mentioned in [3], assuming the availability of surrogate data to the adversary.

[44] used a genetic algorithm to generate an adversarial PDF sample that evades a random forest and SVM classifiers used for malware detection. The fitness of the genetic variants obtained by mutation was defined in terms of the black-box model's class probability predictions. In contrast, our proposed attack was verified against many classifier types, and is less computationally expensive.

3) DNN Adversarial Examples: [40] and [3] both formalize the search for adversarial examples as a similar minimization problem:

$$\arg_{\boldsymbol{r}} \min f(\boldsymbol{x}+\boldsymbol{r}) \neq f(\boldsymbol{x}) \; s.t. \; \boldsymbol{x}+\boldsymbol{r} \in \boldsymbol{D} \tag{9}$$

The input **x**, correctly classified by f, is perturbed with **r** such that the resulting adversarial example **x + r** remains in the input domain **D**, but is assigned a different label than **x**. To solve Eq. (9), we need to transform the constraint $f(\boldsymbol{x}+\boldsymbol{r}) \neq f(\boldsymbol{x})$ to into an optimizable formulation. Then we can easily use the Lagrangian multiplier to solve it. To do this, we define a loss function $Loss()$, to quantify this constraint. This loss function can be the same with the training loss, or it can be chosen differently, such as hinge loss or cross entropy loss.

When dealing with malware classification tasks between malicious (f(**x**)=1) and benign (f(**x**)=-1), e.g. SVM, [3] suggested to solve Eq. (9) through gradient ascent. To minimize the size of the perturbation and maximize the adversarial effect, the white-box perturbation should follow the gradient direction (i.e., the direction providing the largest increase of model value, from one label to another). Therefore, the perturbation r in each iteration is calculated as:

$$\boldsymbol{r} = \epsilon \nabla_{\boldsymbol{x}} Loss_f(\boldsymbol{x}+\boldsymbol{r}, -1)\ s.t.\ f(\boldsymbol{x}) = 1 \tag{10}$$

,where ϵ is a parameter controlling the magnitude of the perturbation introduced. By varying ϵ, this method can find an adversarial sample **x+r**.

[40] views the (white-box) adversarial problem as a constrained optimization problem, i.e., find a minimum perturbation in the restricted sample space. The perturbation is obtained by using Box-constrained L-BFGS to solve the following equation:

$$\arg_{\boldsymbol{r}} \min(c|\boldsymbol{r}| + Loss_f(\boldsymbol{x}+\boldsymbol{r}, l)\ s.t.\ \boldsymbol{x}+\boldsymbol{r} \in \boldsymbol{D} \tag{11}$$

, where c is a term added for the Lagrange multiplier.

[11] introduced the white-box Fast Gradient Sign Method (FGSM). The intuition behind the attack is to linearize the cost function $Loss()$ used to train a model f around the neighborhood of the training point **x** with a label y, that the adversary wants to force the misclassification of. Under this approximation:

$$\boldsymbol{r} = \epsilon sign(\nabla_{\boldsymbol{x}} Loss_f(\boldsymbol{x}, y)) \tag{12}$$

The white-box Jacobian-based Saliency Map Approach (JSMA) was introduced in [28]. The method iteratively perturbs features of the input that have large adversarial saliency scores. Intuitively, this score reflects the adversarial goal of taking a sample away from its source class towards a chosen target class.

First, the adversary computes the Jacobian of the model: $\left[\frac{\partial f_j}{\partial x_i}(\boldsymbol{x})\right]_{i,j}$, where component (i, j) is the derivative of class j with respect to input feature i. To compute the adversarial saliency map, the adversary then computes the following for each input feature i:

$$S(\boldsymbol{x}, t)[i] = \begin{cases} 0 & if\ \frac{\partial f_t(\boldsymbol{x})}{\partial x_i} < 0\ or\ \sum_{j \neq t} \frac{\partial f_j(\boldsymbol{x})}{\partial x_i} > 0 \\ \frac{\partial f_t(\boldsymbol{x})}{\partial x_i} | \sum_{j \neq t} \frac{\partial f_j(\boldsymbol{x})}{\partial x_i} | & otherwise \end{cases} \tag{13}$$

where t is the target class that the adversary wants the machine learning model to assign. The adversary then selects the input feature i with the largest saliency score $S(\boldsymbol{x}, t)[i]$ and increases its value. The process is repeated until misclassification in the target class is achieved or the maximum number of perturbed features has been reached. It creates smaller perturbations using higher computing cost, comparing to [11].

[29] performs a black-box adversarial example generation in 2 phases: 1) Substitute model training: the attacker queries the black-box model f with synthetic inputs selected by augmenting initial set of inputs representative of the input domain with their FGSM perturbed variants, to build a model $\hat{f}$ approximating f's decision boundaries. 2) Adversarial sample crafting: the attacker uses substitute network $\hat{f}$ to craft adversarial samples, which are then misclassified by f due to the transferability of adversarial examples. The main differences from this paper: 1) It deals with feed-forwards networks (especially CNNs) - and not with RNNs. 2) It doesn't fit the attack requirements in the cyber security domain, that is, not harming the malware functionality. This is because changing a pixel color in an image is legitimate and would rarely cause a human to see a distorted image. However, changing an API call could damage the code's functionality.

[13] presents a white-box evasion technique for Android static analysis malware classifier. The features used were: permissions, hardware components, suspicious API calls, activities\services\content providers\broadcast receivers, intents to communicate with other apps. Most of the features comes from the AndroidManifest.xml file. The attack is performed iteratively in 2 steps, until a benign classification is achieved: 1) Compute the gradient of h with respect to the binary feature vector $\boldsymbol{x}$. 2) Find the element in $\boldsymbol{x}$, whose modification from 0 to 1 (that is, only feature addition and not removal) would cause the maximum change in the benign score and add this manifest feature to the adversarial example. In contrast to our work, this paper didn't deal with dynamic features, which are more challenging to add without harming the functionality. It also didn't focus on a generic attack that can effect many classifier types, as we do.

[17] used API calls uni-grams as static features. That is, if n API types are being used, the feature vector dimension is n. A Generative Adversarial Networks (GAN) was trained where the discriminator simulates the malware classifier while the generator tries to generate adversarial samples that would be classified as benign by the discriminator, which uses labels from the black-box model. However, this attack doesn't fit RNN variants classifiers and is therefore less generic than our proposed method. In addition, we present a full end-to-end approach, using a mimicry attack. Finally, GAN are known for their instable training process ([1]), making such attack method hard to rely on.

Existing researches on adversarial samples mainly focus on images. Images are represented as matrices with fixed dimensions, and the values of the matrices are continuous. API sequences consist of discrete symbols with variable lengths.

Therefore, generating adversarial examples for API sequences will become quite different from generating adversarial examples for images.

4) RNN Adversarial Examples: [30] presented a white-box adversarial examples attack against RNNs, demonstrated against LSTM, for a sentiment classification for movie reviews dataset, where the input is the review and the output is whether the review was positive or negative. The adversary iterate over the words $\boldsymbol{x}[i]$ in the review and change each word to $\boldsymbol{z}$:

$$\boldsymbol{x}[i] = \arg\min_{\boldsymbol{z}} ||sign(\boldsymbol{x}[i] - \boldsymbol{z}) - sign(J_f(\boldsymbol{x})[i, f(\boldsymbol{x})])|| s.t. \boldsymbol{z} \in \boldsymbol{D} \tag{14}$$

, where f(**x**) is the original model label for **x**, $J_f(\boldsymbol{x})[i,j] = \frac{\partial f_j}{\partial x_i}(\boldsymbol{x})$. $sign(J_f(\boldsymbol{x})[i, f(\boldsymbol{x})]$ gives the direction in which one have to perturb each of the word embedding components in order to reduce the probability assigned to the current class, and thus change the class assigned to the sentence. However, the set of legitimate word embeddings is finite. Thus, one cannot set the word embedding coordinates to any real value. Instead, one finds the word $\boldsymbol{z}$ in dictionary $\boldsymbol{D}$ such that the sign of the difference between the embeddings of $\boldsymbol{z}$ and the original input word is closest to $sign(J_f(\boldsymbol{x})[i, f(\boldsymbol{x})]$. This embedding takes the direction closest to the one indicated by the Jacobian as most impactful on the model's prediction. By iteratively applying this heuristic to sequence words, one eventually find an adversarial input sequence misclassified by the model. The differences in this paper are: 1) We present a black-box attack, in which the adversary don't need prior knowledge on the malware classifier, which makes this attack more feasible. 2) We don't modify existing API calls: while this attack is relevant for reviews - it might damage a malware functionality.

Concurrently and independently from our work, [18] proposed a generative RNN based approach to generate irreverent APIs and insert them into the original API sequences. A substitute RNN is trained to fit the victim RNN. Gumbel-Softmax, a one-hot continuous distribution estimator, was used to smooth the API symbols and deliver gradient information between the generative RNN and the substitute RNN. null APIs were added, but while they were omitted to make the adversarial sequence generated shorter (after omitting them), they remained in the gradients of their loss function. This make the attack's success probability lower (as seen for the LSTM classifier), since the substitute model is used to classify the Gumbel-Softmax output, including the null APIs estimated gradients, so it doesn't exactly simulate the malware classifier. In contrast, our attack method don't have this difference between the substitute model and the black box model, since our generated API sequence are shorter. This makes our adversarial example faster and with higher success probability (Table II). Unlike [18], which focused only on LSTM variants, we also show our attack effectiveness against other RNN variants like GRU and conventional RNN, bidirectional and deep variants, non-RNN classifiers, including both feed-forward networks and traditional machine learning classifiers such as SVM, making it truly generic. Finally, the stability issues related with GAN training ([1]) applies to the attack method mentioned in [18], either, making it hard to rely on.

III. Methodology

A. Black-Box API Calls Based Malware Classifier

Our classifier's input is a sequence of API calls made by the inspected code. It uses only the API call type and not the arguments or return value. One might claim that considering this data, which is recorded by tools such-as Cuckoo Sandbox, would make our attack easier to detect by looking for irregularities in the arguments passed to it (e.g., invalid file handles, etc.) or consider only successful API calls and ignore failed APIs. In order to address this issue, we don't use NULL arguments that would fail the function. Instead, arguments that are valid but do nothing, such as writing into a temporary file instead of an invalid file handle, should be used.

In-order to fit all the classifiers we test, we use one-hot encoding for each API call type and feed the entire sequence to the classifier. The output of each classifier is binary: is the inspected code malicious or not. We tested several classifiers: Their types and hyper-parameters are detailed in section IV-B.

B. Black-Box API Calls Based Malware Classifier Attack

The proposed attack has two phases: 1) Creating a surrogate model. 2) Generating adversarial examples against the surrogate model and use it against the attacked black-box model using the transferability property.

1) Creating a surrogate model: We use a similar approach to [29]: Jacobian-based dataset augmentation. This method limits the number of black box model queries and ensures that the surrogate model is an approximation of the targeted model, that is, it learns similar decision boundaries (although not necessarily with similar accuracy).

We query the black box model with synthetic inputs selected by a Jacobian-based heuristic to build a surrogate model $\hat{f}$, approximating the black box model f's decision boundaries. While the adversary is unaware of the architecture of the black box model, we assume the basic features used (here: the recorded API calls type) are known to the attacker. In-order to learn similar decision boundaries as the black box model while using minimal amount of black box model queries, the synthetic training inputs is based on prioritizing directions in which the model's output is varying. This is done by evaluating the sign of the Jacobian matrix dimension corresponding to the label assigned to input **x** by the black box model: $sign(J_{\hat{f}}(\boldsymbol{x})[f(\boldsymbol{x})])$, as done by FGSM (Eq. 12). We use the Jacobian matrix of the surrogate model, since we don't have access to the Jacobian matrix of the black-box model. The new synthetic data point: $\boldsymbol{x} + \epsilon sign(J_{\hat{f}}(\boldsymbol{x})[f(\boldsymbol{x})])$ is added to the training set.

The surrogate model is being trained on exponentially-increasing dataset size: $|X_t| = 2^{t-1}|X_1|$

Algorithm 1 Surrogate Model Training

Input: f (black box model), T (training epochs), X_1(initial dataset), ϵ (perturbation factor)
Define architecture for the surrogate model A
for t=1..T:
Label the synthetic dataset using the black box model:
$D_t = \{(\boldsymbol{x}, f(\boldsymbol{x}))|x \in X_t\}$
(Re-)Train the surrogate model:
$\hat{f}_t = train(A, D_t)$
Perform Jacobian-based dataset augmentation:
$X_{t+1} = \left\{\boldsymbol{x} + \epsilon sign(J_{\hat{f}_t}(\boldsymbol{x})[f(\boldsymbol{x})])|x \in X_t\right\} \cup X_t$
return $\hat{f}_T$

X_1, the initial dataset, should be representative input. The reason is that we want the dataset augmentation to cover all decision boundaries. If we choose, e.g., only samples from a single family of ransomware as the initial dataset, we would focus only in a specific area of the decision boundary and our augmentation would probably takes us only to a certain direction. Choosing more diverse input would increase the augmentations effectiveness.

The samples we used were randomly selected samples from the test set (not the training set) distribution. Those samples were removed from the test set without using them, to prevent biased results by the surrogate model on those samples.

2) Generating adversarial examples: In the context of a malware classifier, an adversarial example is a sequence of API calls classified as malicious by the classifier, perturbed by adding API calls to it, so it would be mis-classified as benign. We cannot remove or modify API calls, only add additional API calls, to prevent damaging the code's functionality.

Algorithm 2 Adversarial Sequence Generation

Input: f (black box model), $\hat{f}$ (surrogate model), $\boldsymbol{x}[0:l-1]$ (malicious sequence to perturb, of length l), D (vocabulary)
$\boldsymbol{x}^* = \boldsymbol{x}$
# While the black-box model's classification of the perturbed example hasn't change (is still malicious):
while $f(\boldsymbol{x^*}) == f(\boldsymbol{x})$:
    Randomly select an API's position i in $\boldsymbol{x^*}$
    # Insert a new adversarial API in position i:
    $\boldsymbol{x^*}[i] = \arg\min_{api} ||sign(\boldsymbol{x}^* - \boldsymbol{x^*}[0:i-1] \perp api \perp \boldsymbol{x^*}[i:l-2]) - sign(J_{\hat{f}}(\boldsymbol{x})[f(\boldsymbol{x})])||$
return $\boldsymbol{x}^*$

D is the vocabulary of available features. In the context of API calls based malware classifier, those are all the API calls recorded by the classifier. $\boldsymbol{x}^*$ is the adversarial API calls sequence, based on $\boldsymbol{l}$. n is the length of API calls sequence, as an input to the classifier.$\perp$ is the concatenation operation: $\boldsymbol{x^*}[0:i-1] \perp api \perp \boldsymbol{x^*}[i:l-2]$ is the insertion of the encoded api vector in position i of $\boldsymbol{x}^*$. Notice that an insertion of an API in position i means that the APIs from position $i..l$ ($\boldsymbol{x^*}[i:l-1]$) are "pushed back" one position to make room for the new API call, to keep them and preserve the original functionality of the code. Since $\boldsymbol{x}^*$has a fixed length, the last API call, $\boldsymbol{x^*}[l-1]$ is being "pushed out" and removed from $\boldsymbol{x}^*$ (this is why the term is: $\perp \boldsymbol{x^*}[i:l-2]$ and not: $\perp \boldsymbol{x^*}[i:l-1]$). This might raise the concern that the proposed attack isn't really effective, but simple omit relevant information from the classifier. However, in section IV-C-2 we would see that even malware classifiers based on shorter sequence lengths have roughly the same accuracy. Thus the APIs being omitted are not the reason for the proposed attack effectiveness. The newly added APIs are.

$\boldsymbol{x^*}[i] = \arg\min_{api} ||sign(\boldsymbol{x}^* - \boldsymbol{x^*}[0:i-1] \perp api \perp \boldsymbol{x^*}[i:l-2]) - sign(J_{\hat{f}}(\boldsymbol{x})[f(\boldsymbol{x})])||$ is the perturbation. The rationale is similar to the adversarial sequence generation described in [30]: $sign(J_{\hat{f}}(\boldsymbol{x})[f(\boldsymbol{x})])$ gives us the direction in which we have to perturb the API calls sequence in order to reduce the probability assigned to the current class, $f(\boldsymbol{x})$, and thus change the class assigned to the sequence. However, the set of legitimate API call embeddings is finite. Thus, we cannot set the new API to any real value in an adversarial sequence $\boldsymbol{x}^*$. To overcome this difficulty, we use the perturbation mentioned above. We find the API call api in dictionary D such that the sign of the difference between the embeddings of the new API added to the original sequence, and the original API calls sequence, is closest to $sign(J_{\hat{f}}(\boldsymbol{x})[f(\boldsymbol{x})])$. This API call embedding takes the direction **closest** to the one indicated by the Jacobian as most impactful on the model's prediction. By iteratively applying this heuristic to sequence words, we eventually find an adversarial input sequence misclassified by the model. Notice that in [30], the authors *replaced* a word in a movie review with another word, so they only needed a single element out of the Jacobian: for element i, which was replaced. All other elements remained the same, so no gradient change took place. In contrast, since we add an API call, all the existing API calls following it shift their position, so we need to consider the difference in the gradients of all partial derivatives for the label $f(\boldsymbol{x})$ following the position i, and not just in, as in [30].

The method mentioned in [30] could work on an API calls based malware classifier (and provide a better performance for the adversarial sequence generation) only if a single API *equivalence attack* exists, that is, if one could find an alternative API that could replace an existing one, preserving its functionality. The problem is that in practice, even if such equivalent API exist, replacing it might still break the modified code's business logic. For instance, although *fopen()* and *CreateFile()* both open a file, *fopen()* returns a file descriptor, while *CreateFile()* returns a file handle. Therefore, subsequent calls to, e.g., *ReadFile()* would not work if no additional APIs are added to transform the file descriptor to a file handle. Those additional APIs make to original, more efficient partial Jacobian computation to be similar to our method. Therefore, we decided to add no-op API calls, giving us the ability to add any API call (with no-op arguments, such as *WriteFile()* of a temporary file) instead-of limiting ourselves only to API calls which have equivalents. The computational complexity remains the same.

While the proposed attack is designed for API calls based classifiers, it can be generalized to any adversarial sequence generation. This generalization has an interesting property: it's a high-performance attack, based on Definitions 3 and 4. This can be seen in section IV-C-3, where we compare the proposed attack to [30] on the IMDB sentiment classification dataset.

We make an assumption that the attacker knows D, that is, what API calls are available and how each of them is encoded (one-hot encoding in this paper). This is a reasonable assumption about the attacker's knowledge, commonly accepted (e.g. in [19]).

a) Implementation Using Mimicry Attack: In-order to add API calls in a way that doesn't hurt the code's functionality, we would generate a *mimicry attack*, following the methods mentioned in [42].

Equivalence attacks (i.e., using different API calls to implement the same desired functionality) and *disguise attacks* (using the same API type as the benign sequence with different parameters to implement malicious functionality) don't fit, since they lack the flexibility needed to modify every API call. Therefore the attacker can implemented a *no-op attack,* calling APIs which would have no effect on the code's functionality. This method has two variants: 1) Calling the desired API with invalid parameters, making it fail. For example: opening a (non-existent) file. This can be done to nearly all API calls with arguments. API calls without arguments can usually be called and ignored. However, since some API call monitors, such as Cuckoo Sandbox, also monitor the return value of an API call, and might ignore failed API calls, we need to implement a more robust way. 2) No-Op API calls with valid parameters, e.g., reading (0 bytes) from a file. This might be more challenging to add, since a different implementation should be taken for each API. However, this effort can be done once and then be used for every malware. Once done, it makes detecting that those APIs are invalid much harder, since the return value is success. Further measures can be taken by the attacker to prevent the detection of the no-op APIs by analyzing the arguments of the API calls, e.g., choosing randomly between several no-op arguments per API call, instead-of hard-coding them.

3) Generic Attack Using Transferability: So far we saw an attack against a white-box surrogate model. We now show that the same adversarial examples generated against the surrogate model would be effective against both the black box model and other classifier types as well. This can be explained using the principle of transferability. As mentioned in section II-D-1, since the decision boundaries of classifiers using datasets with a similar distribution are close to each other, gradient-based attacks that works against one could work against the other as well.

Transferability has two forms relevant to this paper: 1) The adversary can craft adversarial examples against a surrogate model, such as the one generated in section III-A. If the decision boundary of the surrogate model is close enough to that of the black box model (or if the perturbation is large enough to cover the difference), the same adversarial example would work against both ([29]). 2) An adversarial example crafted against one classifier type might work against a different classifier type, if their decision boundaries are close enough ([39]). This means that the attack should be generic enough to be effective (see Definition 4) against other classifiers type. In-order to define what makes adversarial attacks generic, we define the following property:

Definition 1. An adversarial attack is *transferable by transitivity* if: 1) The training set of the classifier being attacked is large enough to reflect the distribution of the samples in the wild. 2) An attack that doesn't use any property which is specific to the classifier it's targeted against, and thus can work against other classifiers without limitations. 3) An attack targeted against a generalized (more complex) model - and would therefore be transferable to simpler models, with similar decision boundaries, and simpler gradients. The bigger the perturbation of the adversarial examples, the bigger the chance that the adversarial sample would cross the decision boundaries difference between the models.

The rationale behind the third part of Definition 1 is the following: Assuming both complex and simpler models are train on the same dataset (or on samples from the same distribution), training on large dataset should result in similar decision boundaries for both: the decision boundaries that reflect the exact distribution of the samples, that is, the "ideal" classifier for that training set. The larger the training-sets, the closer they are to the same distribution, of actual benign vs malicious samples in the wild. Therefore, assuming the training sets are large enough, the main difference are the gradients of the classifier: most attacks, including Algorithm 2, perturb in the direction of the gradient, either approximated ([11]), fully evaluated (Algorithm 2), or in a specific direction ([30]). Thus, if the gradients of the models are similar, the attack is more likely to be transferable. Why "attack is transferable from complex to simpler models"? Since a more complex model is a general case of the simpler model, it is assumed to be a better approximation of the ideal classifier: more non-linear correlations between features or features' abstractions, etc. If the model better represent the ideal classifier, its gradients better represent the (feature-wise) direction in which the classification changes.

The proposed attack, an adversarial attack against RNN variants is *transferable by transitivity* to the other tested classifiers: 1) It doesn't use any specific property of RNN classifiers (and in fact, a similar attack was proposed against feed-forward networks in [29]). 2) RNN variants are a generic form of feed-forward networks (where the hidden state weight is always 0) and therefore their decision boundaries are a generalized case of feed-forward networks for the same dataset, either. Feed-forward DNNs, in turn, are generalized form of several traditional machine learning classifiers. For instance, the linear SVM classification score is: $y = \boldsymbol{w}\boldsymbol{x} + b$. It is based on the sum of features multiplied by the weights of the SVM model $\boldsymbol{w}$ and an added bias b. A linear SVM's equation is almost identical to Eq.(1), an artificial neuron

equation. However, adding more neurons in the same layer or hidden layers in a DNN make it a generalized form of linear SVM. Therefore, the attack presented here against RNN variants would be *transferable by transitivity* to feed-forward networks simplified case and from there *transferable by transitivity* to the linear SVM classifier simplified case. In this way, attacks against generalized models are transferable and effective against simpler models. The evaluation to back-up this claim is provided in section IV-D.

To our knowledge, this is the first time both forms of transferability mentioned above are being evaluated in the context of RNN variants, proving that the proposed attack is effective against the widest range of classifiers ever reviewed against a single attack: against RNN variants (like LSTM, GRU, etc.), against feed-forward DNNs (including CNNs) and against traditional machine learning classifiers such as SVM and random forest.

IV. Experimental Evaluation

A. Dataset

Our dataset contains 500,000 files, 250,000 benign samples and 250,000 malware samples, including the latest variants. For instance, we have, among the malware samples, ransomware families such as Cerber, Locky, CryptoWall, TeslaCrypt, Petya and many other families and malware types.

Since we're implementing a supervised learning classifier, we need reliable labels. We therefore labeled our dataset using VirusTotal[1], an on-line scanning service which contains more than 80 different security products. Our ground truth is: malicious sample is one with more than 15 positive (that is, malware) classification from the various products in VirusTotal. A benign sample is one with 0 positive classifications. All other samples (with 1-14 positives) were omitted to prevent FP contamination of the dataset.

In-order to extract the API calls of the inspected code, we run each sample in Cuckoo Sandbox[2], a commonly-used malware analysis system. We used a python script to parse the JSON file generated by it, and extracted the API call sequences generated by the inspected code. Those are the malware classifier's features. Although the JSON can be used as raw input for a NN classifier, as done in [35], we parsed it, since we wanted to focus on API calls only, without adding other features such as file paths or other features being extracted by Cuckoo Sandbox.

The overview of the malware classification process is shown in Figure 1.

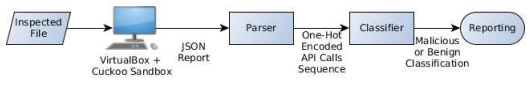

Figure 1. Overview of the Malware Classification Process

[1]https://www.virustotal.com/

[2]http://www.cuckoosandbox.org/

We run the samples on a VirtualBox's[3] snapshot with Windows 7 OS, since most malware target the Windows OS[4].

Since Cuckoo Sandbox is a tool known to malware writers, some of them write code to detect if the malware is running in a Cuckoo Sandbox (or any virtual machine, for that matter) and if so - quit immediately to prevent reversing efforts. In those cases, the file is malicious, but its behavior recorded in Cuckoo Sandbox, or API calls sequence in this case - isn't. To mitigate the contamination of our dataset, we used two counter-measures: 1) We applied Yara rules[5] to detect sandbox programs such as Cuckoo and omitted all samples that uses those techniques. 2) We considered only API call sequences with more than 15 API calls (the same as in [33]), to make sure there are enough API calls to make a valid classification. This filtering left us with 400,000 valid samples, after removing some benign samples to keep the dataset balanced. One might argue that those evasive malware, which apply those techniques, are extremely challenging and are of interest, but one should bare in mind we focus on the adversarial attack and it is generic enough to work for those evasive malware either, if other mitigation techniques, such as anti-anti-VM, would be applied to monitor them.

Another issue is the maximal sequence length. While RNN variants can handle variable sequence lengths, other classifiers, for which we want to apply the attack by transferability, don't. Therefore, adversarial examples that fit variable sequence lengths might be too long for such classifiers, e.g., fully-connected DNN. Moreover, using variable sequence lengths, or even fixed but long sequence length increase the training time substantially, with only marginal effect on the accuracy, as shown in [33], Figure 3. Finally, since malware code tends to run for shorter time than benign code (e.g. dropper software that downloads a malicious payload and quits vs. a benign UI application, remaining running until the user decides to quit), not limiting the API calls sequence length would cause the classifier to learn: "long sequence is benign", which is not the API based decision we are looking for. Therefore, we limited our maximum sequence length to $n = 140$ API calls and padded the sequence with zeros, which stands for a null API in our one-hot encoding, if it was shorter, similarly to [33]. While it is true that such a malware classifier can be evaded, e.g., by running 140 benign API calls before staring the malicious functionality, this knowledge is available for a white-box attacker, not a black-box one. This problem can also be mitigated by randomly selecting the number of API calls for the training set each time the model is being trained, or creating an ensemble various API call sequence length models. However, none of those techniques damage the generality of the black-box attack we described in section III-B, which fit any API calls sequence length.

The final filtered training set size was 360,000 samples, 36,000 samples of which were the validation set. The test

[3]https://www.virtualbox.org/

[4]https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf

[5]https://github.com/Yara-Rules/rules

set size was 36,000 samples. All sets are balanced between malicious and benign samples.

B. Malware Classifier Performance

Since no open-source API calls based deep learning intrusion detection system is available, we created our own black-box malware classifiers. We tested several classifiers. As mentioned in section IV-A, all of them get as input the vector of n=140 first API calls of the analyzed code in one-hot encoding and produce a binary classification: malicious or benign.

We used the Keras[6] implementation for all NN classifiers, with Tensorflow back-end. Scikit-Learn[7] was used for all other classifiers.

The loss function we used for training was binary cross-entropy. We used the Adam optimizer for all NNs. The output layer was fully connected with sigmoid activation for all NNs. The hyper-parameters were chosen based on the state-of-the-art papers, e.g., number of hidden layers from [20], section 5.4 and [13], section 4.1), drop-out rate from [20], section 5.3 and number of trees in a random forest classifier and the decision trees splitting criteria from [34], and were fine-tuned by us. For NNs, A rectified linear unit, $ReLU(x) = max(0, x)$, was chosen as an activation function due-to its fast convergence ([10]) comparing to $sigmoid()$ or $\tanh()$, and dropout ([38]) was used to improve the generalization potential of the network. Training was conducted for maximum 100 epochs, but convergence usually reached after 15-20 epochs, depending on the classifier's type. For training, we used a batch size of 32 samples.

The classifiers also have the following classifier-specific hyper-parameters:

- DNN - 2 fully connected hidden layers of 128 neurons, each with ReLU activation and dropout rate of 0.2.
- CNN - 1D ConvNet with 128 output filters, stride length of 1, 1D convolution window size of 3 and a ReLU activation, followed by a global max pooling 1D layer, followed by a fully connected layer of 128 neurons with ReLU activation and dropout rate of 0.2.
- RNN, LSTM, GRU, BRNN, BLSTM, Bidirectional GRU - a hidden layer of 128 units, with dropout rate of 0.2 for both inputs and recurrent states.
- Deep LSTM and BLSTM - 2 hidden layer of 128 units, with dropout rate of 0.2 for both inputs and recurrent states in both layers.
- Linear SVM and logistic regression classifiers with a regularization parameter C=1.0 and L2 norm penalty.
- Decision Tree with unlimited maximum depth and the Gini criteria for choosing the best split.
- Random forest classifier, using 10 decision trees with unlimited maximum depth and the Gini criteria for choosing the best split.

We measure the performance of the classifiers using the accuracy ratio, which gives an equal weight to both FP and FN (unlike precision or recall), therefore giving an unbiased overall performance indicator:

Definition 2.

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{15}$$

, where: TP - True Positives (malicious samples being classified as malicious by the black-box classifier), TN - True Negatives, FP - False Positives (benign samples being classified as malicious), FN - False Negatives.

The classifier types' performance is shown in Table 1. The accuracy is measured on the test set, containing 36,000 samples.

Table I
CLASSIFIER PERFORMANCE

Classifier Type	Accuracy (%)
RNN	82.29
BRNN	82.60
LSTM	97.61
Deep LSTM	97.48
BLSTM	97.77
Deep BLSTM	96.91
GRU	96.42
Bidirectional GRU	96.02
Fully-Connected DNN	97.59
1D CNN	97.63
Random Forest	97.36
Decision Tree	96.93
SVM	95.33
Logistic Regression	95.40

We can see from Table 1 that RNN classifier has worse performance than LSTM and GRU and LSTM is better than GRU. This can be explained by the fact that both GRU and LSTM classifiers don't suffer from the exploding and vanishing gradients problem as much as vanilla RNN, and can thus leverage long term context, as explained in section II-B-1. LSTM is better than GRU due-to the more flexible (and complex) architecture (input, output and forget gates instead-of update and reset gates in GRU allowing finer-grained control of what parts of the state to propagate), which gives it a better accuracy, at the expanse of a longer training time, as explained in section II-B-2. Those results are aligned with the comparison in [2]. We see that the bidirectional variants are better than the unidirectional ones (with the exception of GRU), due-to their ability to "remember" the beginning of long sequences, as explained in section II-B-3. BLSTM is the best malware classifier, accuracy-wise, and, as would be shown in Table 2, is also one of the most resistant to the proposed attack, meaning the proposed attack had poor performance against BLSTM, by Definitions 3 and 4. The deep LSTM and BLSTM variants under-perform in comparison to their shallow variants. This means that higher abstraction of this dataset by representation learning doesn't help gaining any additional correlation between the feature abstractions while issues related with training of deep models, such as vanishing gradients and more required training epochs to converge,

[6]https://keras.io/

[7]http://scikit-learn.org/stable/

actually slightly reduce the classifier's performance. However, they are the most resistant to our attack, as shown in Table 2.

We also see that the feed-forward networks, both a fully-connected DNN and 1D CNN have a very good performance. The reason is the small sequence length that allows the entire sequence to observed at once, finding all the correlations between the sequence elements. However, feed-forward networks are expected to be less suited to real-world classifiers, with longer sequence lengths or on-line classification based on partial API call traces. However, we added them to show that the proposed attack is generic and that transferability works between RNN variants and feed-forward networks, either.

An interesting difference from [2] is the performance of CNN. In contrast to [2], our 1D CNN was the second best, accuracy-wise. It was also the second most resistant to the proposed attack, as would be shown in section IV-C. The reason is that the in [2], a 9 layer 2D CNN was used, regarding each API call as a single character in a sentence. While 2D CNN might make sense for NLP, where most sentences are of equal lengths and correlations between two sentences exist, this method makes less sense for API call sequences, there is no reason to assume that each n-th API would have (spatial) correlation, where n (the 2nd dimension) is chosen arbitrarily. In contrast, we used a 1D CNN, treating each API as a word in a paragraph, which doesn't create artificial spatial correlation between non-adjacent API calls, while preserving the actual spatial correlation between adjacent API calls, making our architecture more suited to the task and perform better.

Finally, we see that traditional machine learning classifiers perform worse than RNN variants and feed-forward networks. This is due-to the fact that they have less levels of non-linear correlations between the features, making them less accurate. A notable exception is random forest, which compensate this deficiency with its ensemble model, with different decision trees for different API call flows, e.g., a decision tree for ransomware vs full disk encryption benign applications, etc.

C. Attack Performance

In-order to measure the performance of an attack, we consider two factors:

The attack effectiveness is the number of malware samples, out of the training set which were successfully classified by the black-box malware classifier as malicious, but their adversarial sequences generated by Algorithm 2 were mis-classified by it.

Definition 3.

$$attack\ effectiveness = \frac{|\{f(x) = Malicious \vee f(x^*) = Benign\}|}{|\{f(x) = Malicious\}|} \quad (16)$$

$$s.t.\ x \in TestSet(f), \hat{f_T} = Algorithm1(f, T, X_1, \epsilon),$$

$$x^* = Algorithm2(f, \hat{f_T}, x, D)$$

We also consider the overhead that is caused by the proposed attack. The attack overhead is the average additional API calls added by Algorithm 2 to a malware sample in-order make it classified as benign (therefore calculated only for successful attacks) by the black-box model, as a rate of the total API calls sequence length:

Definition 4.

$$attack\ overhead = \frac{average\ added\ APIs}{n} \quad (17)$$

A high-performance attack would be combining the maximal attack effectiveness with the minimal attack overhead.

Based on Definitions 3 and 4, the proposed attack performance is specified in Table 2.

Table II
ATTACK PERFORMANCE

Classifier Type	Attack Effectiveness (%)	Additional API Calls (%)
RNN	100.00	8.36
BRNN	99.99	9.64
LSTM	100.00	3.19
Deep LSTM	89.89	20.02
BLSTM	91.09	20.04
Deep BLSTM	91.75	29.63
GRU	99.71	5.64
Bidirectional GRU	99.23	18.88
Fully-Connected DNN	99.90	6.19
1D CNN	94.44	12.99
Random Forest	99.98	7.43
Decision Tree	100.00	3.27
SVM	99.86	4.76
Logistic Regression	98.76	6.66

We can see from Table 2 that the proposed attack has a very high effectiveness against all tested malware classifiers, usually over 99%, while the overhead is usually 5-10% additional APIs. BLSTM and the deep variants of LSTM and BLSTM are the most resistant to the proposed attack, with 1D CNN close behind, but even in those cases the attack effectiveness is more than 90% while the attack overhead is less than 30%.

We also see a strong negative correlation between the attack effectiveness and the additional API calls. This is a direct result of Algorithm 2 implementation: the harder it is for the algorithm to generate a successful mis-classification, the more APIs would be added. An exception to this rule is Bidirectional GRU, which has a relatively high attack effectiveness but requires a high attack overhead.

As mentioned in section IV-A, $|TestSet(f)| = 36,000$ samples and the test set $TestSet(f)$ is balanced, so the attack performance was measured on about: $|\{f(x) = Malicious | x \in TestSet(f)\}| = 18,000$ samples.

For the surrogate model used in this attack we used a perturbation factor of $\epsilon = 0.2$ and a learning rate of 0.1. $|X_1| = 70$ (randomly selected out of a test set of 36,000 samples) was used. We used T=6 surrogate epochs. Thus, as shown in section III-B-1, the training set size for the surrogate model is: $|X_6| = 2^5 * 70 = 2240$ samples, only 70 of them were picked from the test set distribution and all others were synthetically generated. Using lower values, e.g., $|X_1| = 50$ or T=5 achieved a much worse attack performance, meaning

those parameters are needed to closely approximate the black-box model's decision boundaries. For simplicity and training time, we always used $l = n$ for Algorithm 2, that is, the adversarial example has exactly the number of API calls used by the black-box classifier. However, even if his is not the case the attack effectiveness doesn't degrade by much.

The adversary is choosing the architecture for the surrogate model without any knowledge of the attacked model architecture. However, some choices lead to better performance. We have chosen GRU surrogate model with 64 units, while an LSTM model with the same number performed about the same but took longer to train. Other surrogate models performed worse. Besides the classifier's type and architecture, we also used a different optimizer for the surrogate model: Adadelta optimizer, instead-of Adam for the black-box model (section IV-A).

In our implementation, we used the cleverhans library[8].

1) Comparison to Random Perturbation: In order to make sure that our attack is indeed effective, we need a baseline comparison. We used *random perturbation* for that: we keep adding random (instead of Jacobean difference based on a surrogate model, as the attack method described in Algorithm 2) API calls to the malicious sample sequence until a benign classification by the black box model is achieved. We have used the deep BLSTM malware classifier with the same hyper-parameters mentioned in section IV-B, since it's the most resistant to the proposed attack, so the difference from the random attack should be the smallest. However, we see that even in this scenario, the attack effectiveness (by Definition 3) of the random perturbation method is 70.14% while the attack overhead (by Definition 4) is 47.86%: lower effectiveness and 1.5 times higher overhead, even in the successful adversarial examples. For shallow BLSTM the attack effectiveness of the random perturbation method is 87.41% while the attack overhead is 30.12%. Again, lower effectiveness and 1.5 times higher overhead. For other cases, not shown due to space limit, the results are similar, showing that the proposed attack performs better than random perturbation. In section IV-C-3 we show another comparison to previous work attack.

2) Sequence Length's Effect on Classification: One might claim that the reason the proposed attack is successful is that when adding a no-op API call, the last "actual" API call is deleted to keep the lengths of $\boldsymbol{x}$ and $\boldsymbol{x}^*$ equal (Algorithm 2). Therefore, we're actually deleting some of the data used by the malware classifier for the classification and this is the reason for the attack effectiveness and not the added API calls. In-order to verify that this deletion of relevant data is not the reason that the attack is effective, we tried smaller sequence lengths. We show the results in Table 3 only for an LSTM malware classifier due to space limit, but those numbers are similar to other classifiers, either:

Table III
LSTM CLASSIFIER SEQUENCE LENGTH'S EFFECT ON ACCURACY

Sequence Length	Accuracy (%)
100	96.76
120	97.46
140	97.61

We see that while longer sequence lengths do cause an increase in the classifier's accuracy, this effect is negligible (less than 1% accuracy difference).

Since, as mentioned in section IV-A, we used a maximal sequence length of n=140 API calls, an attack overhead of 20.04% (for BLSTM, in Table 2) is actually an addition of 28.06 API calls on average to the first 140 API calls in the inspected sequence, by Definition 4.

In the worse case from the attack overhead perspective, BLSTM, in which there are additional 28.06 API calls on average, 28 API calls are being deleted from the end of the sequence, so the first 110 API calls from the original sequence still exists in the adversarial sequence. Therefore, according to Table 3, one would expect the accuracy to reduce by less than 1%. However, BLSTM has an attack effectiveness of more than 90%, meaning 9 out of 10 attacks are successful, not 1 out of 100, due-to the accuracy reduction caused by the usage of shorter original sequences. This means that the deleted APIs have a negligible effect and the added APIs are the cause to the attack effectiveness.

3) Comparison to Previous Work: Besides [18], which was written concurrently and independently from our work, [30] is the only currently published RNN adversarial attack, to the best of our knowledge. [30] is somewhat similar to our proposed attack, specified in section III-B. The differences from this paper, as mentioned in section II-D-4, are: 1) Our attack is black-box, not white-box. 2) We don't modify existing API calls to prevent damage to the malware functionality. However, we want to compare our attack to previous work in terms of performance, that-is, in terms of effectiveness (Definition 3) and overhead (Definition 4).

In order to do that, we implemented both our proposed attack and the attack presented in [30], using the IMDB sentiment classification dataset. We used 5,000 samples for training, validation and 5,000 samples for test. We used the first 80 words in each review, padding with zeros if the review was shorter. All sequences were one-hot encoded. We used the 20,000 most frequent words in the reviews. For our attack, we used the same surrogate model type and hyper-parameters as in sections IV-B and IV-C.

The performance comparison is shown in Table 4. Notice that for Papernot et al. '16, the overhead is the number of *modified* API calls - not *added* API calls.

Table IV
PERFORMANCE COMPARISON TO PAPERNOT ET AL. '16

Attack Type	Attack Effectiveness (%)	API calls modified\added (%)
Papernot et al. '16	100	51.25
Algorithm 2	100	11.25

[8]https://github.com/tensorflow/cleverhans

We can see that although our attack is a black-box attack while the other attack is a white-box attack, both have the same effectiveness. While both attacks have high effectiveness, our attack has a much lower overhead. A possible reason is that adding words instead of modifying existing words cause less damage to the structure of the original sentence. For instance, an example shown in [30]: "I wouldn't rent this one even on dollar rental night." was modified into "Excellent wouldn't rent this one even on dollar rental night." by their attack, destroying the original sentence structure. However, using our attack would generate the sequence: "I wouldn't resist rent this one even on dollar rental.", which has a more coherent structure.

D. Transferability for RNN Models

While transferability was covered in the past in the context of DNNs (e.g. [40]), to our knowledge, this is the first time it is being evaluated in the context of RNNs, proving that the proposed attack is generic not just against specific RNN variant, but also between RNN variants (like LSTM, GRU, etc.), between feed-forward DNNs (including CNNs) and traditional machine learning classifiers such as SVM and random forest, due to its *transferability by transitivity*.

As mentioned in section III-B-3, transferability has two different levels relevant to this paper: 1) The adversary can craft adversarial examples against a surrogate model with a different architecture from the black-box model and the same adversarial example would work against both. 2) An adversarial example crafted against one black-box classifier type might work against a different black-box classifier type.

Both of the forms can be evaluated in Table 2: 1) As mentioned in section IV-C, we used a GRU surrogate model. However, as can be seen from the first eight lines of Table 2, the attack effectiveness is high even when the black-box classifier is not GRU. Even when the black-box classifier is indeed GRU, the hyper-parameters, such as the number of units and the optimizer are different. 2) The attack was designed against RNN variants. However, we tested it and found it effective against both feed-forward networks and traditional machine learning classifiers, as can be seen from the last six lines of Table 2.

Finally, we want to show another case of *transferability by transitivity*, that is, as hypothesized in section III-B-3, the property of an attack designed against generalized models is generic enough to deal with simpler forms of classifiers, while the opposite is not always true. To do that, we used a more complex and generalized surrogate classifier model, deep GRU, instead of shallow GRU, since it has more hidden layers. When generating an adversarial sequence against a deep GRU (complex) surrogate model and then performing it against a (simple) BLSTM black-box classifier, the attack effectiveness is 94.58% and the overhead is 24.26%. However, when doing the opposite, attacking a deep BLSTM black box model using an adversarial sequence generated against a shallow GRU surrogate model, we can see from Table 2 that the performance is much worse: attack effectiveness is 91.75% and the overhead is 29.63%. Other types exhibit the same trend, not shown due to space limits. This means that a sequence generated against the more generalized surrogate case has a higher effectiveness against a simple black-box model than the other way around.

V. Conclusions and Future Work

In this paper, we have shown a generic black-box API calls adversarial sequence generation algorithm, which was designed against RNN malware classifier. Unlike previous adversarial attacks, we have shown that this attack is generic due-to its *transferability by transitivity* property, and is the first attack with a verified effectiveness against all relevant common classifiers: RNN variants, feed-forward networks and traditional machine learning classifiers. Therefore, it can be called a black-box attack in the wide sense, since it requires the attacker to know nothing about the classifier besides the monitored APIs. We have also shown that this attack out-performs both base-line random attack and previously published attacks, both in its overhead and effectiveness.

The most resistant classifiers to this attack are BLSTM, deep BLSTM and deep LSTM, followed by 1D CNN. However, 1D CNN fits less to real world scenarios, such-as on-line classifiers which classify partial API calls feed, in-order to classify a malware before any damage infection is done to the machine, unlike RNNs, which handles those cases better. In addition, the order of the API calls has significance (e.g., *ReadFile()* before *WriteFile()* has different meaning than a reversed order) and is difficult to account for by using CNN.

Our future work would be in two areas: Defense mechanisms against such attacks and attack modifications to cope with such mechanisms. Due to space limits, we would publish an in-depth analysis of various defense mechanisms to future work. However, in this section we shortly describe previously explored work in the area and its relevance for our attack.

The defense mechanisms are divided into two sub-groups: 1) Detection of adversarial examples. 2) Making the classifier resistant to adversarial attack.

A. Detection of Adversarial Examples

Several methods have been suggested to detect whether a sample is an adversarial example.

[14] leverages the fact that adversarial samples have a different distribution from normal samples. The statistical differences between them can be detected using a high-dimensional statistical test of maximum mean discrepancy. [8] detects adversarial examples using two new features: kernel density estimates in the subspace of the last hidden layer of the original classifier, and Bayesian neural network uncertainty estimates. [25] augments deep neural networks with a small "detector" subnetwork which is trained on the binary classification task of distinguishing genuine data from data containing adversarial perturbations.

To the best of our knowledge, there is currently no published and evaluated method to detect RNN adversarial sequences. This would be part of our future work.

Moreover, the main issue with all of those methods is that in the cyber security domain, the question of what to do after you discovered an adversarial example remains open. Would such a sequence be blocked? Be classified as malicious? This might increase the FP rate, due to misclassifying samples as adversarial examples and than treat them as malicious. Reporting such cases is also not an option in the cyber security domain since this allows malicious files using this attack evade being blocked.

B. Making the Classifier Resistant to Adversarial Attacks

Another option, instead-of actively trying to detect adversarial examples is passively trying to make the classifier more robust against such attacks.

Distillation ([31]) is a mechanism designed to compress large models into smaller ones while preserving prediction accuracy. In order to do that, the large model labels data with class probabilities, which are then used to train the small model. Instead of compression, the authors apply distillation to increase the robustness of DNNs to adversarial samples. They report that the additional entropy in probability vectors (compared to labels) yields models with smoother output surfaces. However, [29] showed that defensive distillation can be evaded using a black-box attack. We would not show that our attack is also immune to distillation, due to space limits. This would be detailed in our future work. However, the reason for our attack immunity is explained in [27]: When applying defense methods that smooth a model's output surface, the adversary cannot craft adversarial examples because the gradients it needs to compute (e.g., the derivative of the model output with respect to its input) have values close to zero. However, the adversary may instead use a surrogate model to craft adversarial examples, as done in this paper. The surrogate model is not impacted by the defensive mechanism and will still have the gradients necessary to find adversarial directions. Due to the adversarial example transferability property, the adversarial examples crafted using the surrogate would also be misclassified by the black-box model.

Adversarial training was suggested in [40]: injecting adversarial samples, correctly labeled, in the training set as a means to make the model robust. In [11] this reduced the misclassification rate of a MNIST model from 89.4% to 17.9% on adversarial examples. We would not show that our attack is more resistant to adversarial training than other attacks discussed in those papers due to space limit. This would be shown in future work. However, the intuition for that is: API calls have a limited range of discrete values, in-contrast to, e.g., images, which have a wider range of possible values. Thus, when we follow the gradients in Algorithm 2, we actually transforming our samples to be closer to real benign samples distribution. This is not what happens with images, where adversarial examples have a third distribution, which is similar to neither the malicious distribution, nor the benign one, and can thus be labeled as malicious and easily classified correctly.

Finally, since our attack modifies only a specific feature type (API calls), combining several types of features, either dynamic ([43]) or both static and dynamic features ([20]) might make the system more resistant to adversarial examples against a specific feature type. We would show on our future work that combining attacks against feature-specific classifiers, e.g., as done in [24], is an effective way to bypass classifiers based on multiple types of features, due-to transferability.

To the best of our knowledge, there is currently no published and evaluated method to make an RNN model resistant to adversarial sequences. This would also be a part of our future work.

REFERENCES

[1] M. Arjovsky and L. Bottou, Towards principled methods for training generative adversarial networks, arXiv preprint arXiv:1701.04862, 2017.

[2] V. Athiwaratkun and J. W. Stokes, Malware Classification with LSTM and GRU language models and a character-level CNN, in in Acoustics, Speech and Signal Processing (ICASSP) 17.

[3] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. ęrndić, P. Laskov, G. Giacinto, and F. Roli, Evasion attacks against machine learning at test time, in Machine Learning and Knowledge Discovery in Databases, pp. 387–402. Springer, 2013.

[4] K. Cho, B. V. Merrienboer, D. Bahdanau, and Y. Bengio, On the properties of neural machine translation: Encoder-decoder approaches, in Proceedings of the Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST), 2014.

[5] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence qodeling, arXiv preprint arXiv:1412.3555, 2014.

[6] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, Large-scale malware classification using random projections and neural networks, in Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference, pp. 3422–3426

[7] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, A survey on automated dynamic malware-analysis techniques and tools, in: ACM Computing Surveys, Vol. 44, No. 2, Article 6, pp. 1-42 (2012)

[8] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, Detecting adversarial samples from artifacts, arXiv preprint arXiv:1703.00410, 2017.

[9] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longsta, A sense of self for Unix processes, in: IEEE Symposium on Security and Privacy, pp. 120-128, IEEE Press, USA, 1996

[10] X. Glorot, A. Bordes, and Y. Bengio, Deep sparse rectifier neural networks, in Proc. 14th International Conference on Artificial Intelligence and Statistics 315–323, 2011.

[11] I. J. Goodfellow, J. Shlens, and C. Szegedy, Explaining and harnessing adversarial examples, in International Conference on Learning Representations. Computational and Biological Learning Society, 2015.

[12] A. Graves and J. Schmidhuber, Framewise phoneme classification with bidirectional LSTM and other neural network architectures, in Neural Networks, 18(5-6):602– 610, June/July 2005.

[13] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, Adversarial perturbations against deep neural networks for malware classification, arXiv preprint arXiv:1606.04435, 2016.

[14] K. Grosse, P. Manoharan, N. Papernot, M. Backes, P. McDaniel, On the (Statistical) Detection of Adversarial Examples, arXiv preprint arXiv:1702.06280, 2017.

[15] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, DL4MD: A deep learning framework for intelligent malware detection, in International Conference on Data Mining (DMIN), 2016.

[16] S. Hochreiter and J. Schmidhuber, Long short-term memory, in Proceedings of Neural Computation, 1997, pp. 1735–1780.

[17] W. Hu and Y. Tan, Generating adversarial malware examples for black-box attacks based on GAN, arXiv preprint arXiv:1702.05983, 2017.

[18] W. Hu and Y. Tan, Black-box attacks against RNN based malware detection algorithms, arXiv preprint arXiv:1705.08131, 2017.

[19] L. Huang, A. Joseph, B. Nelson, B. Rubinstein B., and J. Tygar, Adversarial machine learning, in: 4th ACM Workshop on Artificial Intelligence and Security, pp. 43–57 (2011)

[20] W. Huang and J. W. Stokes, Mtnet: A multi-task neural network for dynamic malware classfication, in Proceedings of Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2016, pp. 399–418.

[21] H. Jiang, J. Nagra, and P. Ahammad, Applying machine learning in security - a survey, arXiv preprint arXiv:1611.03186, 2016.

[22] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, Deep learning for classification of malware system call sequences, in Australasian Joint Conference on Artificial Intelligence, pp. 137–149. Springer, 2016.

[23] Y. Lecun, Y. Bengio, and G. Hinton (2015). Deep learning. Nature, 521(7553), 436-444. DOI: 10.1038/nature14539

[24] Y. Liu, X. Chen, C. Liu, and D. Song, Delving into transferable adversarial examples and black-box attacks, in International Conference on Learning Representations, 2017.

[25] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, On detecting adversarial perturbations, in International Conference on Learning Representations, 2017.

[26] A. Moser, C. Kruegel, and E. Kirda, Limits of static analysis for malware detection, in: 23rd Annual Computer Security Applications Conference, pp. 421-430 (2007)

[27] N. Papernot, P. McDaniel, A. Sinha, and M. Wellman, Towards the science of security and privacy in machine learning, arXiv preprint arXiv:1611.03814, 2016

[28] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, The limitations of deep learning in adversarial settings, in Proceedings of the 1st IEEE European Symposium on Security and Privacy, IEEE, 2016.

[29] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, Practical black-box attacks against deep learning systems using adversarial examples, arXiv preprint arXiv:1602.02697, 2016

[30] N. Papernot, P. McDaniel, A. Swami, and R. Harang, Crafting adversarial input sequences for recurrent neural networks, in Military Communications Conference (MILCOM), pp. 49–54. IEEE, 2016.

[31] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks, in Symposium on Security & Privacy, pp. 582–597, San Jose, CA, 2016.

[32] N. Papernot, P. McDaniel, and I. Goodfellow, Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples, arXiv preprint arXiv:1605.07277, 2016

[33] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, Malware classification with recurrent networks, in ICASSP'15, pp. 1916–1920.

[34] I. Rosenberg, and E. Gudes, Bypassing system calls-based intrusion detection systems, Concurrency Computat: Pract Exper, doi:10.1002/cpe.4023, 2016.

[35] I. Rosenberg, G. Sicard, and E. O. David, DeepAPT: nation-state APT attribution using end-to-end deep neural networks, in International Conference on Artificial Neural Networks (ICANN) 2017

[36] J. Saxe and K. Berlin, Deep neural network based malware detection using two dimensional binary program features, arXiv preprint arXiv:1508.03096v2, 2015.

[37] M. Schuster and K. K. Paliwal, Bidirectional recurrent neural networks, in IEEE Transactions on Signal Processing, 45:2673–2681, 1997.

[38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting. J. Machine Learning Res. 15, 1929–1958, 2014.

[39] N. ęrndić and P. Laskov, Practical evasion of a learning- based classifier - A case study, in: Proceedings of the 2014 IEEE Symposium on Security and Privacy, pp. 197–211 (2014)

[40] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, Intriguing properties of neural networks, in Proceedings of the 2014 International Conference on Learning Representations. Computational and Biological Learning Society, 2014.

[41] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, Stealing machine learning models via prediction apis, in USENIX Security, 2016.

[42] D. Wagner and P. Soto, Mimicry attacks on host-based intrusion detection systems, in Proceedings of the 9th ACM conference on Computer and Communications Security, pp. 255-264 (2002)

[43] X. Wang and S. M. Yiu, A multi-task learning model for malware classification with useful file access pattern from API call sequence, arXiv preprint arXiv:1610.05945, 2016.

[44] W. Xu, Y. Qi, and D. Evans, Automatically evading classifiers, in Proceedings of the Network and Distributed Systems Symposium, 2016.