Generic Black-Box End-to-End Attack Against State of the Art API Call Based Malware Classifiers

Ishai Rosenberg(✉), Asaf Shabtai, Lior Rokach, and Yuval Elovici

Software and Information Systems Engineering Department, Ben Gurion University, Beersheba, Israel
ishairos@post.bgu.ac.il

Abstract. In this paper, we present a black-box attack against API call based machine learning malware classifiers, focusing on generating adversarial sequences combining API calls and static features (e.g., printable strings) that will be misclassified by the classifier without affecting the malware functionality. We show that this attack is effective against many classifiers due to the transferability principle between RNN variants, feed forward DNNs, and traditional machine learning classifiers such as SVM. We also implement GADGET, a software framework to convert any malware binary to a binary undetected by malware classifiers, using the proposed attack, without access to the malware source code.

Keywords: Adversarial attacks · Malware classification
Deep neural networks · Dynamic analysis · Transferability

1 Introduction

Machine learning malware classifiers, in which the model is trained on features extracted from the analyzed file, have two main advantages over current signature based/black list classifiers: (1) Automatically training the classifier on new malware samples saves time and expense, compared to manually analyzing new malware variants. (2) Generalization to currently unseen and unsigned threats is better when the classifier is based on features and not on a fingerprint of a specific and exact file (e.g., a file's hash).

Next generation anti-malware products, such as Cylance, CrowdStrike, and Sophos, use machine and deep learning models instead of signatures and heuristics. Those models can be evaded and in this paper, we demonstrate an evasive *end-to-end attack*, generating a malware binary that can be executed while not being detected by such machine learning malware classifiers.

Application programming interface (API) calls, often used to characterize the behavior of a program, are a common input choice for a classifier and used by products such as SentinelOne. Since only the sequence of API calls gives each

M. Bailey et al. (Eds.): RAID 2018, LNCS 11050, pp. 490–510, 2018.
https://doi.org/10.1007/978-3-030-00470-5_23

API call its context and proper meaning, API call sequence based classifiers provide state of the art detection performance [9].

Machine learning classifiers and algorithms are vulnerable to different kinds of attacks aimed at undermining the classifier's integrity, availability, etc. One such attack is based on the generation of adversarial examples which are originally correctly classified inputs that are perturbed (modified) so they (incorrectly) get assigned a different label. In this paper, we demonstrate an attack like this on binary classifiers that are used to differentiate between malicious and benign API call sequences. In our case, the adversarial example is a malicious API call sequence, originally correctly classified, which is classified by the classifier as benign (a form of evasion attack) after the perturbation (which does not affect the malware functionality).

Generating adversarial examples for API sequences differs from generating adversarial examples for images [2], which is the main focus of the existing research, in two respects: (1) API sequences consist of discrete symbols with variable lengths, while images are represented as matrices with fixed dimensions, and the values of the matrices are continuous. (2) In adversarial API sequences one must verify that the original functionality of the malware remains intact. Attacks against RNN variants exist [7,12], but they are not practical attacks, in that they don't verify the functionality of the modified samples or handle API call arguments and non-sequence features, etc. The differences from our attack are specified in Sect. 2.

The contributions of our paper are as follows:

1. We implement a novel *end-to-end black-box method* to generate adversarial examples for many state of the art machine learning malware classifiers. This is the first attack to be evaluated against RNN variants (like LSTM), feed forward DNNs, and traditional machine learning classifiers (such as SVM). We test our implementation on a large dataset of 500,000 malware and benign samples.
2. Unlike previous papers that focus on images, we focus on the cyber security domain. We implement GADGET, an evasion framework generating a new malware binary with the perturbed features *without access to the malware source code* that allows us to *verify that the malicious functionality remains intact.*
3. Unlike previous papers, we extend our attack to *bypass multi-feature (e.g., static and dynamic features) based malware classifiers*, to fit real world scenarios.
4. We focus on *the principle* of *transferability* in RNN variants. To the best of our knowledge, this is *the first time it has been evaluated in the context of RNNs and in the cyber security domain*, proving that the proposed attack is effective against the largest number of classifiers ever reviewed in a single study: RNN, LSTM, GRU, and their bidirectional and deep variants, and feed forward DNN, 1D CNN, SVM, random forest, logistic regression, GBDT, etc.

2 Background and Related Work

Most black-box attacks rely on the concept of *adversarial example transferability* [18]: Adversarial examples crafted against one model are also likely to be effective against other models, even when the models are trained on different datasets. This means that the adversary can train a *surrogate model*, which has decision boundaries similar to the original model, and perform a white-box attack on it. Adversarial examples that successfully fool the surrogate model are likely to fool the original model as well [11]. A different approach uses the confidence score of the targeted DNN to estimate its gradients directly instead of using the surrogate model's gradients to generate adversarial examples [3]. However, attacker knowledge of confidence scores (not required by our attack) is unlikely in black-box scenarios. *Decision based attack,* which uses only the target classifier's classes, without the confidence score, result in lower attack effectiveness and higher overhead [17].

In *mimicry attacks*, an attacker is able to code a malicious exploit that mimics the system calls' trace of benign code, thus evading detection [21]. Several methods were presented: (1) *Disguise attacks* - Causing benign system calls to generate malicious behavior by modifying only the system calls' parameters. (2) *No-op Attacks* - Adding semantic *no-ops* - system calls with no effect, or those with an irrelevant effect, e.g., opening a non-existent file. (3) *Equivalence attack* - Using a different system call sequence to achieve the same (malicious) effect.

The search for adversarial examples can be formalized as a minimization problem [18]:

$$\arg_{\mathbf{r}} \min f(\mathbf{x}+\mathbf{r}) \neq f(\mathbf{x}) \; s.t. \; \mathbf{x}+\mathbf{r} \in \mathbf{D} \tag{1}$$

The input $\mathbf{x}$, correctly classified by the classifier f, is perturbed with $\mathbf{r}$ such that the resulting adversarial example $\mathbf{x}+\mathbf{r}$ remains in the input domain $\mathbf{D}$, but is assigned a different label than $\boldsymbol{x}$.

A substitute model was trained with inputs generated by augmenting the initial set of representative inputs with their FGSM [4] perturbed variants, and then the substitute model was used to craft adversarial samples [11]. This differs from our paper in that: 1) It deals *only* with convolutional neural networks, as opposed to all state of the art classifiers, including RNN variants. 2) It deals with images and doesn't fit the attack requirements of the cyber security domain, i.e., not harming the malware functionality. 3) No end-to-end framework to implement the attack in the cyber-security domain was presented.

A white-box evasion technique for an Android static analysis malware classifier was implemented using the gradients to find the element whose addition would cause the maximum change in the benign score, and add this feature to the adversarial example [5]. In contrast to our work, this paper didn't deal with RNNs or dynamic features which are more challenging to add without harming the malware functionality. This study also did not focus on a generic attack that can affect many types of classifiers, as we do. Finally, our black-box assumption is more feasible than a white-box assumption. In Sect. 5.3 we created a black-box variant of this attack.

API call uni-grams were used as static features, as well [6]. A generative adversarial network (GAN) was trained to generate adversarial samples that would be classified as benign by the discriminator which uses labels from the black-box model. This attack doesn't fit sequence based malware classifiers (LSTM, etc.). In addition, the paper does not present a end-to-end framework which preserves the code's functionality. Finally, GANs are known for their unstable training process [1], making such an attack method hard to rely on.

A white-box adversarial example attack against RNNs, demonstrated against LSTM architecture, for sentiment classification of a movie reviews dataset was shown in [12]. The adversary iterates over the movie review's words $\boldsymbol{x}[i]$ in the review and modifies it as follows:

$$\mathbf{x}[i] = \arg\min_{\mathbf{z}} ||sign(\mathbf{x}[i] - \mathbf{z}) - sign(J_f(\mathbf{x})[i, f(\mathbf{x})])|| \; s.t. \; \mathbf{z} \in \mathrm{D} \tag{2}$$

where $f(\mathbf{x})$ is the original model label for $\mathbf{x}$, and $J_f(\mathbf{x})[i,j] = \frac{\partial f_j}{\partial x_i}(\mathbf{x})$. This differs from our paper in that: (1) We present a black-box attack, not a white-box attack. (2) We implement a practical cyber domain attack. For instance, we don't modify existing API calls, because while such an attack is relevant for reviews - it might damage a malware functionality which we wish to avoid. (3) We deal with multiple-feature classifiers, as in real world malware classifiers. (4) Our attack has better performance, as shown in Sect. 4.3.

Concurrently and independently from our work, a RNN GAN to generate invalid APIs and insert them into the original API sequences was proposed [7]. Gumbel-Softmax, a one-hot continuous distribution estimator, was used to deliver gradient information between the generative RNN and the substitute RNN. Null APIs were added, but while they were omitted to make the generated adversarial sequence shorter, they remained in the gradient calculation of the loss function. This decreases the attack effectiveness compared to our method (88% vs. 99.99% using our method, for an LSTM classifier). In contrast, our attack method doesn't have this difference between the substitute model and the black-box model, and our generated API sequences are shorter. This also makes our adversarial example faster. Unlike [7], which only focused on LSTM variants, we also show our attack's effectiveness against other RNN variants such as GRUs and conventional RNNs, bidirectional and deep variants, and non-RNN classifiers (including both feed forward networks and traditional machine learning classifiers such as SVM), making it truly generic. Moreover, the usage of Gumbel-Softmax approximation in [7] makes this attack limited to one-hot encoded inputs, while in our attack, any word embedding can be used, making it more generic. In addition, the stability issues associated with GAN training [1], which might not converge for specific datasets, apply to the attack method mentioned in [7] as well, making it hard to rely on. While such issues might not be visible when using a small dataset (180 samples in [7]), they become more apparent when using larger datasets like ours (500,000 samples). Finally, we developed an end-to-end framework, generating a mimicry attack (Sect. 5). While previous works inject arbitrary API call sequences that might harm the malware functionality (e.g., by inserting the *ExitProcess()* API call in the middle of the

malware code), our attack modifies the code such that the original functionality of the malware is preserved (Sect. 5.1). Moreover, our approach works in real world scenarios including hybrid classifiers/multiple feature types (Sect. 5.3) and API arguments (Sect. 5.2), non of which is addressed by [7].

3 Methodology

3.1 Black-Box API Call Based Malware Classifier

Our classifier's input is a sequence of API calls made by the inspected code. In this section, it uses only the API call type and not its arguments or return value. IDSs that verify the arguments tend to be much slower (4–10 times slower, in [19]). One might claim that considering arguments would make our attack easier to detect. This could be done, e.g., by looking for irregularities in the arguments of the API calls (e.g., invalid file handles, etc.) or by considering only successful API calls and ignoring 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, are used in our framework, as described in Sect. 5. We also discuss an extension of our attack that handles API call arguments in Sect. 5.2.

Since API call sequences can be long (some samples in our dataset have millions of API calls), it is impossible to train on the entire sequence at once due to GPU memory and training time constraints. Thus, we used a sliding window approach: Each API call sequence is divided into windows with size m. Detection is performed on each window in turn, and if any window is classified as malicious, the entire sequence is malicious. This method helps detect cases like malicious payloads injected into goodware (e.g., using Metasploit), where only a small subset of the sequence is malicious. We use one-hot encoding for each API call type in order to cope with the limitations of sklearn's implementation of decision trees and random forests[1]. The output of each classifier is binary (is the inspected code malicious or not). The tested classifiers and their hyper parameters are described in Sect. 4.2.

3.2 Black-Box API Call Based Malware Classifier Attack

The proposed attack has two phases: (1) creating a surrogate model using the target classifier as a black-box model, and (2) generating adversarial examples with white-box access to the surrogate model and using them against the attacked black-box model, by the transferability property.

[1] For details, see: https://roamanalytics.com/2016/10/28/are-categorical-variables-getting-lost-in-your-random-forests/.

Creating a Surrogate Model. We use Jacobian-based dataset augmentation, an approach similar to [11]. The method is specified in Algorithm 1.

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 (the recorded API call types) are known to the attacker. In order to learn decision boundaries similar to the black-box model while minimizing the number of black-box model queries, the synthetic training inputs are based on prioritizing directions in which the model's output varies. This is done by evaluating the sign of the Jacobian matrix dimension corresponding to the label assigned to input $\mathbf{x}$ by the black-box model, $sign(J_{\hat{f}}(\mathbf{x})[f(\mathbf{x})])$, as calculated by FGSM [4]. 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 $\mathbf{x} + \epsilon sign(J_{\hat{f}}(\mathbf{x})[f(\mathbf{x})])$ is added to the training set.

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:

$D_t = \{(\mathbf{x}, f(\mathbf{x}))|\mathbf{x} \in X_t\}$ # Label the synthetic dataset using the black-box model

$\hat{f}_t = train(A, D_t)$ # (Re-)Train the surrogate model

$X_{t+1} = \left\{\mathbf{x} + \epsilon sign(J_{\hat{f}_t}(\mathbf{x})[f(\mathbf{x})])|\mathbf{x} \in X_t\right\} \cup X_t$ # Perform Jacobian-based dataset augmentation

return $\hat{f}_T$

On each iteration we add a synthetic example to each existing sample. The surrogate model dataset size is: $|X_t| = 2^{t-1}|X_1|$

The samples used in the initial dataset, X_1, were randomly selected from the test set distribution, but they were not included in the training and test sets to prevent bias. X_1 should be representative so the dataset augmentation covers all decision boundaries to increase the augmentation's effectiveness. For example, if we only include samples from a single family of ransomware in the initial dataset, we will only be focusing on a specific area of the decision boundary, and our augmentation would likely only take us in a certain direction. However, as shown in Sect. 4.3, this doesn't mean that all of the malware families in the training set must be represented to achieve good performance.

Generating Adversarial Examples. An adversarial example is a sequence of API calls classified as malicious by the classifier that is perturbed by the addition of API calls, so that the modified sequence will be misclassified as benign. In order to prevent damaging the code's functionality, we cannot remove or modify API calls; we can only add additional API calls. In order to add API calls in

a way that doesn't hurt the code's functionality, we generate a *mimicry attack* (Sect. 5). Our attack is described in Algorithm 2.

Algorithm 2. Adversarial Sequence Generation

Input: f (black-box model), $\hat{f}$ (surrogate model), $\mathbf{x}$ (malicious sequence to perturb, of length l), n (size of adversarial sliding window), D (vocabulary)
for each sliding window $\mathbf{w}_j$ of n API calls in $\mathbf{x}$:
  $\mathbf{w}_j{}^* = \mathbf{w}_j$
  while $f(\mathbf{w}_j^*) = malicious$:
    Randomly select an API's position i in $\mathbf{w}$
    # Insert a new adversarial API in position $i \in \{1..n\}$:
    $\mathbf{w}_j^*[i] = \arg\min_{api} ||sign(\mathbf{w}_j{}^* - \mathbf{w}_j^*[1 : i-1] \perp api \perp \mathbf{w}_j^*[i : n-1]) - sign(J_{\hat{f}}(\mathbf{w}_j)[f(\boldsymbol{w}_j)])||$
  Replace $\mathbf{w}_j$ (in $\mathbf{x}$) with $\mathbf{w}_j{}^*$
return (perturbed) $\boldsymbol{x}$

D is the vocabulary of available features, that is, the API calls recorded by the classifier. The adversarial API call sequence length of l might be different than n, the length of the sliding window API call sequence that is used by the adversary. Therefore, like the prediction, the attack is performed sequentially on $\lceil \frac{l}{n} \rceil$ windows of n API calls. Note that the knowledge of m (the window size of the classifier, mentioned in Sect. 3.1) is not required, as shown in Sect. 4.3. $\perp$ is the concatenation operation. $\mathbf{w}_j^*[1 : i-1] \perp api \perp \mathbf{w}_j^*[i : n-1]$ is the insertion of the encoded API vector in position i of $\mathbf{w}_j^*$. The adversary randomly chooses i since he/she does not have any way to better select i without incurring significant statistical overhead. Note that an insertion of an API in position i means that the APIs from position $i..n$ ($\mathbf{w}_j^*[i : n]$) are "pushed back" one position to make room for the new API call, in order to maintain the original sequence and preserve the original functionality of the code. Since the sliding window has a fixed length, the last API call, $\mathbf{w}_j^*[n]$, is "pushed out" and removed from $\mathbf{w}_j^*$ (this is why the term is $\perp \mathbf{w}_j^*[i : n-1]$, as opposed to $\perp \mathbf{w}_j^*[i : n]$). The APIs "pushed out" from $\mathbf{w}_j$ will become the beginning of $\mathbf{w}_{j+1}$, so no API is ignored.

The newly added API call is $\mathbf{w}_j^*[i] = \arg\min_{api} ||sign(\mathbf{w}_j{}^* - \mathbf{w}_j^*[0 : i] \perp api \perp \mathbf{w}_j^*[i : n-1]) - sign(J_{\hat{f}}(\mathbf{w}_j)[f(\boldsymbol{w}_j)])||$. $sign(J_{\hat{f}}(\mathbf{w}_j)[f(\boldsymbol{w}_j)])$ gives us the direction in which we have to perturb the API call sequence in order to reduce the probability assigned to the malicious class, $f(\mathbf{x})$, and thus change the predicted label of the API call sequence. However, the set of legitimate API call embeddings is finite. Thus, we cannot set the new API to any real value. We therefore find the API call *api* in D whose insertion directs us closest to the direction indicated by the Jacobian as most impactful on the model's prediction. We iteratively apply this heuristic until we find an adversarial input sequence misclassified as benign. Note that in [12] the authors *replaced* a word in a movie review, so they only needed a single element from the Jacobian (for word i, which was replaced). All other words remained the same, so no gradient change took

place. In contrast, since we *add* an API call, all of the API calls following it shift their position, so we consider the aggregated impact.

While the proposed attack is designed for API call based classifiers, it can be generalized to any adversarial sequence generation. This generalization is a high performance in terms of attack effectiveness and overhead (Eqs. 4 and 5). This can be seen in Sect. 4.3, where we compare the proposed attack to [12] for the IMDB sentiment classification task. In Sect. 4.3 we show why the same adversarial examples generated against the surrogate model would be effective against both the black-box model and other types of classifiers due to the principle of *transferability*.

We assume that the attacker knows what API calls are available and how each of them is encoded (one-hot encoding in this paper). This is a commonly accepted assumption about the attacker's knowledge [8].

4 Experimental Evaluation

4.1 Dataset

Our dataset contains 500,000 files (250,000 benign samples and 250,000 malware samples), including the latest variants. We have ransomware families such as Cerber, Locky, Ramnit, Matsnu, Androm, Upatre, Delf, Zbot, Expiro, Ipamor. and other malware types (worms, backdoors, droppers, spyware, PUA, and viruses), each with the same number of samples, to prevent a prediction bias towards the majority class. 80% of the malware families' (like the Virut virus family) samples were distributed between the training and test sets, to determine the classifier's ability to generalize to samples from the same family. 20% of the malware families (such as the WannaCry ransomware family) were used only on the test set to assess generalization to an unseen malware family. The temporal difference between the training set and the test set is several months (meaning all test set samples are newer than the training set samples), based on VirusTotal's 'first seen' date. We labeled our dataset using VirusTotal, an on-line scanning service which contains more than 60 different security products. Our ground truth is that a malicious sample is one with 15 or more positive (i.e., malware) classifications from the 60 products. A benign sample is one with zero positive classifications. All samples with 1–14 positives were omitted to prevent false positive contamination of the dataset.

We ran each sample in Cuckoo Sandbox, a commonly-used malware analysis system, for two minutes per sample.[2] We parsed the JSON file generated by Cuckoo Sandbox and extracted the API call sequences generated by the

[2] Tracing only the first seconds of a program execution might not detect certain malware types, like "logic bombs" that commence their malicious behavior only after the program has been running some time. However, this can be mitigated both by classifying the suspension mechanism as malicious, if accurate, or by tracing the code operation throughout the program execution life-time, not just when the program starts.

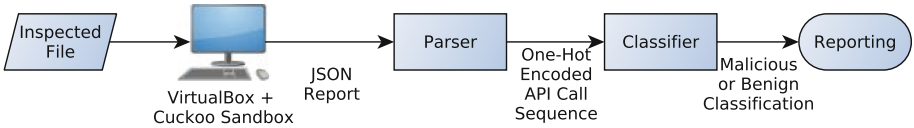


Fig. 1. Overview of the malware classification process

inspected code during its execution. The extracted API call sequences are the malware classifier's features. Although the JSON can be used as raw input for a neural network classifier (as done in [16]), we parsed it, since we wanted to focus only on API calls without adding other features, such as connected network addresses, which are also extracted by Cuckoo Sandbox.

The overview of the malware classification process is shown in Fig. 1. Figure 2a present a more detailed view of the classifier's structure.

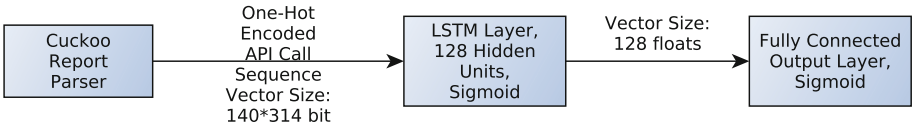
We run the samples on a VirtualBox's snapshot with Windows 8.1 OS,[3] since most malware target the Windows OS.

Cuckoo Sandbox is a tool known to malware writers, some of whom write code to detect if the malware is running in a Cuckoo Sandbox (or on virtual machines) and if so, the malware quit immediately to prevent reversing efforts. In those cases, the file is malicious, but its behavior recorded in Cuckoo Sandbox (its API call sequence) isn't malicious, due to its anti-forensic capabilities. To mitigate such contamination of our dataset, we used two countermeasures: (1) We applied YARA rules to find samples trying to detect sandbox programs such as Cuckoo Sandbox and omitted all such samples. (2) We considered only API call sequences with more than 15 API calls (as in [13]), omitting malware that, e.g., detect a VM and quit. This filtering left us with about 400,000 valid samples, after balancing the benign samples number. The final training set size is 360,000 samples, 36,000 of which serve as the validation set. The test set size is 36,000 samples. All sets are balanced between malicious and benign samples. One might argue that the evasive malware that apply such anti-VM techniques are extremely challenging and relevant. However, in this paper we focus on the adversarial attack. This attack is generic enough to work for those evasive malware as well, assuming that other mitigation techniques (e.g., anti-anti-VM), would be applied.

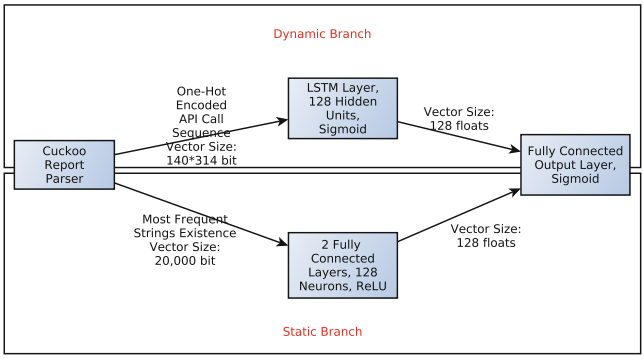
4.2 Malware Classifier Performance

No open source or commercial trail versions of API calls based deep learning intrusion detection systems are available, as such products target enterprises. Dynamic models are not available in VirusTotal as well. Therefore, we created our own black-box malware classifiers. This also allows us to evaluate the attack effectiveness (Eq. 4) against many classifier types.

[3] While it is true that the API calls sequence would vary across different OSs or configurations, both the black-box classifier and the surrogate model generalize across those differences, as they capture the "main features" over the sequence, which are not vary between OSs.



(a) Dynamic Classifier Architecture



(b) Hybrid Classifier Architecture

Fig. 2. Classifier architecture overview

We limited our maximum input sequence length to $m = 140$ API calls (longer sequence lengths, e.g., $m = 1000$, had no effect on the accuracy) and padded shorter sequences with zeros. A zero stands for a null API in our one-hot encoding. Longer sequences are split into windows of m API calls, and each window is classified in turn. If any window is malicious the entire sequence is considered malicious. Thus, the input of all of the classifiers is a vector of $m = 140$ API call types in one-hot encoding, using 314 bits, since there were 314 monitored API call types in the Cuckoo reports for our dataset. The output is a binary classification: malicious or benign. An overview of the LSTM architecture is shown in Fig. 2a.

We used the Keras implementation for all neural network classifiers, with TensorFlow used for the back end. XGBoost and Scikit-Learn were used for all other classifiers.

The loss function used for training was binary cross-entropy. We used the Adam optimizer for all of the neural networks. The output layer was fully-connected with sigmoid activation for all NNs. We fine-tuned the hyper parameters for all classifiers based on the relevant state of the art papers, e.g., window size from [13], number of hidden layers from [5,9], dropout rate from [9], and number of trees in a random forest classifier and the decision tree splitting criteria from [15]. For neural networks, a rectified linear unit, $ReLU(x) = max(0, x)$, was chosen as an activation function for the input and hidden layers due to its fast convergence compared to $sigmoid()$ or tanh(), and dropout was used to improve the generalization potential of the network. Training was conducted

Table 1. Classifier performance

Classifier type	Accuracy (%)	Classifier type	Accuracy (%)
RNN	97.90	Bidirectional GRU	98.04
BRNN	95.58	Fully-Connected DNN	94.70
LSTM	98.26	1D CNN	96.42
Deep LSTM	97.90	Random forest	98.90
BLSTM	97.90	SVM	86.18
Deep BLSTM	98.02	Logistic regression	89.22
GRU	97.32	Gradient boosted decision tree	91.10

for a maximum of 100 epochs, but convergence was usually reached after 15–20 epochs, depending on the type of classifier. Batch size of 32 samples was used.

The classifiers also have the following classifier-specific hyper parameters: DNN - Two fully-connected hidden layers of 128 neurons, each with ReLU activation and a dropout rate of 0.2; CNN - 1D ConvNet with 128 output filters, stride length of one, 1D convolution window size of three and ReLU activation, followed by a global max pooling 1D layer and a fully connected layer of 128 neurons with ReLU activation and a dropout rate of 0.2; RNN, LSTM, GRU, BRNN, BLSTM, bidirectional GRU - a hidden layer of 128 units, with a dropout rate of 0.2 for both inputs and recurrent states; Deep LSTM and BLSTM - Two hidden layers of 128 units, with a dropout rate of 0.2 for both inputs and recurrent states in both layers; Linear SVM and logistic regression classifiers - A regularization parameter $C = 1.0$ and L2 norm penalty; Random forest classifier - Using 10 decision trees with unlimited maximum depth and the Gini criteria for choosing the best split; Gradient boosted decision tree - Up to 100 decision trees with a maximum depth of 10 each.

We measured the performance of the classifiers using the accuracy ratio, which applies equal weight to both FP and FN (unlike precision or recall), thereby providing an unbiased overall performance indicator:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3}$$

where: TP are true positives (malicious samples classified as malicious by the black-box classifier), TN are true negatives, FP stands for false positives (benign samples classified as malicious), and FN are false negatives. The FP rate of the classifiers varied between 0.5-1%.[4]

The performance of the classifiers is shown in Table 1. The accuracy was measured on the test set, which contains 36,000 samples.

[4] The FP rate was chosen to be on the high end of production systems. A lower FP rate would mean lower recall either, due-to the trade-off between them, therefore making our attack even more effective.

As can be seen in Table 1, the LSTM variants are the best malware classifiers, accuracy-wise, and, as shown in Table 2, BLSTM is also one of the classifiers most resistant to the proposed attack.

4.3 Attack Performance

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

The *attack effectiveness* is the number of malware samples in the test set which were detected by the target classifier, for which the adversarial sequences generated by Algorithm 2 were misclassified by the target malware classifier.

$$attack\_effectiveness = \frac{|\{f(\mathbf{x}) = Malicious \vee f(\mathbf{x}^*) = Benign\}|}{|\{f(\mathbf{x}) = Malicious\}|} \tag{4}$$

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

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

We also consider the overhead incurred as a result of the proposed attack. The *attack overhead* is the average percentage of the number of API calls which were added by Algorithm 2 to a malware sample successfully detected by the target classifier, in order to make the modified sample classified as benign (therefore calculated only for successful attacks) by the black-box model:

$$attack\_overhead = avg(\frac{added\_APIs}{l}) \tag{5}$$

The average length of the API call sequence is: $avg(l) \approx 100,000$. The adversary chooses the architecture for the surrogate model without any knowledge of the target model's architecture. We chose a GRU surrogate model with 64 units (different from the malware classifiers used in Sect. 4.2), which has a shorter training time compared to other RNN variants, e.g., LSTM, which provides similar attack effectiveness. Besides the classifier's type and architecture, we also used a different optimizer for the surrogate model (ADADELTA instead of Adam). In our implementation, we used the CleverHans library.

Based on Eqs. 4 and 5, the proposed attack's performance is specified in Table 2 (average of three runs).

We can see in Table 2 that the proposed attack has very high effectiveness and low attack overhead against all of the tested malware classifiers. The attack effectiveness is lower for traditional machine learning algorithms, such as SVM, due to the greater difference between the decision boundaries of the GRU surrogate model and the target classifier. Randomly modifying APIs resulted in significantly lower effectiveness for all classifiers (e.g., 50.29% for fully-connected DNN).

As mentioned in Sect. 4.1, $|TestSet(f)| = 36,000$ samples, and the test set $TestSet(f)$ is balanced, so the attack performance was measured on: $|\{f(\mathbf{x}) =$

Table 2. Attack Performance

Classifier Type	Attack Effectiveness (%)	Additional API Calls (%)
RNN	100.0	0.0023
BRNN	99.90	0.0017
LSTM	99.99	0.0017
Deep LSTM	99.31	0.0029
BLSTM	93.48	0.0029
Deep BLSTM	96.26	0.0041
GRU	100.0	0.0016

Classifier Type	Attack Effectiveness (%)	Additional API Calls (%)
Bidirectional GRU	95.33	0.0023
Fully-Connected DNN	95.66	0.0049
1D CNN	100.0	0.0005
Random Forest	99.44	0.0009
SVM	70.90	0.0007
Logistic Regression	69.73	0.0007
Gradient Boosted Tree	71.45	0.0027

$Malicious|\mathbf{x} \in TestSet(f)\}| = 18,000$ samples. For the surrogate model we used a perturbation factor of $\epsilon = 0.2$ and a learning rate of 0.1. $|X_1| = 70$ samples were randomly selected from the test set of 36,000 samples. We used $T = 6$ surrogate epochs. Thus, as shown in Sect. 3.2, the training set size for the surrogate model is: $|X_6| = 2^5 * 70 = 2240$ samples; only 70 ($= |X_1|$) of the samples were selected from the test set distribution, and all of the others were synthetically generated. Using lower values, e.g., $|X_1| = 50$ or $T = 5$, achieved worse attack performance, while larger values do not improve the attack performance and result in a longer training time. The 70 samples from the test set don't cover all of the malware families in the training set; the effectiveness of the surrogate model is due to the synthetic data.

For simplicity and training time, we used $m = n$ for Algorithm 2, i.e., the sliding window size of the adversary is the same as that used by the black-box classifier. However, even if this is not the case, the attack effectiveness isn't degraded significantly. If $n > m$, the adversary would keep trying to modify different API calls' positions in Algorithm 2, until he/she modifies the ones impacting the black-box classifier as well, thereby increasing the attack overhead without affecting the attack effectiveness. If $n < m$, the adversary can modify only a subset of the API calls affecting the black-box classification, and this subset might not be diverse enough to affect the classification as desired, thereby reducing the attack effectiveness. The closer n and m are, the better the attack performance. For $n = 100, m = 140$, there is an average decrease of attack effectiveness from 99.99% to 99.98% for a LSTM classifier.

Comparison to Previous Work. Besides [7] which was written concurrently and independently from our work, [12] is the only recently published RNN adversarial attack. The differences between that attack and the attack addressed in this paper are mentioned in Sect. 2. We compared the attacks in terms of performance. The attack effectiveness for the IMDB dataset was the same (100%), but our attack overhead was better: 11.25 added words per review (on average), instead of 51.25 words using the method mentioned in [12].

4.4 Transferability for RNN Models

While transferability was covered in the past in the context of DNNs (e.g., [18]), to the best of our knowledge, this is the first time it is *evaluated* in the context of RNNs, proving that the proposed attack is generic, not just effective against a specific RNN variant, but is also transferable between RNN variants (like LSTM, GRU, etc.), feed forward DNNs (including CNNs), and even traditional machine learning classifiers such as SVM and random forest.

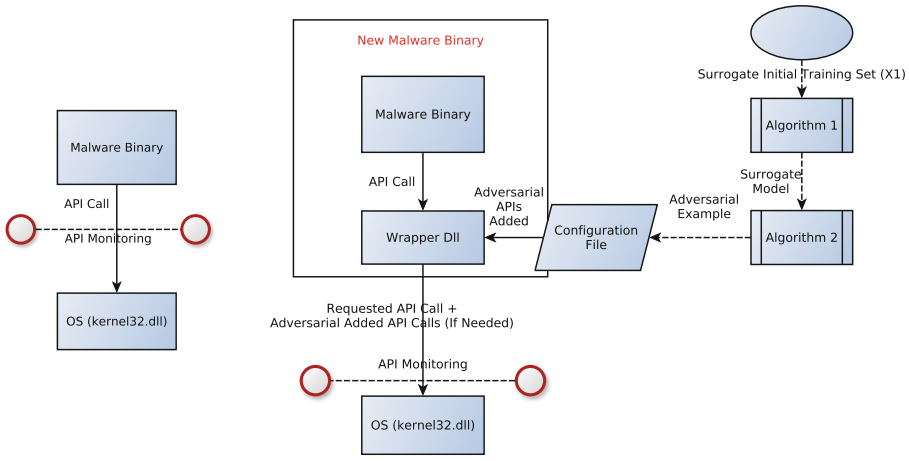
Two kinds of transferability are relevant to this paper: (1) the adversary can craft adversarial examples against a surrogate model with a different architecture and hyper parameters than the target model, and the same adversarial example would work against both [11], and (2) an adversarial example crafted against one target classifier type might work against a different type of target classifier.

Both forms of transferability are evaluated as follows: (1) As mentioned in Sect. 4.3, we used a GRU surrogate model. However, as can be seen in Table 2, the attack effectiveness is high, even when the black-box classifier is not GRU. Even when the black-box classifier is 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 the attack to be effective against both feed forward networks and traditional machine learning classifiers, as can be seen in the last six lines of Table 2. Our attack is therefore effective against all malware classifiers.

5 GADGET: End-to-End Attack Framework Description

To verify that an attacker can create an end-to-end attack using the proposed method (Sect. 3), we implemented **GADGET**: **G**enerative **A**pi a**D**versarial **G**eneric **E**xample by **T**ransferability framework. This is an end-to-end attack generation framework that gets a black-box classifier (f in Sect. 3) as an input, an initial surrogate model training set (X_1 in Algorithm 1), and a malware binary to evade f, and outputs a modified malware binary whose API call sequence is misclassified by f as benign, generating the surrogate model ($\hat{f}$ in Algorithm 1) in the process.

GADGET contains the following components: (1) Algorithms 1 and 2, implemented in Python, using Keras with TensorFlow back end, (2) A C++ Wrapper to wrap the malware binary and modify its generated API call sequence during run time, and (3) A Python script that wraps the malware binary with the



(a) Malware Binary, without GADGET

(b) Malware Binary, with GADGET

Fig. 3. Malware binary, with and without GADGET

above mentioned wrapper, making it ready to deploy. The components appear in Fig. 3.

Adding API Calls Without Damaging Functionality. As mentioned in Sect. 3.2, we implemented Algorithm 2 using a *mimicry attack* [21]. We discarded *equivalence attacks* and *disguise attacks* (Sect. 2), since they lack the flexibility needed to modify *every API call*, and thus are not robust enough to camouflage every malware. Therefore, we implemented a *no-op attack,* adding APIs which would have no effect on the code's functionality. 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 decided to implement the API addition by adding no-op API calls with valid parameters, e.g., reading 0 bytes from a valid file. This was more challenging to implement than calling APIs with invalid arguments (e.g., reading from an invalid file handle), since a different implementation should be used for each API. However, this effort can be done once and can subsequently be used for every malware, as we've done in our framework. This makes detecting those no-op APIs much harder, since the API call runs correctly, with a return value indicative of success. The functionality validation of the modified malware is discussed in Sect. 5. Further measures, such as randomized arguments, can be taken by the attacker to prevent the detection of the no-op APIs by analyzing the arguments of the API calls. Attacking a classifier with argument inputs is discussed in Sect. 5.

Implementing a Generic Framework. The requirements for the generic framework are: (1) there is no access to the malware source code (access only to

the malware binary executable), and (2) the same code should work for every adversarial sample: no adversarial example-specific code should be written. The reasons for these requirements are two-fold. First, adding the code as a wrapper, without changing the malware's business logic makes the framework more robust to modification of the malware classifier model, preventing another session of malware code modification and testing. Second, with the Malware-as-a-Service trend, not everyone who uses a malware has its code. Some ransomwares are automatically generated using minimal configuration (e.g., only the CNC server is modified by the user), without source code access. Thus, the GADGET framework expands the number of users that can produce an evasive malware from malware developers to every person that purchases a malware binary, making the threat much greater.

In order to meet those requirements, we wrap the malware binary from the outside with proxy code between the malware code and the OS DLLs implementing the API calls (e.g., kernel32.dll), fulfilling requirement #1. The wrapper code gets the adversarial sequence for the malware binary, generated by Algorithm 2, as a configuration file. The logic of this wrapper code is to hook all APIs that will be monitored by the malware classifier. These API calls are known to the attacker, as mentioned in Sect. 3.2. These hooks call the original APIs (to preserve the original malware functionality), keep track of the API sequence executed so far, and call the adversarial example's additional APIs in the proper position based on the configuration file (so they will be monitored by the malware classifier), instead of hard-coding the adversarial sequence to the code (fulfilling requirement #2). This flow is presented in Fig. 3b.

We generated a new malware binary that contains the wrapper's hooks by patching the malware binary's IAT using IAT Patcher, redirecting the IAT's API calls' addresses to the matching C++ wrapper API hook implementation. That way, if another hook (e.g., Cuckoo Sandbox) monitors the API calls, the adversarial APIs are already being called and monitored like any regular API call. To affect dynamic libraries, *LdrGetProcedureAddress()*\ *GetProcAddress*() hook has additional functionality: it doesn't return a pointer to the requested procedure, but instead returns a pointer to a wrapper function that implements the previously described regular static hook functionality around the requested procedure (e.g., returning a pointer to a wrapper around *WriteFile()* if "WriteFile" is the argument to *GetProcAddress*()). When the malware code calls the pointer, the hook functionality will be called, transparent to the user.

The code is POC and does not cover all corner cases, e.g., wrapping a packed malware, which requires special handling for the IAT patching to work, or packing the wrapper code to evade statically signing it as malicious (its functionality is implemented inline, without external API calls, so dynamic analysis of it is challenging). We avoided running Algorithm 2 inside the wrapper, and used the configuration file to store the modified APIs instead, thus preventing much greater overhead for the (wrapped) malware code.

5.1 Adversarial Example Functionality Validation

In order to *automatically* verify that we do not harm the functionality of the malware we modify, we monitored each sample in Cuckoo Monitor before and after the modification. We define the modified sample as *functionality preserving* if the API call sequence after the modification is the same as before the modification when comparing API type, return value and order of API calls, except for the added API calls, which return value should always be a success value. We found that all of the 18,000 modified samples are *functionality preserving.*

One of the families that did not exist in the training set was the WannaCry ransomware. This makes it an excellent candidate to *manually* analyze GADGET's output. First, we ran the sample via Cuckoo Sandbox and recorded its API calls. The LSTM malware classifier mentioned in Sect. 4.2 successfully detected it as malicious, although it was not part of the training set. Then we used GADGET to generate a new WannaCry variant, providing this variant the configuration file containing the adversarial sequence generated by Algorithm 2. We ran the modified WannaCry binary, wrapped with our framework and the configuration file, in Cuckoo Sandbox again, and fed the recorded API call sequence to the same LSTM malware classifier. This time, the malware classifier classification was benign, although the malicious functionality remains: files were still being encrypted by the new binary, as can be seen in the Cuckoo Sandbox snapshot and API call sequence. This means that the proposed attack was successful, end-to-end, without damaging WannaCry's functionality.

5.2 Handling API Arguments

We now modify our attack to evade classifiers that analyze arguments as well. In order to represent the API call arguments, we used MIST [20], as was done by other malware classifiers, e.g., MALHEUR [14]. MIST (Malware Instruction Set) is a representation for monitored behavior of malicious software, optimized for analysis of behavior using machine learning. Each API call translates to an instruction. Each instruction has levels of information. The first level corresponds to the category and name of a monitored API call. The following levels of the instruction contain different blocks of arguments. The main idea underlying this arrangement is to move "noisy" elements, such as the loading address of a DLL, to the end of an instruction, while discriminative patterns, such as the loaded DLL file path, are kept at the beginning of the instruction. We used MIST level 2. We converted our Cuckoo Sandbox reports to MIST using Cuckoo2Mist. We extracted a total of 220 million lines of MIST instructions from our dataset. Of those, only several hundred of lines were unique, i.e., different permutations of argument values extracted in MIST level 2. This means that most API calls differ only in arguments that are not relevant to the classification or use the same arguments. To handle MIST arguments, we modified our attack in the following way: Instead of one-hot encoding every API call type, we one-hot encoded every unique [API call type, MIST level 2 arguments] combination. Thus, *LoadLibrary* ("kernel32.dll") and *LoadLibrary* ("user32.dll") are now regarded as separate

APIs by the classifier. Our framework remains the same, where Algorithm 2 selects the most impactful combination instead of API type. However, instead of adding combinations that might harm the code's functionality (e.g., *ExitWindowsEx*()), we simply add a different API call type (the one with the minimal Jacobian value) in Algorithm 2, which would **not** cause this issue. We now assume a more informed attacker, who knows not just the exact encoding of each API type, but also the exact encoding of every argument combination. This is a reasonable assumption since arguments used by benign programs, like Windows DLLs file paths, are known to attackers [8].

Handling other API arguments (and not MIST level 2) would be similar, but require more preprocessing (word embedding, etc.) with a negligible effect on the classifier accuracy. Thus, focusing only on the most important arguments (MIST level 2) that can be used by the classifier to distinguish between malware and benign software, as done in other papers [9], proves that analyzing arguments is not an obstacle for the proposed attack.

5.3 Handling Hybrid Classifiers and Multiple Feature Types

Since our attack modifies only a specific feature type (API calls), combining several types of features might make the classifier more resistant to adversarial examples against a specific feature type. Some real-world next generation anti-malware products (such as SentinelOne) are hybrid classifiers, combining both static and dynamic features for a better detection rate.

Our attack can be extended to handle hybrid classifiers using two phases: (1) the creation of a *combined surrogate* model, including all features, using Algorithm 1, and (2) attacking *each feature type in turn* with a specialized attack, using the surrogate model. If the attack against a feature type fails, we continue and attack the next feature type until a benign classification by the target model is achieved or until all feature types have been (unsuccessfully) attacked.

We decided to use printable strings inside a PE file as our static features, as they are commonly used as the static features of state of the art hybrid malware classifiers [9], although any other modifiable feature type can be used. Strings can be used, e.g., to statically identify loaded DLLs and called functions, recognize modified file paths and registry keys, etc. Our architecture for the hybrid classifier, shown in Fig. 2b, is: (1) A dynamic branch that contains an input vector of 140 API calls, each one-hot encoded, inserted into a LSTM layer of 128 units, and sigmoid activation function, with a dropout rate of 0.2 for both inputs and recurrent states. (2) A static branch that contains an input vector of 20,000 Boolean values: for each of the 20,000 most frequent strings in the entire dataset, do they appear in the file or not? (analogous to a similar procedure used in NLP, which filters the least frequent words in a language). This vector is inserted into two fully-connected layers with 128 neurons, a ReLU activation function, and a dropout rate of 0.2 each. The 256 outputs of both branches are inserted into a fully-connected output layer with sigmoid activation function. Therefore, the input of the classifier is a vector containing 140 one-hot encoded APIs and 20,000 Boolean values, and the output is malicious or

benign classification. All other hyper parameters are the same as in Sect. 4.2. The surrogate model used has a similar architecture to the attacked hybrid model described above, but it uses a different architecture and hyper parameters: GRU instead of LSTM in the dynamic branch and 64 hidden units instead of 128 in both static and dynamic surrogate branches. Due to hardware limitations, we used just a subset of the dataset: 54,000 training samples and test and validation sets of 6,000 samples each. The dataset was representative and maintained the same distribution as the dataset described in Sect. 4.1. Trained on this dataset, a classifier using only the dynamic branch (Fig. 2a) reaches 92.48% accuracy on the test set, a classifier using only the static branch attains 96.19% accuracy, and a hybrid model, using both branches (Fig. 2b) achieves 96.94% accuracy, meaning that using multiple feature types improves the accuracy.

We used two specialized attacks: an attack against API call sequences and an attack against printable strings. The API sequence attack is Algorithm 2. When performing it against the hybrid classifier, without modifying the static features of the sample, the attack effectiveness (Eq. 4) decreases to 45.95%, compared to 96.03% against a classifier trained only on the dynamic features, meaning that the attack was mitigated by the use of additional features. The strings attack is a variant of the attack described in [5], using the surrogate model instead of the attacked model used in [5] to compute the gradients in order to select the string to add, while the adversarial sample's maliciousness is still tested against the attacked model, making this method a black-box attack. In this case, the attack effectiveness is 68.66%, compared to 77.33% against a classifier trained only on the static features. Finally, the combined attack's effectiveness against the hybrid model was 82.27%. Other classifier types provide similar results which are not presented here due to space limits.

We designed GADGET with the ability to handle a hybrid model, by adding its configuration file's static features' modification entries. Each such string is appended to the original binary before being IAT patched, either to the EOF or to a new section, where those modifications don't affect the binary's functionality.

To summarize, we have shown that while the usage of hybrid models decreases the specialized attacks' effectiveness, using our suggested hybrid attack achieves high effectiveness. While not shown due to space limits, the attack overhead isn't significantly affected.

6 Conclusions and Future Work

In this paper, we demonstrated a generic black-box attack, generating adversarial sequences against API call sequence based malware classifiers. Unlike previous adversarial attacks, we have shown an attack with a verified effectiveness against all relevant common classifiers: RNN variants, feed forward networks, and traditional machine learning classifiers. Therefore, this is a true black-box attack, which requires no knowledge about the classifier besides the monitored APIs. We also created the GADGET framework, showing that the generation of the

adversarial sequences can be done end-to-end, in a generic way, without access to the malware source code. Finally, we showed that the attack is effective, even when arguments are analyzed or multiple feature types are used. Our attack is *the first practical* ***end-to-end*** *attack* dealing with all of the subtleties of the cyber security domain, posing a concrete threat to next generation anti-malware products, which have become more and more popular. While this paper focus on API calls and printable strings as features, the proposed attack is valid for every modifiable feature type, static or dynamic.

Our future work will focus on two areas: defense mechanisms against such attacks and attack modifications to cope with such mechanisms. Due to space limits, we plan to publish an in depth analysis of various defense mechanisms in future work. The defense mechanisms against such attacks can be divided into two subgroups: (1) detection of adversarial examples, and (2) making the classifier resistant to adversarial attacks. To the best of our knowledge, there is currently no published and evaluated method to detect or mitigate RNN adversarial sequences. This will be part of our future work. We would also compare between the effectiveness of different surrogate models' architecture.

References

1. Arjovsky, M., Bottou, L.: Towards principled methods for training generative adversarial networks. In: ICLR (2017)
2. Carlini, N., Wagner, D.: Towards evaluating the robustness of neural networks. In: IEEE S&P (2017)
3. Chen, P.Y., Zhang, H., Sharma, Y., Yi, J., Hsieh, C.J.: Zoo: zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In: ACM Workshop on Artificial Intelligence and Security (2017)
4. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: ICLR (2015)
5. Grosse, K., Papernot, N., Manoharan, P., Backes, M., McDaniel, P.: Adversarial examples for malware detection. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10493, pp. 62–79. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66399-9_4
6. Hu, W., Tan, Y.: Generating adversarial malware examples for black-box attacks based on GAN. ArXiv e-prints, abs/1702.05983 (2017)
7. Hu, W., Tan, Y.: Black-box attacks against RNN based malware detection algorithms. ArXiv e-prints, abs/1705.08131 (2017)
8. Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I.P., Tygar, J.D.: Adversarial machine learning. In: ACM Workshop on Security and Artificial Intelligence (2011)
9. Huang, W., Stokes, J.W.: MtNet: a multi-task neural network for dynamic malware classification. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 399–418. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40667-1_20
10. Papernot, N., McDaniel, P., Jha, S.H., Fredrikson, M., Celik, Z.B., Swami, A.: The limitations of deep learning in adversarial settings. In: IEEE European Symposium on Security and Privacy (2016)
11. Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z.B., Swami, A.: Practical black-box attacks against machine learning. In: ASIA CCS (2017)

12. Papernot, N., McDaniel, P., Swami, A., Harang, R.: Crafting adversarial input sequences for recurrent neural networks. In: IEEE MILCOM (2016)
13. Pascanu, R., Stokes, J.W., Sanossian, H., Marinescu, M., Thomas, A.: Malware classification with recurrent networks. In: IEEE ICASSP (2015)
14. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. J. Comput. Secur. **19**, 639–668 (2011)
15. Rosenberg, I., Gudes, E.: Bypassing system calls-based intrusion detection systems. Concurr. Comput.: Pract. Exp. (2016)
16. Rosenberg, I., Sicard, G., David, E.O.: DeepAPT: nation-state APT attribution using end-to-end deep neural networks. In: Lintas, A., Rovetta, S., Verschure, P.F.M.J., Villa, A.E.P. (eds.) ICANN 2017. LNCS, vol. 10614, pp. 91–99. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68612-7_11
17. Rosenberg, I., Shabtai, A., Rokach, L., Elovici, Y.: Low resource black-box end-to-end attack against state of the art API call based malware classifiers, arXiv preprint arXiv:1804.08778 (2018)
18. Szegedy, C., et al.: Intriguing properties of neural networks. In: ICLR (2014)
19. Tandon, G., Chan, P.K.: On the learning of system call attributes for host-based anomaly detection. Int. J. Artif. Intell. Tools **15**, 875–892 (2006)
20. Trinius, P., Willems, C., Holz, T., Rieck, K.: A malware instruction set for behavior-based analysis. In: Sicherheit (2010)
21. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: ACM CCS (2002)