Deceiving End-to-End Deep Learning Malware Detectors using Adversarial Examples

Felix Kreuk, Assi Barak, Shir Aviv, Moran Baruch, Benny Pinkas, Joseph Keshet
Dept. of Computer Science, Bar-Ilan University, Israel
felixkreuk@gmail.com

Abstract

In recent years, deep learning has shown performance breakthroughs in many prominent problems. However, this comes with a major concern: deep networks have been found to be vulnerable to adversarial examples. Adversarial examples are modified inputs designed to cause the model to err. In the domains of images and speech, the modifications are untraceable by humans, but affect the model's output. In binary files, small modifications to the byte-code often lead to significant changes in its functionality/validity, therefore generating adversarial examples is not straightforward. We introduce a novel approach for generating adversarial examples for discrete input sets, such as binaries. We modify malicious binaries by injecting a small sequence of bytes to the binary file. The modified files are detected as benign, while preserving their malicious functionality. We applied this approach to an end-to-end convolutional neural malware detector and present a high evasion rate. Moreover, we showed that our payload is transferable within different positions of the same file and across different files and file families.

1 Introduction

End-to-end (E2E) deep-learning has recently become ubiquitous in the field of AI. In E2E deep-learning, manual feature engineering is replaced with a unified model that maps raw input to output. This approach has been successfully applied to many prominent problems [1, 2, 3, 4, 5], leading to state-of-the-art models. Naturally, it is becoming a technology of interest in malware detection [6, 7, 8], where manual feature extraction is gradually being augmented/replaced by E2E models that take raw inputs.

Despite its success, deep networks have shown to be vulnerable to *adversarial examples* (AEs). AEs are artificial inputs that are generated by modifying legitimate inputs so as to fool classification models. In the fields of image and speech, modified inputs are considered adversarial when they are indistinguishable by humans from the legitimate inputs, and yet they fool the model [9, 10, 11]. Such attacks were demonstrated in [12, 13, 14]. Conversely, discrete sequences are inherently different than speech and images, as changing one element in the sequence may completely alter its meaning. For example, one word change in a sentence may hinder its grammatical validity or meaning. Similarly, in a binary file, where the input is a discrete sequence of bytes, changing one byte may result in invalid byte-code or different runtime functionality.

In malware detection, an AE is a binary file that is generated by modifying an existing malicious binary. While the original file is correctly classified as *malicious*, its modified version is misclassified as *benign*. Recently, a series of works [15, 16, 17] have shown that AEs cause catastrophic failures in malware detection systems. These works assumed a malware detection model trained on a set of handcrafted features such as file headers and API calls, whereas our work is focused on *raw* binaries. In parallel to our work, [18] suggested an AE generation method for binaries by injecting a sequence of bytes. Our work differs in two main aspects: (i) the injection procedure in [18] is based

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

on end-of-file injections, while our method utilises multiple injection locations; (ii) the evasion rate of [18] is linearly correlated with the length of the injected sequence, while the evasion rate of our method is invariant to the length of the injected sequence.

In this work, we analyze malware detectors based on Convolutional Neural Networks (CNNs), such as the one described in [8]. Such detectors are not based on handcrafted features, but on the entire *raw binary* sequence. Results suggest that E2E deep learning based models are susceptible to AE attacks.

Contributions. We re-evaluate the reliability of E2E deep neural malware detectors beyond traditional metrics, and expose their vulnerability to AEs. We propose a novel technique for generating AEs for discrete sequences by injecting local perturbations. As raw binaries cannot be directly perturbed, local perturbations are generated in an embedding space and reconstructed back to the discrete input space. We demonstrate that the same payload can be injected into different locations, and can be effective when applied to different malware files and families.

2 Problem Setting

In this section, we formulate the task of malware detection and set the notations for the rest of the paper. A binary input file is composed of a sequence of bytes. We denote the set of all bytes as $\mathbf{X} \subseteq [0, N-1]$, where $N = 256$. A binary file is a sequence of L bytes $\mathbf{x} = (x_1, x_2, ...x_L)$, where $x_i \in \mathbf{X}$ for all $1 \leq i \leq L$. Note that the length L varies for different files, thus L is not fixed. We denote by $\mathbf{X}^*$ the set of all finite-length sequences over $\mathbf{X}$, hence $\mathbf{x} \in \mathbf{X}^*$. The goal of a malware detection system is to classify an input binary file as malicious or benign. Formally, we denote by $f_\theta : \mathbf{X}^* \rightarrow [0, 1]$ the malware detection function implemented as a neural network with a set of parameters θ. If the output of $f_\theta(\mathbf{x})$ is greater than 0.5 then the prediction is considered as 1 or *malicious*, otherwise the prediction is considered as 0 or *benign*.

Given a malicious file, which is correctly classified as malicious, our goal is to modify it, while retaining its runtime functionality, so that it will be classified as benign. Formally, denote by $\tilde{\mathbf{x}}$ the modified version of the malicious file $\mathbf{x}$. Note that if $\mathbf{x}$ is classified as malicious, $f_\theta(\mathbf{x}) > 0.5$, then we would like to design $\tilde{\mathbf{x}}$ such that $f_\theta(\tilde{\mathbf{x}}) < 0.5$, and the prediction is benign. In this work, we focus on generating AEs for E2E models, specifically, *MalConv* architecture [8], which is a CNN based malware detector. The model is composed of embedding layer denoted $\mathbf{M} \in \mathbb{R}^{N \times D}$, mapping from $\mathbf{X}$ to $\mathbf{Z} \subseteq \mathbb{R}^D$. This is followed by a Gated-Linear-Unit [19]. The output is then passed to a *temporal-max-pooling* layer, which is a special case of *max-pooling*. For more details see [8].

3 Generating Adversarial Examples

Generating AEs is usually done by adding small perturbations to the original input in the direction of the gradient. While such methods work for continuous input sets, they fail in the case of discrete input sets, as bytes are arbitrarily represented as scalars in $[0, N-1]$. Hence, we generate AEs in a continuous embedding space $\mathbf{Z}$ and reconstruct them to $\mathbf{X}$.

Generation Given an input file that is represented in the embedded domain $\mathbf{z} \in \mathbf{Z}^*$, an AE is a perturbed version $\tilde{\mathbf{z}} = \mathbf{z} + \delta$, where $\delta \in \mathbf{Z}^*$ is an additive perturbation that is generated to preserve functionality of the corresponding $\mathbf{x}$, yet cause f_θ to predict an incorrect label. We assume that f_θ was trained and the parameters θ were fixed. An adversarial example is generated by solving the following problem: $\tilde{\mathbf{z}} = \arg\min_{\tilde{\mathbf{z}}: \|\tilde{\mathbf{z}} - \mathbf{z}\|_p \leq \epsilon} \bar{\ell}\big(\tilde{\mathbf{z}}, \tilde{y}; \theta\big)$, where $\tilde{y}$ is the desired target label, ϵ represents the strength of the adversary, and p is the norm value. Assuming the loss function $\bar{\ell}$ is differentiable, when using the max-norm, $p = \infty$, the solution is $\tilde{\mathbf{z}} = \mathbf{z} - \epsilon \cdot \text{sign}\big(\nabla_{\tilde{\mathbf{z}}} \bar{\ell}(\tilde{\mathbf{z}}, \tilde{y}; \theta)\big)$, which corresponds to the Fast Gradient Sign Method proposed (FGSM) in [20]. When choosing $p = 2$ we get $\tilde{\mathbf{z}} = \mathbf{z} - \epsilon \cdot \nabla_{\tilde{\mathbf{z}}} \bar{\ell}(\tilde{\mathbf{z}}, \tilde{y}; \theta)$. After $\tilde{\mathbf{z}}$ was crafted, we need to reconstruct the new binary file $\tilde{\mathbf{x}}$. The most straightforward approach is to map each $\mathbf{z}_i$ to its closest neighbor in the embedding matrix $\mathbf{M}$.

Preserving Functionality Utilizing the above approach directly might not preserve the functionality of the file as the changes could affect the whole file. To mitigate that we propose to contain modifications to a small chunk of bytes (payload).

Algorithm 1: Adversarial Examples Generation

Data: A binary file $\mathbf{x}$, target label y, conv size c
$k \leftarrow c + (c - \text{len}(\mathbf{x}) \bmod c)$ *// k is the payload size*
$\mathbf{x}^{\text{payload}} \sim \mathbf{U}(0, N-1)^k$
$\mathbf{z}^{\text{payload}},\ \mathbf{z},\ \tilde{\mathbf{z}}^{\text{payload}} \leftarrow \mathbf{M}(\mathbf{x}^{\text{payload}}),\ \mathbf{M}(\mathbf{x}),\ \mathbf{z}^{\text{payload}}$
$\tilde{\mathbf{z}}^{\text{new}} \leftarrow [\mathbf{z}; \tilde{\mathbf{z}}^{\text{payload}}]$
while $g_\theta(\tilde{\mathbf{z}}^{new}) > 0.5$ **do**
  $\tilde{\mathbf{z}}^{\text{payload}} \leftarrow \tilde{\mathbf{z}}^{\text{payload}} - \epsilon \cdot \text{sign}\Big(\nabla_{\mathbf{z}} \bar{\ell}^*(\tilde{\mathbf{z}}^{\text{new}}, \tilde{y}; \theta)\Big)$
  $\tilde{\mathbf{z}}^{\text{new}} \leftarrow [\mathbf{z}; \tilde{\mathbf{z}}^{\text{payload}}]$
end
for $i \leftarrow 0$ **to** *len*$(\mathbf{x})$ **do**
  $\tilde{\mathbf{x}}_i^{\text{new}} \leftarrow \arg\min_j d(\tilde{\mathbf{z}}_i^{\text{new}}, M_j)$
end
return $\tilde{\mathbf{x}}^{\text{new}}$

This payload can be injected to the original file in one of two ways: (i) *mid-file injection* - the payload is placed in existing contingent unused bytes of sections where the physical size is greater than the virtual size; (ii) *end-of-file injection* - treating the payload as a new section and appending it to the file. Either approach would rDesult in a valid and runnable code as the payload is inserted into non-executable code sections. We applied both methods and report results in Section 4.

Algorithm 1 presents pseudo-code for generating a payload and injecting it at end-of-file. In this case, we append a uniformly random sequence of bytes of length k, $\mathbf{x}^{\text{payload}} \in \mathbf{X}^k$, to the original file $\mathbf{x}$. We then embed the new binary $\mathbf{x}^{\text{new}} = [\mathbf{x}; \mathbf{x}^{\text{payload}}]$, to get $\mathbf{z}^{\text{new}} = [\mathbf{z}; \mathbf{z}^{\text{payload}}]$. Next, we iteratively perturb the appended segment $\mathbf{z}^{\text{payload}}$ and stop when f_θ misclassifies $\mathbf{z}^{\text{new}}$. By perturbing only the appended segment, we ensure that $\mathbf{z}$ is kept intact and the functionality of $\mathbf{x}$ is preserved. This results in $\mathbf{x}^{\text{new}}$ that behaves identically to $\mathbf{x}$ but evades detection by f_θ. For the setting of mid-file injections we use the same perturbed payload $\mathbf{x}^{\text{payload}}$ depicted in the previous paragraph, and inject it into contingent unused bytes where the physical size is greater than the virtual size.

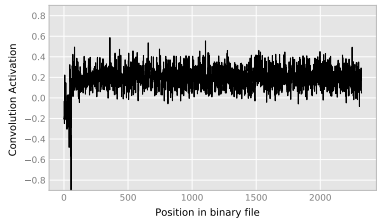
4 Experimental Results

Dataset eWe evaluated the effectiveness of our method using a dataset for the task of malware detection. Two sets of files were used: (i) The first set is a collection of benign binary files that were collected as follows. We deployed a fresh Windows 8.1 32-bit image and installed software from over 50 different vendors using "ninite" [21]. We then collected all exe/dll with size 32KB or greater. This resulted in 7,150 files, which were labeled as *benign*; (ii) The second set is a collection of 10,866 malicious binary files from 9 different malware families[1]. These files were taken from the *Microsoft Kaggle 2015 dataset*[2] [22] and were labeled as *malicious*. The entire data was shuffled and split into training, validation and test sets using 80/10/10 ratio respectfully. Note that in the Kaggle 2015 dataset files were missing PE headers. Therefore, for the generation of AEs, we used an unseen set of malicious binary files from *VirusShare* with PE headers intact.

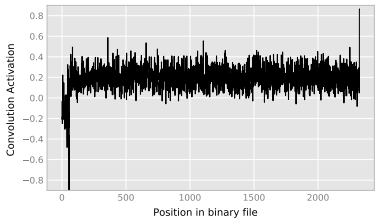
Detection Evasion We trained a CNN E2E malware detector [8] on the Kaggle 2015 dataset, until it reached a classification accuracy of 98% on a validation set. This corresponded to accuracy of 97% on the test set. The gradients of this model were later used for the the FGSM. For the generation of AEs, we appended/injected to the file a payload denoted as $\mathbf{x}^{\text{payload}}$ that was initiated uniformly at random. Then, using iterative-FGSM, we perturbed the embedded representation of the payload, until the whole-file was misclassified. The FGSM procedure was implemented using $p = \infty$ and $p = 2$. We evaluated the effectiveness of our method on the test set. The injection procedure resulted in an evasion rate of 99.21% and 98.83% for $p = 2$ and $p = \infty$ respectfully. These results were achieved under the constraint of retaining the original functionality and the payload length varied from 500 to 999 bytes (depending on the length of file modulo 500).

[1] Ramnit, Lollipop, Kelihos, Vundo, Simda, Tracur, Gatak, Obfuscator.ACY

[2] We thank Microsoft for allowing the use of their data for research.



(a) Average convolution activation.



(b) Average convolution activation.

Evading detection can be explained by a shift in the model's attention. This phenomenon was previously demonstrated in [23]. To exemplify this point, we depicted the average activation of the convolutional layer along the position in a specific file before and after injecting the payload in Figure 1b and Figure 1b respectfully. Recall that in the *MalConv* architecture, a temporal-max-pooling operation is performed after the convolution. It is seen from Figure 1a and Figure 1b that for the AE the max was achieved at the payload location, acting as a "distraction" to the malicious part of the code.

Payload We now turn to introduce a set of experimental results to provide further analysis regarding the payloads' properties. For these experiments we obtained 223 (previously-unseen) malicious binary files from 5 malware families: *Gatak, Kelihos, Kryptic, Simda* and *Obfuscator.ACY*. The original header was included to test the effectiveness of our method on files "seen in the wild".

Spatial invariance - OWe now demonstrate that the generated payload can fool the exploited network at multiple positions in the file. An attacker can benefit from such invariance as it allows for flexibility in the injection position. For each file, we generated an adversarial version where the payload was appended at the end-of-file. Then, the injected payload was re-positioned to a mid-file position. To fit the payload inside the convolution window, the position was a multiple of $ca + len(\mathbf{x})\text{mod } c$, where $a \in \mathbb{N}$, c is the convolution kernel size, and $len(\mathbf{x})$ is the file size. We then evaluated the files using the exploited model before and after re-positioning. 100% of files were still misclassified as benign. Meaning, the payload can be shifted to various positions and still fool the exploited model.

File transferability - We tested the transferability of the payload across different files: we injected a payload generated for file A to file B . We then evaluated these files using the malware detector: 75% were still misclassified as benign. Meaning, the same payload can be injected to multiple files and evade detection with high probability.

Payload size - Our payload size depends on $len(x) \mod c$, where $len(x)$ is the size of the binary, and c is the convolution kernel size. For the case of *MalConv* our maximal payload size is 999 bytes. To validate the minimalism property, we carried an attack using payload with maximal sizes of 1000, 1500, 2000, 2500 and recorded evasion rates of 99.02%-99.21% for $p = \infty$ and 98.04%-98.83% for $p = 2$. We conclude that the evasion rate is not affected by the payload length given c.

Entropy - Entropy-based malware detectors flag suspicious sections by comparing their entropy with standard thresholds [24]. Regular text sections have an entropy of 4.5-5, entropy of above 6 is considered compressed/encrypted code. We analyzed the change in entropy caused by payload injection by calculating the entropy before and after injection. We found that the entropy change from 3.75 to 4.1 on average, and remained within an unsuspicious range.

Functionality Preserving - The functionality of the modified files is preserved by modifying non-executable sections. To further verify the functionality of the reconstructed binary files, we evaluated and compared the behavior graph of the files before and after the payload injection and found that they are identical.

5 Future Work

In future work, we would like to decouple the generation process from the model's embedding layer to produce black-box attacks. We would also like to asses the reliability of real-world hybrid systems that use a wider array of detection techniques.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] R. Socher, B. Huval, B. Bath, C. D. Manning, and A. Y. Ng, "Convolutional-recursive deep learning for 3d object classification," in *Advances in Neural Information Processing Systems*, 2012, pp. 656–664.

[4] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International Conference on Machine Learning*, 2016, pp. 173–182.

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[6] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 611–626. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin

[7] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 99–116. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua

[8] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware Detection by Eating a Whole EXE," *ArXiv e-prints*, Oct. 2017.

[9] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[10] S. M. Moosavi Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, no. EPFL-CONF-218057, 2016.

[11] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533*, 2016.

[12] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 39–57.

[13] C. Xiao, J.-Y. Zhu, B. Li, W. He, M. Liu, and D. Song, "Spatially transformed adversarial examples," *arXiv preprint arXiv:1801.02612*, 2018.

[14] F. Kreuk, Y. Adi, M. Cisse, and J. Keshet, "Fooling end-to-end speaker verification by adversarial examples," in *ICASSP*, 2018.

[15] L. R. Ishai Rosenberg, Asaf Shabtai and Y. Elovici, "Generic black-box end-to-end attack against rnns and other API calls based malware classifiers," *CoRR*, vol. abs/1707.05970, 2017. [Online]. Available: http://arxiv.org/abs/1707.05970

[16] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, "Adversarial examples for malware detection," in *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds., vol. 10493. Springer, 2017, pp. 62–79. [Online]. Available: https://doi.org/10.1007/978-3-319-66399-9_4

[17] W. Hu and Y. Tan, "Black-box attacks against rnn based malware detection algorithms," *arXiv preprint arXiv:1705.08131*, 2017.

[18] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," *arXiv preprint arXiv:1803.04173*, 2018.

[19] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier, "Language modeling with gated convolutional networks," *arXiv preprint arXiv:1612.08083*, 2016.

[20] C. S. Ian J Goodfellow, Jonathon Shlens, "Explaining and harnessing adversarial examples." ICLR, 2015.

[21] P. Swieskowski and S. Kuzins. (2018) Ninite. https://ninite.com/.

[22] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," *arXiv preprint arXiv:1802.10135*, 2018.

[23] P. Stock and M. Cisse, "Convnets and imagenet beyond accuracy: Explanations, bias detection, adversarial examples and model criticism," *arXiv preprint arXiv:1711.11443*, 2017.

[24] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security & Privacy*, vol. 5, no. 2, 2007.