Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables

Bojan Kolosnjaji*, Ambra Demontis†, Battista Biggio†‡, Davide Maiorca†, Giorgio Giacinto†‡,
Claudia Eckert*, and Fabio Roli†‡
*Technical University of Munich
kolosnjaji@sec.in.tum.de, claudia.eckert@in.tum.de,
†University of Cagliari, Italy
{ambra.demontis, battista.biggio, davide.maiorca, giacinto, roli}@diee.unica.it
‡Pluribus One, Italy

***Abstract*—Machine learning has already been exploited as a useful tool for detecting malicious executable files. Data retrieved from malware samples, such as header fields, instruction sequences, or even raw bytes, is leveraged to learn models that discriminate between benign and malicious software. However, it has also been shown that machine learning and deep neural networks can be fooled by evasion attacks (also known as adversarial examples), i.e., small changes to the input data that cause misclassification at test time. In this work, we investigate the vulnerability of malware detection methods that use deep networks to learn from raw bytes. We propose a gradient-based attack that is capable of evading a recently-proposed deep network suited to this purpose by only changing few specific bytes at the end of each malware sample, while preserving its intrusive functionality. Promising results show that our adversarial malware binaries evade the targeted network with high probability, even though less than 1% of their bytes are modified.**

I. Introduction

Detection of malicious binaries still constitutes one of the major quests in computer security [22]. To counter their growing number, sophistication and variability, machine learning-based solutions are becoming increasingly adopted also by anti-malware companies [13]. Although past research work on binary malware detection has explored the use of traditional learning algorithms on n-gram-based, system-call-based, or behavior-based features [1], [19], [21], [26], more recent work has considered the possibility of using deep-learning algorithms on raw bytes as an effective way to improve accuracy on a wide range of samples [18]. The rationale is that such algorithms should automatically learn the relationships among the various sections of the executable file, thus extracting a number of features that correctly represent the role of specific byte groups in specific sections (e.g., if a byte belongs to the code section or simply to a section pointer).

While machine learning can be used to map the features from malware analysis to a decision on classifying programs as benign or malicious, this process is also vulnerable to adversaries that may manipulate the programs in order to bypass detection. It has been shown that deep-learning methods and neural networks are particularly vulnerable to these evasion attacks, also known as *adversarial examples*, i.e., input samples specifically manipulated to be misclassified [3], [23]. While the existence of adversarial examples has been widely demonstrated on computer-vision tasks (see, e.g., [5]), it is common to consider that it is not trivial to practically implement the same attack on executable files [2], [18], [25], as one mistake at changing the code section or the headers may completely compromise the file functionality.

In this work, we show that it is instead possible to evade deep-learning systems for malware detection by performing few changes to malware binaries without compromising their functionality. In particular, we introduce a gradient-based attack to generate *adversarial malware binaries*, i.e., evasive variants of malware binaries. The underlying idea of our attack is to manipulate some bytes in each malware to maximally increase the probability that the input sample is classified as benign. Although our attack can ideally manipulate every byte in the file, in this work we only consider the manipulation of padding bytes appended at the end of the file, to guarantee that the intrusive functionality of the malware binary is preserved. We nevertheless discuss throughout the paper which other bytes and sections of the file can be modified while still preserving its functionality. Our attack is conceived against *MalConv*, i.e., a deep neural network trained on raw bytes for malware binary detection, recently proposed by Raff et al. [18]. To our knowledge, this is among the first attacks proposed at the *byte-level* scale, similarly to [14], as most work in adversarial machine learning for malware detection has considered injection and removal of API calls or similar characteristics [3], [6], [9], [11], [12], [16], [24], [27], [28].

We perform our experiments on 13,195 Windows Portable Executable (PE) samples, showing that the accuracy of *MalConv* is decreased by over 50% after injecting only $10,000$ padding bytes in each malware sample, i.e., less than 1% of the bytes passed as input to the deep network. We also show that our attack outperforms random byte injections, and explain why being capable of manipulating even fewer bytes *within* the file content (rather than appending them at the end) may drastically increase the success of the attack.

With this paper, we aim to claim that it may be very difficult to deploy a robust detection methodology that blindly analyzes the executable bytes. Learning algorithms can not automatically learn the hard-to-manipulate, *invariant* information that reliably characterizes malware, if not proactively designed to

ISBN 978-90-827970-1-5 © EURASIP 2018

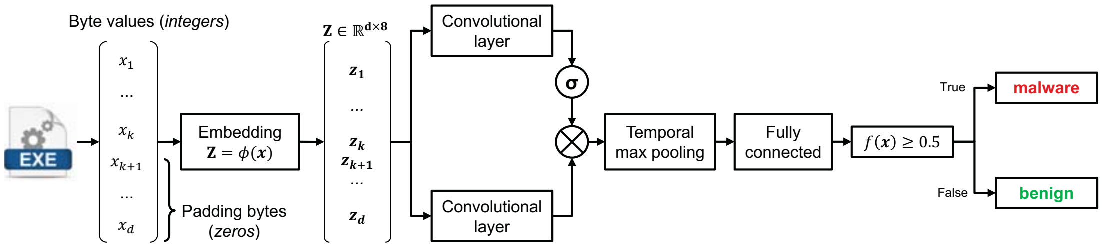

Fig. 1: Architecture of the *MalConv* deep network for malware binary detection [18].

keep that into account [4], either by providing proper training examples or encoding a-priori knowledge of which bytes can be maliciously manipulated. Robustness against adversarial attacks provided by well-motivated miscreants is thus a crucial design characteristic. This work provides preliminary evidence of this issue, which we aim to further investigate in the future.

II. Portable Executable (PE) Format

We provide here a brief description of the structure of PE files, and the prominent approaches that can be used to practically change their bytes.

A. PE File Basics

PE files are executables that are characterized by an organized structure, which will be briefly described in the following (more details can be found in [17]).

Header. A data structure that contains basic information on the executable, such as the number and size of its sections, the operating system type, and the role performed by the file itself (e.g., a dynamically-linked library). Such header is organized in three sub-sections: (i) a DOS header, as the first bytes of a PE executable essentially represent a DOS program; (ii) the true PE header; (iii) an *optional header* which contains information such as the entry point of the file (e.g., the address of the first loaded instruction), the size of the code sections, the magic number, etc.

Section Table. A table that describes the characteristics of each file section, with a special focus on a virtual address range that represents how that section will be mapped in memory once the process is loaded. It also contain clear references to where the data generated by the compiler/assembler are stored for each section.

Data. The actual data related to each section. The most important ones are `.text` (which contains code instructions), `.data` (which contains the initialized global and static variables), `.rdata` (which contains constants and additional directories such as debug), and `.idata` (which contains information about the used imports in the file).

B. Manipulating PE Files

Manipulating PE files with the goal of preserving their functionality is in general a non-trivial task, as it can be quite easy to compromise them by even changing one byte. As reported by Anderson et al. [2], possible and simple solutions to perform manipulations include either *injecting* bytes in part of the files that are not used (e.g., adding new sections that are never reached by the code), or directly *appending* them at the end of the file. Of course, these strategies are prone to detection by simply inspecting the file header or the section table (in the simplest case of byte appending), or by checking if such sections are accessed by the code itself (in case of more complex injections).

There are some special cases in which it is possible to directly perform changes to the executable without compromising its functionality. A popular example is changing bytes related to debug information, which are simply used as reference by code developers. Packing (i.e., compressing part of the executable that is then decompressed at runtime) is another possibility, which is however not adequate to perform fine-grained modifications to the file.

More complex changes require precise knowledge of the architecture of the file, and may be not always feasible. For instance, changing the `.text` section may entirely break the program. However, more trivial changes can be quite dangerous for the file integrity; for example, adding bytes to an existing section would require changing the header and section table accordingly. For the sake of simplicity, in this paper we only refer to byte appending as modification strategy.

III. Deep Learning for Malware Binary Detection

The deep neural network attacked in this paper is the *MalConv* network proposed by Raff et al. [18], depicted in Fig. 1. Let us denote with $\mathcal{X} = \{0, \ldots, 255\}$ the set of possible integer values corresponding to a byte. Then, the aforementioned network works as follows. The k bytes $(x_1, \ldots, x_k) \in \mathcal{X}^k$ extracted from the input file are padded with zeros to form an input vector $\boldsymbol{x}$ of d elements (if $k < d$, otherwise the first d bytes are only considered without padding). This ensures that the input vector provided to the network has a fixed dimensionality regardless of the length of the input file. Each byte x_j is then embedded as a vector $\boldsymbol{z}_j = \phi(x_j)$ of 8 elements (through a fixed mapping ϕ learned by the network during training). This amounts to encoding $\boldsymbol{x}$ as a matrix $\mathbf{Z} \in \mathbb{R}^{d \times 8}$. This matrix is then fed to two convolutional layers, respectively using Rectified Linear Unit (ReLU) and sigmoidal activation functions, which are subsequently combined through *gating* [8]. This mechanism multiplies element-wise the matrices outputted by the two layers, to avoid the vanishing gradient problem caused by

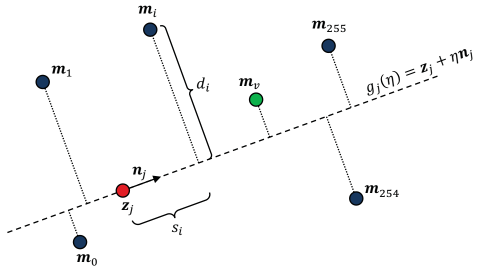

Fig. 2: Representation of an exemplary two-dimensional byte embedding space, showing the distance d_i and the projection length s_i of each byte $\boldsymbol{m}_i$ with respect to the line $g_j(\eta)$. In this case, the padding byte $\boldsymbol{z}_j$ will be modified by the attack algorithm to $\boldsymbol{m}_v$, as $d_v = \min_{i:s_i>0} d_i$, i.e., $\boldsymbol{m}_v$ is the closest byte with a projection on $g_j(\eta)$ aligned with $\boldsymbol{n}_j$.

sigmoidal activation functions. The obtained values are then fed to a temporal max pooling layer which performs a 1-dimensional max pooling, followed by a fully-connected layer with ReLU activations. To avoid overfitting, Raff et al. [18] use DeCov regularization [7], which encourages a non-redundant data representation by minimizing the cross-covariance of the fully-connected layer outputs. The deep network eventually outputs the probability of $\boldsymbol{x}$ being malware, denoted in the following with $f(\boldsymbol{x})$. If $f(\boldsymbol{x}) \geq 0.5$, the input file is thus classified as malware (and as benign, otherwise).

IV. Adversarial Malware Binaries

We discuss here how to manipulate a source malware binary $\boldsymbol{x}_0$ into an *adversarial* malware binary $\boldsymbol{x}$ by appending a set of carefully-selected bytes after the end of file. As in previous work on evasion of machine-learning algorithms [3], our attack aims to minimize the confidence associated to the malicious class (i.e., it maximizes the probability of the adversarial malware sample being classified as benign), under the constraint that $q_{\max}$ bytes can be injected. Note that, to append $q_{\max}$ bytes to $\boldsymbol{x}_0$, we have to ensure that $k + q_{\max} \leq d$, where k is the size of $\boldsymbol{x}_0$ (i.e., the number of informative bytes it contains) without considering the padding zeros. This means that the maximum number of bytes that can be injected by the attack is $q = \min(k + q_{\max}, d) - k$.[1] This can be characterized as the following constrained optimization problem:

$$\min_{\boldsymbol{x}} \quad f(\boldsymbol{x}) \,, \tag{1}$$

$$\text{s.t.} \quad d(\boldsymbol{x}, \boldsymbol{x}_0) \leq q \,, \tag{2}$$

where the distance function $d(\boldsymbol{x}, \boldsymbol{x}_0)$ counts the number of padding bytes in $\boldsymbol{x}_0$ that are modified in $\boldsymbol{x}$.

We solve this problem with a gradient-descent algorithm similar to that originally proposed in [3], by optimizing the padding bytes one at a time. Ideally, we would like to compute the gradient of the objective function f with respect to the padding byte under optimization. However, the *MalConv* architecture is not differentiable in an end-to-end manner, as the embedding layer is essentially a lookup table that maps each input byte x_j to an 8-dimensional vector $\boldsymbol{z}_j = \phi(x_j)$. We denote the embedding matrix containing all bytes with $\mathbf{M} \in \mathbb{R}^{256 \times 8}$, where the row $\boldsymbol{m}_i \in \mathbb{R}^8$ represents the embedding of byte i, for $i = 0, \ldots, 255$. To overcome the non-differentiability issue of the embedding layer, we first compute the (negative) gradient of f (as we aim to minimize its value) with respect to embedded representation $\boldsymbol{z}_j$, denoted with $\boldsymbol{w}_j = -\nabla_\phi(x_j) \in \mathbb{R}^8$. We then define a line $g_j(\eta) = \boldsymbol{z}_j + \eta \boldsymbol{n}_j$, where $\boldsymbol{n}_j = \boldsymbol{w}_j / \|\boldsymbol{w}_j\|_2$ is the normalized (negative) gradient direction. This line is parallel to $\boldsymbol{w}_j$ and passes through $\boldsymbol{z}_j$. The parameter η characterizes its geometric locus, i.e., by varying $\eta \in (-\infty, \infty)$ one obtains all the points belonging to this line. Ideally, assuming that the gradient remains constant, the point $\boldsymbol{z}_j$ will be gradually shifted towards the direction $\boldsymbol{n}_j$ while minimizing f. We thus consider a good heuristic to replace the padding byte x_j with that corresponding to the embedded byte $\boldsymbol{m}_i$ closest to the line g_j, provided that its projection on the line is aligned with $\boldsymbol{n}_j$, i.e., that $s_i = \boldsymbol{n}_j^\top (\boldsymbol{m}_i - \boldsymbol{z}_j) > 0$. Recall that the distance of each embedded byte $\boldsymbol{m}_i$ to the line g_j can be computed as $d_i = \|\boldsymbol{m}_i - (\boldsymbol{z}_j + s_i \cdot \boldsymbol{n}_j)\|_2$. A conceptual representation of this discretization process is shown in Fig. 2. This procedure is then repeated for each modifiable padding byte (starting from a random initialization), and up to a maximum number of iterations T, as described in Algorithm 1.

Algorithm 1 Adversarial Malware Binaries

```
Input: $\boldsymbol{x}_0$, the input malware (with $k$ informative bytes, and $d-k$ padding bytes); $q$, the maximum number of padding bytes that can be injected (such that $k + q \leq d$); $T$, the maximum number of attack iterations.
Output: $\boldsymbol{x}'$: the adversarial malware example.
Set $\boldsymbol{x} = \boldsymbol{x}_0$.
Randomly set the first $q$ padding bytes in $\boldsymbol{x}$.
Initialize the iteration counter $t = 0$.
repeat
    Increase the iteration counter $t \leftarrow t + 1$.
    for $p = 1, \ldots, q$ do
        Set $j = p + k$ to index the padding bytes.
        Compute the gradient $\boldsymbol{w}_j = -\nabla_\phi(x_j)$.
        Set $\boldsymbol{n}_j = \boldsymbol{w}_j / \|\boldsymbol{w}_j\|_2$.
        for $i = 0, \ldots, 255$ do
            Compute $s_i = \boldsymbol{n}_j^\top (\boldsymbol{m}_i - \boldsymbol{z}_j)$.
            Compute $d_i = \|\boldsymbol{m}_i - (\boldsymbol{z}_j + s_i \cdot \boldsymbol{n}_j)\|_2$.
        end for
        Set $x_j$ to $\arg\min_{i:s_i>0} d_i$.
    end for
until $f(\boldsymbol{x}) < 0.5$ or $t \geq T$
return $\boldsymbol{x}'$
```

Generation of Adversarial Malware Binaries. Although the padding bytes are generated by manipulating the input vector $\boldsymbol{x}$, creating the corresponding executable file without corrupting the malicious functionality of the source file is quite

[1] Note that $q \leq 0$ if $k \geq d$, which means that no byte can be manipulated by this attack.

ISBN 978-90-827970-1-5 © EURASIP 2018

easy, as also explained in Sect. II and in [2]. It is however worth mentioning that our attack is general, i.e., it can be used to manipulate any byte within the input file. To this end, one can first identify which bytes can be manipulated without affecting the file functionality, and then optimize them (instead of optimizing only the padding bytes).

V. Experiments

We practically reproduced the deep neural network proposed in [18], and performed the evasion attacks according to the algorithm described in Sect. IV. In the following, we first describe the employed setup, and then we discuss the results obtained by comparing the efficiency of the proposed gradient-based method with trivial random byte addition.

Dataset. We employed a dataset composed of $9,195$ malware samples, which were retrieved from a number of sources including `VirusShare`, `Citadel` and `APT1`. Additionally, to evaluate the performances of the network we employed $4,000$ benign samples, randomly retrieved and downloaded from popular search engines.

Network Performances. We evaluated the performances of the deep neural network by splitting our dataset into a training and a test set, each of them containing 50% of the samples of the initial dataset. To avoid results that could be biased by a specific training-test division, we repeated this process three times and averaged the results. Under this setting, we obtained an average precision of $92.83 \pm 5.56\%$ and an average recall of $84.68 \pm 11.71\%$ (mean and standard deviation).[2]

A. *Results on Evasion Attacks*

We performed our tests by modifying 200 randomly-chosen malicious test samples with Algorithm 1 to generate the corresponding *adversarial malware binaries*. As for Algorithm 1, we set the maximum number of attack iterations $T = 20$, and the maximum number of injected bytes $q_{max} = 10,000$. As a result, we chose all malware samples that satisfied the condition $k + q_{max} \leq d$, where k is the file size and $d = 10^6$. While some malware samples are larger than this threshold, the average file length in our sample set is 339,803 bytes. The attack was performed by appending, at the end of each file, bytes that were chosen according to two different strategies: a *random* attack injecting random byte values, and our *gradient-based* attack strategy. To verify the efficacy of the attack, we measured for each amount of added bytes the average *evasion rate*, i.e., the percentage of malicious samples that managed to evade the network. Fig. 3 provides the attained results as the number of bytes progressively increases, averaged on the three aforementioned training-test splits. Notably, adding random bytes is not really effective to evade the network. Conversely, our gradient-based attack allows evading *MalConv* in 60% of the cases when $10,000$ padding bytes are modified, even if this amounts to manipulating less than 1% of the input bytes.

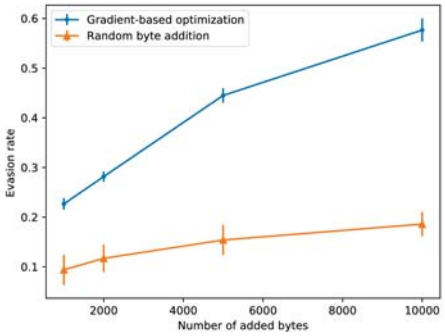

Fig. 3: Evasion rate against number of injected bytes.

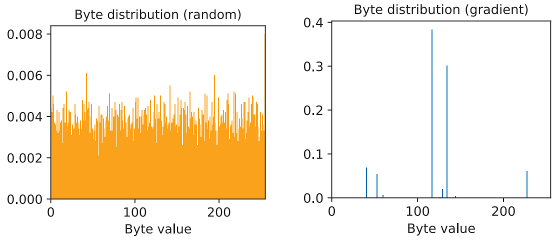

Fig. 4: Distribution of the $10,000$ padding byte values injected by the random (*left*) and gradient-based (*right*) attacks into a randomly-picked malware sample.

The success of our gradient-based approach relies on the fact that it guides the decision of which bytes to add, thus creating an organized *padding byte pattern* specific to each sample. To better clarify this concept, in Fig. 4 we consider a sample that successfully evaded the network, and show the distribution of the $10,000$ bytes added by the two attacks. Note how, in the optimized case, only a small group of byte values is consistently injected. This shows that the gradient guides the choice of specific byte values that are repeatedly injected, identifying a clear *padding byte pattern* for evasion.

B. *Limitations of Our Analysis*

We discuss here some limitations related to our analysis. First, we have only evaluated the effectiveness of our attack using 200 malware samples due to the computational complexity of the attack, leaving a more extensive analysis as future work. In comparison to [18], we have then employed a smaller dataset, and considered an input file size d of 10^6 rather than $2 \cdot 10^6$. These are both factors that may facilitate evasion of *MalConv*. Conversely, we found that appending bytes to the end of the file reduces the effectiveness of the gradient-based approach. To better realize this, in Fig. 5 we show that the average norm of the gradient $\boldsymbol{w}$ computed over all attack samples is much higher for the *first bytes* in the file. This is reasonable, as files have different lengths, and the probability of finding informative (non-padding) bytes for discriminating malware and benign files decreases as we move away from the first bytes. From the attacker's perspective, this also means that modifying the first bytes may cause a much larger decrease of $f(\boldsymbol{x})$ and, consequently, a much higher probability of evasion. However, as described in Sect. II, modifying bytes within

[2] Note that these results are different from [18], as we use different data and experimental settings.

ISBN 978-90-827970-1-5 © EURASIP 2018

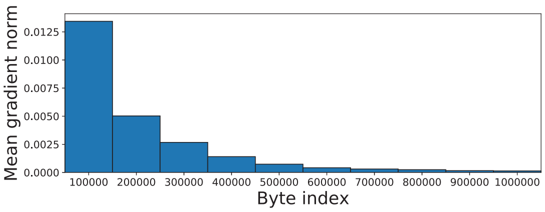

Fig. 5: Mean gradient norm (per byte) over all attack samples.

the file while preserving functionality may be quite complex, depending on the specific file and the content of its sections. This is definitely an interesting avenue for future research.

VI. Conclusions and Future Work

In this work, we evaluated the robustness of neural network-based malware detection methods that use raw bytes as input. We proposed a general gradient-based approach that chooses which bytes should be modified in order to change the classifier decision. We applied it by injecting a small number of optimized bytes at the end of a set of malicious samples, and we used them to attack the *MalConv* network architecture, attaining a maximum evasion rate of 60%.

These results question the adequateness of byte-based analysis from an adversarial perspective. In particular, the use of deep learning on raw byte sequences may give rise to novel security vulnerabilities. Binary-based approaches are usually based on the hypothesis that all sections have the same importance from the *learning perspective*. However, such claim is challenged by the fact that there are typically strong semantic differences between sections containing instructions (e.g., `text`) and those containing, for example, debug information. Hence, performing manipulations directly on the targeted files might be easier than expected.

In future work, we plan to particularly investigate this issue, by exploring fine-grained, automatic changes to executables that may be more difficult to counter than the injection of padding bytes at the end of file. We also plan to repeat the assessment of this paper on a larger dataset, more representative of recent malware trends (as advocated by Rossow et al. [20]). We anyway believe that our work highlights a severe vulnerability of deep learning-based malware detectors trained on raw bytes, highlighting the need for developing more robust and principled detection methods. Notably, recent research on the interpretability of machine-learning algorithms may also offer interesting insights towards this goal [10], [15].

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research, under grant 16KIS0327 (IUNO); by the EU H2020 project ALOHA, under the European Union's Horizon 2020 research and innovation programme (grant no. 780788); and by the PIS- DAS project, funded by the Sardinian Regional Administration (CUP E27H14003150007).

References

[1] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In COMPSAC '04, pp. 41–42, Washington, DC, USA, 2004. IEEE CS.

[2] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth. Evading machine learning malware detection. In *Black Hat*, 2017.

[3] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *ECML PKDD, Part III*, vol. 8190 of *LNCS*, pp. 387–402. Springer Berlin Heidelberg, 2013.

[4] B. Biggio, G. Fumera, and F. Roli. Security evaluation of pattern classifiers under attack. *IEEE TKDE*, 26(4):984–996, 2014.

[5] B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *ArXiv e-prints*, 2018.

[6] L. Chen, S. Hou, and Y. Ye. Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks. In *ACSAC*, pp. 362–372. ACM, 2017.

[7] M. Cogswell, F. Ahmed, R. Girshick, L. Zitnick, and D. Batra. Reducing Overfitting in Deep Networks by Decorrelating Representations. *arXiv:1511.06068 [cs, stat]*, Nov. 2015.

[8] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier. Language Modeling with Gated Convolutional Networks. *arXiv:1612.08083 [cs]*, Dec. 2016.

[9] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Trans. Dependable and Secure Computing*, In press.

[10] F. Doshi-Velez and B. Kim. Towards A Rigorous Science of Interpretable Machine Learning. *ArXiv e-prints*, 2017.

[11] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel. Adversarial examples for malware detection. In *ESORICS (2)*, vol. 10493 of *LNCS*, pp. 62–79. Springer, 2017.

[12] A. Huang, A. Al-Dujaili, E. Hemberg, and U.-M. O'Reilly. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. *ArXiv e-prints*, 2018.

[13] Kaspersky. Machine learning for malware detection, 2017.

[14] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. *CoRR*, abs/1802.04528, 2018.

[15] Z. C. Lipton. The mythos of model interpretability. In *ICML Workshop on Human Interpretability in Machine Learning*, pp. 96–100, 2016.

[16] D. Maiorca, B. Biggio, M. E. Chiappe, and G. Giacinto. Adversarial detection of flash malware: Limitations and open issues. *ArXiv*, 2017.

[17] M. Pietrek. Peering inside the PE: A tour of the win32 portable executable file format, 1994.

[18] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. Malware detection by eating a whole EXE. *arXiv preprint arXiv:1710.09435*, 2017.

[19] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In DIMVA '08, pp. 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *IEEE Symp. Security and Privacy*, pp. 65–79. IEEE, 2012.

[21] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *IEEE Symp. Security and Privacy*, SP '01, pp. 38–49, Washington, DC, USA, 2001. IEEE CS.

[22] Symantec. Internet security threat report, 2017.

[23] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *Int'l Conf. Learn. Repr.*, 2014.

[24] L. Tong, B. Li, C. Hajaj, C. Xiao, and Y. Vorobeychik. Hardening classifiers against evasion: the good, the bad, and the ugly. *ArXiv*, 2017.

[25] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *23rd NDSS*. The Internet Society, 2016.

[26] G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In DIMVA'13, pp. 41–61, Berlin, Heidelberg, 2013. Springer-Verlag.

[27] W. Yang, D. Kong, T. Xie, and C. A. Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *ACSAC*, pp. 288–302. ACM, 2017.

[28] F. Zhang, P. Chan, B. Biggio, D. Yeung, and F. Roli. Adversarial feature selection against evasion attacks. *IEEE T. Cyb.*, 46(3):766–777, 2016.