Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection

LUCA DEMETRIO, CSEC, Università degli Studi di Genova, ITA
SCOTT E. COULL, FireEye, Inc.
BATTISTA BIGGIO, PraLab, Università degli studi di Cagliari, ITA and Pluribus One, ITA
GIOVANNI LAGORIO, CSEC, Università degli Studi di Genova, ITA
ALESSANDRO ARMANDO, CSEC, Università degli Studi di Genova, ITA
FABIO ROLI, PraLab, Università degli Studi di Cagliari, ITA and Pluribus One, ITA

Recent work has shown that adversarial Windows malware samples - also referred to as adversarial *EXE*mples in this paper - can bypass machine learning-based detection relying on static code analysis by perturbing relatively few input bytes. To preserve malicious functionality, previous attacks either add bytes to existing non-functional areas of the file, potentially limiting their effectiveness, or require running computationally-demanding validation steps to discard malware variants that do not correctly execute in sandbox environments. In this work, we overcome these limitations by developing a unifying framework that not only encompasses and generalizes previous attacks against machine-learning models, but also includes two novel attacks based on practical, functionality-preserving manipulations to the Windows Portable Executable (PE) file format, based on injecting the adversarial payload by respectively extending the DOS header and shifting the content of the first section. Our experimental results show that these attacks outperform existing ones in both white-box and black-box attack scenarios by achieving a better trade-off in terms of evasion rate and size of the injected payload, as well as enabling evasion of models that were shown to be robust to previous attacks. To facilitate reproducibility and future work, we open source our framework and all the corresponding attack implementations. We conclude by discussing the limitations of current machine learning-based malware detectors, along with potential mitigation strategies based on embedding domain knowledge coming from subject-matter experts naturally into the learning process.

CCS Concepts: • **Computing methodologies** → **Machine Learning**; • **Security and privacy** → *Malware and its mitigation.*

Additional Key Words and Phrases: adversarial examples, malware detection, evasion, semantics-invariant manipulations

ACM Reference Format:
Luca Demetrio, Scott E. Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. 2020. Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection. In *Proceedings of ACM TO EDIT conference.* ACM, New York, NY, USA, 23 pages. https://doi.org/TOEDIT

1 INTRODUCTION

Machine learning (ML) has become an important aspect of modern cybersecurity due to its ability to detect new threats far earlier than signature-based defenses. While many cybersecurity companies use machine learning models[1][2][3][4] in their respective product offerings, creating and maintaining these models often represents a significant cost in terms of

[1] https://www.sophos.com/products/intercept-x/tech-specs.aspx
[2] https://www.fireeye.com/blog/products-and-services/2018/07/malwareguard-fireeye-machine-learning-model-to-detect-and-prevent-malware.html
[3] https://www.kaspersky.com/enterprise-security/wiki-section/products/machine-learning-in-cybersecurity
[4] https://www.avast.com/technology/ai-and-machine-learning

expertise and labor in developing useful features to train on, particularly when we consider that each new file type may require a completely different set of features to provide meaningful classification. With this in mind, researchers have recently proposed end-to-end deep learning models that operate directly on the raw bytes of the input files and automatically learn useful feature representations during training, without external knowledge from subject-matter experts. Several byte-based malware detection models for Windows PE files, for example, have demonstrated efficacy that is competitive with traditional ML models [5, 20] (Sect. 2).

While the use of end-to-end deep learning makes it easy to create new models for a variety of file types by simply exploiting the vast number of labeled samples available to such organizations, it also opens up the possibility of attacking these models using *adversarial evasion* techniques popularized in the image classification space. In particular, recent work has shown how an attacker can create what we call here *adversarial EXEmples*, i.e. , Windows malware samples carefully perturbed to evade learning-based detection while preserving malicious functionality [2, 4, 7, 8, 13, 16, 22, 24].

Unlike adversarial evasion attacks in other problem areas, such as image classification, manipulating malware while simultaneously preserving its malicious payload can be difficult to accomplish. In particular, each perturbation made to the input bytes during the attack process may lead to changes in the underlying syntax, semantics, or structure that could prevent the binary from executing its intended goal. To address this problem, the attacker can take one of two approaches: apply invasive perturbations and use dynamic analysis methods (e.g. emulation) to ensure that functionality of the binary is not compromised [4, 23], or focus the perturbation on areas of the file that do not impact functionality [7, 8, 13, 16, 24] (e.g. appending bytes). Naturally, this leads to a trade-off between strong yet time-consuming attacks on one extreme, and weaker but more computationally-efficient attacks on the other.

In this work, we overcome these limitations by proposing a unifying framework, called **RAMEn** (Sect. 3), built on top of a family of *practical* manipulations to the Windows Portable Executable (PE) file format that can alter the structure of the input malware without compromising its semantics. Our framework encompasses and generalizes previously-proposed attacks against learning-based Windows malware detectors based on static code analysis, including both white-box attacks that exploit full knowledge of the target algorithm, and black-box attacks that only require query access to it. The practical, functionality-preserving manipulations defined in our framework are not limited to perturbing bytes at the end of malware programs and do not require computationally-demanding validation steps during the attack optimization, thereby overcoming the limitations of existing attacks. In particular, we encode two novel practical manipulations that exploit the ambiguity in the specifications of the Windows PE file format: *Extend*, which enlarges the DOS header, thus enabling manipulation of these extra DOS bytes; and *Shift*, which shifts the content of the first section, carving additional space for the adversarial payload.

Our experimental results (Sect. 4) show that these attacks outperform existing ones in both white-box and black-box attack scenarios against different machine-learning models, deep network architectures, activation functions (i.e. linear vs. non-linear models), and training regimes. In particular, our *Extend* and *Shift* attacks enable evading some models that are not affected by previously-proposed attacks, while generally achieving a better trade-off in terms of evasion rate and size of the injected payload; they create fully-functional, evasive malware by perturbing roughly 2% of the input bytes against most of the considered classifiers.

An additional finding from our experimental analysis is that, while dataset size and activation functions do not seem to play a significant role in improving adversarial robustness, model architecture does, at least to some extent, with all attacks working well against Raff et al.'s MalConv classifier [20] and only content-shifting attacks working well against Coull et al.'s classifier [5], possibly due to the importance of spatial locality in its design. This identifies a promising line of research towards strengthening models against adversarial attacks through inclusion of additional structure in the training process.

We conclude the paper by discussing related work (Sect. 5) along with promising research directions to improve robustness of learning-based Windows malware detectors against adversarial attacks (Sect. 6). Besides considering network architectures that exploit spatial locality, we discuss other potential strategies to embed external domain knowledge directly into the learning process (e.g., via suitable constraints and loss functions) with the goal of learning more meaningful and robust representations from data [17]. We believe that this novel learning paradigm may help significantly improve adversarial robustness of such models, while at the same time exploiting knowledge from domain experts in an efficient manner.

To summarize, we highlight our contributions below.

- We propose RAMEn, a general framework for expressing white-box and black-box adversarial attacks on learning-based Windows malware detectors based on static code analysis.
- We propose two novel attacks based on practical, functionality-preserving manipulations, named *Extend* and *Shift*, which improve the trade-off between the probability of evasion and the amount of manipulated bytes in both white-box and black-box attack settings.
- We release the implementations of all the aforementioned white-box and black-box attacks encompassed by our framework (including previous attacks, *Extend* and *Shift*), as an open-source project available at https://github.com/zangobot/secml_malware.
- We identify promising future research directions towards improving adversarial robustness of learning-based Windows malware detectors leveraging static code analysis.

2 BACKGROUND

Before diving into the details of our proposed attack framework, we first provide some necessary background on the Windows Portable Executable (PE) file format (Section 2.1) and the malware classifiers that we will examine in our experiments (Section 2.2).

2.1 Executable File Format

The *Windows Portable Executable* (PE)[5] format specifies how executable programs are stored as a file on disk. The OS loader parses this structure and maps the code and data into memory, following the directives specified by the header of the file. We show the components of the format[6] in Figure 1.

- *DOS Header and Stub*: contains metadata for loading the executable inside a DOS environment and a few instructions that will print "*This program cannot be run in DOS mode*" if executed inside a DOS environment. These two components have been kept to maintain compatibility with older Microsoft operating systems. From the perspective of a modern application, the only relevant locations contained inside the DOS Header are (i) the magic number `MZ`, a two-byte signature of the file, and (ii) the four-byte integer at offset `0x3c`, that works as a pointer to the actual header. If one of these two values is altered for some reason, the program is considered corrupted, and it will not be executed by the OS;

[5] https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

[6] https://en.wikipedia.org/wiki/Portable_Executable under Attribution 4.0 International (CC BY 4.0)https://creativecommons.org/licenses/by/4.0/)

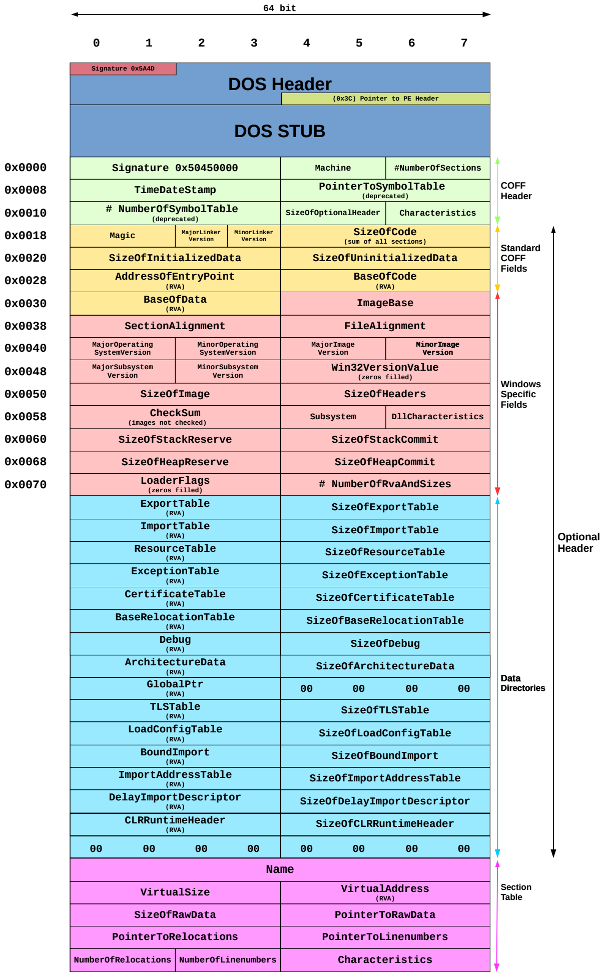


Fig. 1. The Windows PE file format. Each colored section describes a particular characteristic of the program.

- *PE Header*: contains the magic number PE and the characteristics of the executable, such as the target architecture that can run the program, the size of the header and the attributes of the file;
- *Optional Header*: contains the information needed by the OS for initializing the loading program. It also contains offsets that point to useful structures, like the Import Table needed by the OS for resolving dependencies, the Export Table to find functions that can be referenced by other programs, and more;
- *Section Table*: a list of entries that indicates the characteristics of each section of the program, like the code section (*.text*), initialized data (*.data*), relocations (*.reloc*) and more.

After the header, the file contains the executable code, along with other assets, stacked one after the other. It is clear that, even without executing the program contained inside a file, it is possible to infer some useful information from its headers, imports, exports, and sections.

2.2 Malware Classifiers

Here, we describe two recent byte-based convolutional neural network models for malware detection. Both take as input the raw bytes from the Windows PE file on disk, use an embedding layer to project the bytes into a higher-dimensional space, and then apply one or more convolutional layers to learn relevant features that are fed to a fully-connected layer for classification with a sigmoid function. While they share common design concepts, they differ in their overarching architecture and, as we will see, this difference is the key to their respective robustness to the various adversarial evasion attacks described in this paper. In addition to these two deep learning models, we also consider a traditional ML model using gradient boosting decision trees on hand-engineered features, which we use as a baseline for purposes of comparison and to understand transferability of attacks from byte-based models to models with semantically-rich features.

MalConv: Proposed by Raff et al. [20], MalConv is a convolutional neural network model that combines an eight-dimensional, learnable embedding layer with one-dimensional gated convolution. The striding and kernel size of the convolutional layer effectively means that the convolutional layer iterates over non-overlapping windows of 500 bytes each, with a total of 128 convolutional filters. A global max pooling is applied to the gated outputs of the convolutional layer resulting in a set of the 128 largest-activation features from among all convolutional windows without regard to the structure or locality of those features within the binary, which are used as input to a fully-connected layer for classification. While the original MalConv paper specifies a maximum size of 2MB, the implementation used in our experiments was provided by Anderson et al. [3] with a maximum file size of 1MB along with a pre-trained model using the EMBER dataset. Files exceeding the maximum allowable size are truncated, while shorter files are padded using a distinguished padding token separate from the standard bytes in the file (i.e. resulting in 257 unique tokens).

DNN with Linear (DNN-Lin) and ReLU (DNN-ReLU) activations: Jeffrey Johns[7] and Coull et al. [5] proposed a deep convolutional neural network that combines a ten-dimensional, learnable embedding layer with a series of five interleaved convolutional and max pooling layers arranged hierarchically so that the original input size is reduced by one quarter (1/4) at each level. The outputs of the final convolutional layer are globally pooled to create a fixed-length feature vector that is sent to a fully-connected layer for final classification. Since the convolutional layers are hierarchically arranged, locality information among the learned features is preserved and compressed as it flows upwards towards the final classification layer. The maximum length of this model is 100KB to account for the deep architecture, and like the MalConv model files exceeding this length are truncated and shorter files are padded with a distinguished padding token. Several variations of this architecture are evaluated in this paper, including examining performance with both linear and Rectified Linear Unit (ReLU) activations for the convolutional layers, as well as performance when trained using the EMBER dataset and a proprietary dataset containing more than 10x the number of training samples. An analysis of the model by Coull et al. [5] demonstrates how the network attributes importance to meaningful features inside the binary, such as the name of sections, the presence of the checksum, and other structures.

Gradient Boosting Decision Tree (GBDT): A gradient-boosted tree ensemble model trained provided as part of the EMBER open-source dataset by Anderson et al [3]. The model uses a set of 2,381 hand-engineered features derived from static analysis of the binary using the LIEF PE parsing library, including imports, byte-entropy histograms, header properties, and sections, which generally represent the current state of the art in traditional ML-based malware detection. Given its use of a diverse set of semantically-meaningful static features, it provides an excellent baseline

[7] https://www.camlis.org/2017/jeffreyjohns

to compare the two above byte-based models against, and help demonstrate the gap between the features learned by byte-based neural networks and those created by subject-matter experts.

The MalConv architecture has been extensively studied by previous work and a wide variety of adversarial attacks have shown great success against the model [7, 8, 13, 16, 22, 24]. As pointed out by Suciu et al. [24], the lack of robustness in this model may be strongly tied to its weak notions of spatial locality among the learned features – meaning that the location of the injected adversarial noise does not matter as long as the activation on the noise bytes overwhelms those from the actual binary. By contrast, the deep convolutional net proposed by Johns and Coull et al. [5] enforces spatial locality among features, which means both the location and magnitude of adversarial noise play a role in the success of evasion attacks. Even the GBDT model has been shown to be vulnerable to evasion attacks [4, 8, 22], albeit it with more advanced and computationally-intensive attacks. While each of these models has been previously been evaluated in an ad-hoc manner, we are the first to treat attacks on machine-learning models in a holistic manner using a single unifying framework, and in doing so we uncover two new attack methods that apply to both MalConv and the Coull et al.'s model despite the unique architectural differences between the two.

3 ATTACK FRAMEWORK AND NOVEL PRACTICAL MANIPULATIONS

In this section, we introduce our framework for computing adversarial examples, called RAMEn. We begin by presenting the formalism in Section 3.1, then describe the practical manipulations for perturbing the binaries without impacting their functionality in Section 3.2. Later, we show how to express a range of current state of the art attacks in the RAMEn framework in Section 3.3, and present our novel attack strategies in Section 3.4.

3.1 Attack Framework

We are considering here machine-learning models that take a program as input and aim to classify it either as legitimate or malicious. Since code is written as arbitrarily-long strings of bytes, we define the set of all possible functioning programs in the input space as $\mathcal{Z} \subset \{0, \dots, 255\}^*$. Most classifiers process data through a *feature extraction* step and the attack takes place inside such a feature space. We thus denote the feature extraction step with the function $\phi : \mathcal{Z} \to \mathcal{X}$, where $\mathcal{X} \subseteq \mathbb{R}^d$ is the feature space, and the prediction function with $f : \mathcal{X} \to \mathbb{R}$. This function outputs a continuous value representing, without loss of generality, the probability of the input sample belonging to the malware class. This value is then thresholded to obtain a final decision in the label space $\mathcal{Y} = \{-1, +1\}$, being -1 and $+1$ the class labels for legitimate and malicious samples, respectively.

Under this setting, the attacker aims to craft adversarial malware by perturbing each malicious input program to achieve evasion with high confidence. To this end, the attacker is required to apply *practical manipulations*, i.e. , transformations that alter the representation of the input program without disrupting its original behavior, by exploiting the redundancies offered by the executable file format. We encode these functionality-preserving manipulations as a function $h : \mathcal{Z} \times \mathcal{T} \to \mathcal{Z}$, whose output is a functioning program with the same behavior as the input, but with a different representation. It takes in input a program $\boldsymbol{z} \in \mathcal{Z}$ and a vector $\boldsymbol{t} \in \mathcal{T}$, that represents the parameters of the function h. By optimizing this quantity, the attacker tunes the application of the practical manipulations on a particular sample. Thus, we can denote with $\mathcal{H} = \{h(\boldsymbol{z}, \boldsymbol{t}), \boldsymbol{t} \in \mathcal{T}\}$ the set of programs that can be created by applying practical manipulations on $\boldsymbol{z}$, the original sample. We then use a *loss function* $L : \mathbb{R} \times \mathcal{Y} \to \mathbb{R}$ to measure how likely an input sample is classified as malware, by comparing the output of the prediction $f(\phi(\boldsymbol{z}))$ on a malicious input sample $\boldsymbol{z}$ against the class label $y = -1$ of benign samples.

We are now in the position to present **RAMEn**, a general framework that reduces the problem of computing adversarial malware examples to optimization problems of the form:

$$\min_{\boldsymbol{t} \in \mathcal{T}} \quad L(f(\phi(h(\boldsymbol{z}, \boldsymbol{t})), y)\,. \tag{1}$$

In fact, by minimizing the aforementioned loss function, the attacker aims to reduce the probability of the modified program being recognized as malware, i.e. , increases the probability of evasion, while retaining malicious functionality.

Depending on the differentiability of the components of RAMEn, the attacker can use different strategies for solving the optimization problem, as discussed in the following.

1. Every function is differentiable: If all functions are differentiable, then the optimization can be carried out inside the space of the parameters $\mathcal{T}$ via gradient-descent algorithms, i.e. , iteratively updating the transformation parameters as:

$$\boldsymbol{t}_{(i+1)} = \boldsymbol{t}_{(i)} + \gamma \frac{\partial L}{}, \partial \boldsymbol{t} \tag{2}$$

where γ is the step size of the gradient descent algorithm. A use-case for Equation 2 is the production of adversarial examples against end-to-end image detectors where each pixel is given as input. In this case, the target model is end-to-end differentiable and the practical manipulations consist of differentiable functions, e.g. , rotations or color-space transformations.

2. Non-differentiable manipulations: If all the functions but h are differentiable, then the attacker can leverage a gradient descent technique for creating the next adversarial examples and extract the best vector of coefficients by reconstruction:

$$\boldsymbol{z}' = \boldsymbol{z} + \gamma \frac{\partial L}{\partial h}\,, \tag{3}$$

$$\boldsymbol{t}_{(i+1)} = \arg\min_{\boldsymbol{t} \in \mathcal{T}} \parallel \boldsymbol{z}' - h(\boldsymbol{z}, \boldsymbol{t}) \parallel^2 \;\;. \tag{4}$$

3. Non-differentiable feature extraction: Feature extractors are often non-differentiable and/or non-invertible. When this is the case the attacker must carry out the optimization inside the feature space by solving a minimization problem as shown in Figure 2. Let $\mathcal{H} : \{h(z_0, \boldsymbol{t}), \boldsymbol{t} \in \mathcal{T}\}$ be the set of all possible programs generated by $\boldsymbol{z}$ using practical manipulations, and let $\mathcal{V}$ be a super set of all possible feature vectors that satisfies the constraints posed by the practical manipulations, since they might not be expressed in a closed way. The problem inside the feature space can be posed as:

$$\boldsymbol{x}^\star \in \arg\min_{\boldsymbol{x} \in \mathcal{V}} L(f(\boldsymbol{x}), y) \tag{5}$$

whose solutions are feature vectors that satisfy the constraints inside the feature space. Now, the attacker needs to find the best vector of parameters $\boldsymbol{t}^\star$ that, when applied to $\boldsymbol{z}$, generates the closest sample inside the feature space to $\boldsymbol{x}^\star$. This can be expressed as a reconstruction problem in the form of a minimization:

$$\boldsymbol{t}^\star \in \arg\min_{\boldsymbol{t} \in \mathcal{T}} \|\boldsymbol{x}^\star - \phi(h(\boldsymbol{z}, \boldsymbol{t}))\|^2 \tag{6}$$

Since a one-to-one mapping between $\mathcal{H}$ and $\mathcal{V}$ may not exist, there could be feature vectors that do not correspond to any program inside $\mathcal{H}$. For this reason, the attacker can compute the closest point to the desirable one inside $\mathcal{H}$ by taking the program that minimizes the reconstruction error.

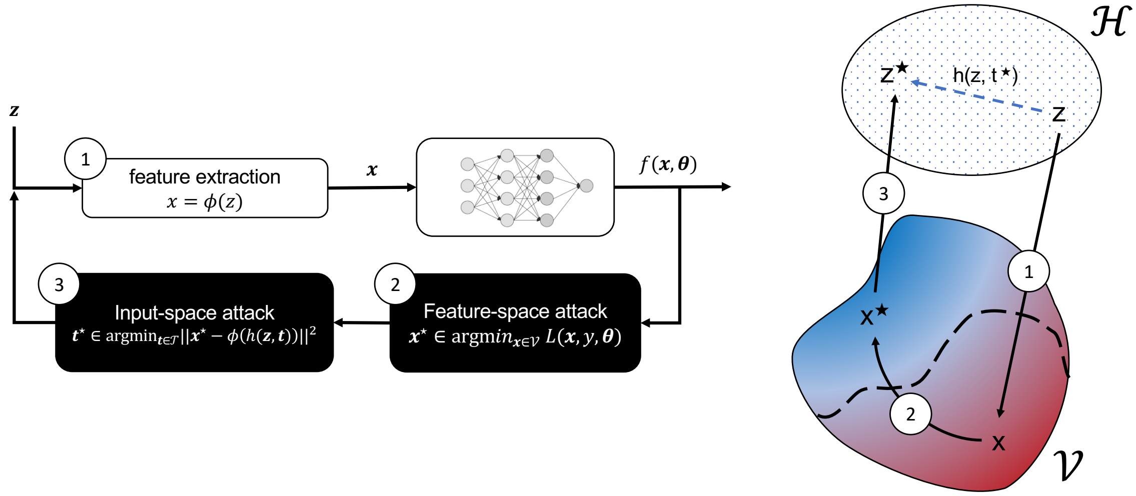

$\boldsymbol{z}$
1
feature extraction
$x = \phi(z)$
$\boldsymbol{x}$
$f(\boldsymbol{x}, \boldsymbol{\theta})$
3
Input-space attack
$\boldsymbol{t}^\star \in \text{argmin}_{t \in \mathcal{T}} ||\boldsymbol{x}^\star - \phi(h(\boldsymbol{z}, \boldsymbol{t}))||^2$
2
Feature-space attack
$\boldsymbol{x}^\star \in \text{argmin}_{x \in \mathcal{V}} L(\boldsymbol{x}, y, \boldsymbol{\theta})$
$\mathcal{H}$
z★
h(z, t★)
z
3
1
x★
2
x
$\mathcal{V}$

Fig. 2. The steps performed by RAMEn to produce adversarial examples.

4. Non-differentiable model: If all the functions are non-differentiable including the model itself (e.g. random forest classifiers), then no gradient information is available, and therefore the attacker can only consider the use of black-box optimization methods.

The use of non-differentiable feature extractors is a recurring pattern in security classifiers, both in traditional models and end-to-end deep learning models. For instance, when truncating and embedding the input bytes, the byte-based deep learning models operate inside a non-differentiable feature extraction regime. In particular, the embedding can be reversed only by using a look-up function that associates each embedding value to the corresponding value inside the input space: this function is clearly non-differentiable. For these reasons, we propose Algorithm 1 for solving Equation 1. By specifying different strategies for solving the two minimization problems, the attacker can customize RAMEn to their needs.

Algorithm 1: General version of RAMEn algorithm

```
Data: malware $z$, iterations $N$, target class label $y$
Result: $\boldsymbol{t}^\star, \boldsymbol{z}^\star$
$\boldsymbol{t}^{(0)} \in \mathcal{T}$
for $i$ in $[0, N-1]$
    $\boldsymbol{x} = \phi(h(z, \boldsymbol{t}^{(i)}))$
    $\boldsymbol{x}^\star = \arg\min_{\boldsymbol{x}' \in \mathcal{V}} L(f(\boldsymbol{x}'), y)$ # initialize $\boldsymbol{x}'$ with $\boldsymbol{x}$ at beginning of optimization
    $\boldsymbol{t}^{(i+1)} = \arg\min_{\boldsymbol{t} \in \mathcal{T}} \parallel \boldsymbol{x}^\star - \phi(h(z, \boldsymbol{t})) \parallel^2$
$\boldsymbol{t}^\star = \boldsymbol{t}^{(N)}$
$z^\star = h(z, \boldsymbol{t}^\star)$
return $\boldsymbol{t}^\star, z^\star$
```

Implementation of RAMEn: Here, we extend the algorithm proposed Demetrio et al. [7] and Kolosnjaji et al. [13], and we encode this strategy inside RAMEn. The attack addresses locations inside the input program that are not considered at run-time, like the DOS header or padding bytes. By optimizing bytes inside these regions, we obtain functioning adversarial malware that evades detection. The first layer of our target networks is an embedding layer, applied to impose a learnable metric over the input bytes and therefore acting as a feature extractor. These networks are differentiable up to this layer, while no derivative can be computed w.r.t. the real bytes of the input programs. The networks have a fixed input size: all the programs longer than a specific amount will be truncated before passing to the embedding layer. Hence, we can encode ϕ as the composition of the truncation and embedding functions of these networks. Since the strategy we use does not need a loss function, but it is required inside RAMEn, we use the probability score assigned by the network itself as a proxy: $L(f(\phi(z)), y) = f(\phi(z))$. Algorithm 2 implements the resolution of both problems stated in Equation 5 and 6, by using a slightly modified version of the solver proposed by Kolosnjaji et al. [13]. First, the optimizer computes the gradient w.r.t. the input inside the embedding space (line 5). The minus sign is applied since we are looking for the direction of the benign class, associated with low values of the model function. As a consequence of the embedding, the computed gradient is a matrix, and each entry is a vector inside the embedding space. The algorithm ignores all the locations that cannot be modified by applying a binary mask $\boldsymbol{m}$ on the entries of the gradient (line 5), and it takes the indexes of the first *gamma* non-zero sorted entries (line 6). The γ parameter is a step-size constant that controls how many bytes are perturbed at each round, modulating how much the space is explored during the optimization. For each byte of the payload, the algorithm computes a line passing from the current value to be replaced, and whose direction is imposed by the gradient in that point. The algorithm proceeds by projecting all the 256 embedded byte values on this direction, computing the distance point-to-line and the alignment with such direction (line 9). The best byte replacement value is the one with the smallest non-zero distance, with positive alignment to the defined direction (line 10). We denote $\hat{\phi}$ the function that encodes a single byte inside the feature space (used in line 1). In this case, $\mathcal{V}$ and $\mathcal{H}$ have a one-to-one match, since the only allowed values are the ones computed by embedding.

Algorithm 2: Implementation of RAMEn used in this work, optimizing and reconstructing single bytes inside the input program.

```
   Data: malware $z$, number of bytes to optimise $\gamma$, iterations $N$
   Result: $\boldsymbol{t}^\star, z^\star$
1  $\mathrm{E}_i = \hat{\phi}(i), \forall i \in [0, 256]$
2  $\boldsymbol{t}^{(0)} \in \mathcal{T}$
3  for $i$ in $[0, N-1]$
4      $\mathrm{X} = \phi(h(z, \boldsymbol{t}^{(i)}))$
5      $\mathrm{G} = -\nabla_{\mathrm{X}} f(\mathrm{X}) \odot \boldsymbol{m}$
6      for $k$ in argsort $(\| G \|)_{0,\ldots,\gamma} \wedge \mathrm{G}_k \neq \mathbf{0}$
7          for $j$ in $[0, \ldots, 255]$
8              $\mathrm{S}_{k,j} = \mathbf{G}_k^t \cdot (\mathbf{E}_j - \mathbf{X}_k)$
9              $\widetilde{\mathbf{X}}_{k,j} = \| \mathbf{E}_j - (\mathbf{X}_k + \mathbf{G}_k \mathrm{S}_{k,j}) \|_2$
10         $\boldsymbol{t}_k^{(i+1)} = \arg\min_{j:\mathrm{S}_{k,j}>0} \widetilde{\mathbf{X}}_{k,j}$
11 $\boldsymbol{t}^\star = t^{(N)}$
12 $z^\star = h(z, \boldsymbol{t}^\star)$
13 return $\boldsymbol{t}^\star, z^\star$
```

3.2 Practical Manipulations

Since the attacker wants to camouflage the malicious content without compromising its functionality, they need an effective way of perturbing the representation on disk. As explained in Section 2.1, the OS loads the executable by following a specific algorithm to map its sections into memory with the correct permissions. To do so, the Windows loader must find the components of the program inside the executable on disk. Since the attacker want to produce space for injecting the adversarial payload, they can alter the representation to their advantage by shifting content within the bounds of the specification. This is not the first work exploring the space of practical manipulations applicable to the Windows PE format [2, 4, 7, 8, 13, 16, 22–24], and we will focus on manipulations that have not yet been proposed. In particular, these transformations can be used to create space inside the input binary, and hide all the malicious code from the target network, thereby making them more difficult to discover and remove, while also increasing the number of adversarial bytes that can be injected. The payload is then optimized using the implementation of RAMEn expressed in Algorithm 2 by specifying inside h how the content is shifted or extended, and providing the new locations inside the $\boldsymbol{m}$ mask.

DOS Header editing: The DOS header is kept for compatibility with older OSes. Since the only two important fields are the magic number `MZ` and the 4 byte-long integer at offset `0x3c`, all the other bytes can be used by the attacker as a space for storing the adversarial payload. Demetrio er al. [7] originally proposed a mutation applied between the magic number and the real header offset, but the whole DOS header can be manipulated without damaging the file structure. To encode this manipulation inside RAMEn, we simply add more indexes to the mask $\boldsymbol{m}$ by looking up where the `PE` magic number is located, this includes bytes in the range [`0x02`, `0x3c`) and the range [`0x40`, PE location). The position of the PE header may vary from binary to binary, but the offset inside the file is expressed inside the header.

DOS Header Extension: The DOS header contains a pointer to the PE header of the program, and the bytes in-between are used only by the loader of a deprecated DOS system. The attacker can abuse this pointer by substituting the original value with a higher one. This will ensure that the Windows loader will try to parse the PE header in another location of the file. To keep the structure intact, the attacker must also shift all the content of the file to match the introduced gap, keeping the alignment as expressed in the PE header. This mutation has no effect at run-time since the introduced content is not used by the program. The attacker has now more space at their disposal to use during the optimization of the adversarial payload.

Content Shifting: Each section is retrieved by the Windows loader using the *physical offset*, that is a four-byte unsigned integer specified inside each *section entry*. Each offset is aligned to a value specified inside the header of the program: if these values do not match, the executable will not be executed, and it is marked as corrupted. While loading the program, the OS does not perform any check on the order of the sections inside the file, and it could potentially map in memory the last section before the first one. The order is imposed only in memory, and the attacker can edit the physical offsets of each section by shuffling the content of the file, with the only constraint imposed by the section alignment. The shift allows the attacker to carve additional space that otherwise would not have been available for their adversarial payload.

A simplified graphical representation of these strategies is depicted in Figure 3. The colored areas highlight where the attacker are placing the adversarial payload, and the length of the boxes indicates how the content is being shifted by the applied noise.

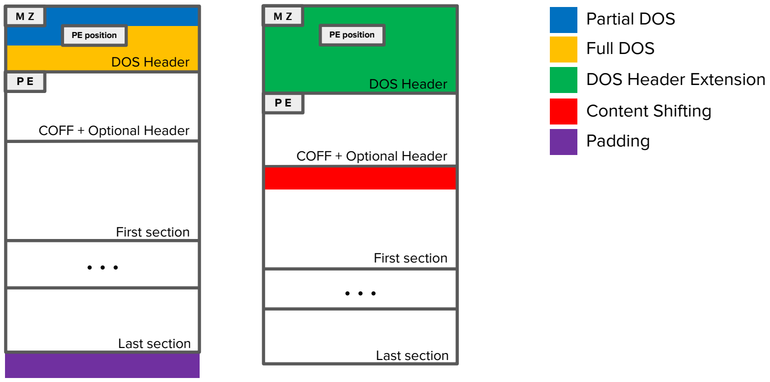


Fig. 3. Graphical representation of the location perturbed by the different strategies. Colours highlight the adversarial payload injection.

3.3 Implementing Existing White-box Attacks within RAMEn

Table 1 contains all the specifications used for implementing the techniques proposed in the previous literature within the RAMEn framework. Every strategy must define two blocks: how they use the gradient to move inside the space and how they reconstruct the sample in the original input space. Both Kolosnjaji et al. [13] and Demetrio et al. [7] share the methodology, since they use the same algorithm, but applied in a different location. The formalization for these two strategies is the same as the approach we introduced in Algorithm 2, but instead of taking only the most influential γ entries, we run the optimization on the all non-zero entries taken into account for the attack. Kolosnjaji et al. [13] append content at the end of the input program, while Demetrio et al. [7] fill a portion of the DOS header of the input program. Both strategies solve the problem in Equation 5 by searching the closest embedding value parallel to the gradient (*closest positive* in Table 1). This is done for each value that needs to be modified. The problem in Equation 6 is solved by inverting the look-up operation performed by the embedding layer (*inverse look-up* in Table 1). This operation can not fail, since the search at the step before chooses only values that correspond to bytes inside the input space.

Kreuk et al. [16] and Suciu et al. [24] apply the fast gradient sign method (FGSM) [9] inside slack space, that is the unused space between sections, and within appended padding. Then, at the end of all the iterations of the algorithm, they project each embedded byte inside the original space. To do so, for each location they take the corresponding closer byte inside the embedding space (*closest* in Table 1).

Sharif et al. [22] apply random manipulations that change instructions inside the *.text* section with semantics-equivalent ones, or they displace the code inside another section, with the use of *jump* instructions. At each iteration of the algorithm, the latest adversarial example is used as a starting point for the new one. Once randomly perturbed, the new and the old versions are projected inside the embedding space, where the two points are used for computing a direction. If this direction is parallel to the gradient of the function in that point, the sample is kept for the next iteration, otherwise it is discarded. In this case, the strategy does not optimize the sample inside the feature space, since

Attack	Proposed in	Loss function	practical manipulations	feat. space optimization	reconstruction
Extend	This work	malware score	extend DOS header	closest positive	inverse look-up
Shift	This work	malware score	shift section content	closest positive	inverse look-up
Padding	Kolosnjaji et al. [13]	malware score	padding	closest positive	inverse look-up
Partial DOS	Demetrio et al. [7]	malware score	partial DOS header	closest positive	inverse look-up
FGSM (iterative)	Kreuk et al. [16]	classification loss	padding + slack space	FGSM	closest
FGSM (1 iteration)	Suciu et al. [24]	classification loss	padding + slack space	FGSM	closest
Equivalent instructions	Sharif et al. [22]	C&W loss	equivalent instructions	random	aligned mutation

Table 1. Implementing the white-box attacks of the state of the art , using RAMEn. Each column explains a particular aspect of the strategy. The feature space optimization refers to how the attack deals with Equation 5, and Equation 6 for reconstruction.

each sample is constructed by applying random transformation. Except for the manipulations proposed by Sharif et al. [22], all the strategies describe so far can be found in the the SecML library released alongside this paper.[8]

3.4 Implementing Novel Attacks within RAMEn: Extend and Shift

Here, we formalize in the form of the pseudo-code both of our novel practical manipulations, described in Section 3.2. Algorithm 3 shows how to enlarge the DOS header to add the adversarial payload, while Algorithm 4 shows how to incorporate the payload before the first section. For both algorithms, we need to insert a string of bytes in the desired position, and we fix all the other constraints such as the position of the `PE` magic number offset and the offset of the content of each section. The length of the inserted string is equal to a multiple of the file alignment. The function $\lceil \cdot \rceil$ expresses the *ceil* function, that rounds the input number to the next integer. With a little abuse of notation, we write `"0x00"*`*align* implying that the single character is concatenated *align* times. The first entry of the vector of parameters specifies how many bytes the attacker would like to insert inside the sample. In our experimental analysis, we set the file alignment to 512 bytes for the *Extend* attack, and 1024 for the *Shift* attack. The vector of parameters is padded with zeros if it is shorter that the computed alignment, adapting itself to the desired length. The Extend technique avoids the re-writing of meaningful locations, such as the `MZ` and the `PE` magic numbers, along with the four-byte offset that points to the COFF header.

3.5 Black-box Attacks with Practical Manipulations

We investigate also the application of our practical manipulation in black-box settings, where the attacker does not have knowledge of the model parameters. We describe the strategies we have implemented for testing this scenario, focusing on transfer attacks and query attacks.

Transfer Attacks: The attacker can compute adversarial examples on a model they own, that acts as a surrogate of the target, and they can try to evade detection by submitting such samples to the victim. The surrogate model can be an approximation of the unavailable one, or it can be a common classifier trained for solving the same task. The details of how to obtain such a surrogate model are beyond the scope of this paper, but suffice it to say that there are several methods for accomplishing this through model stealing, using open-source models, or simply training a new model on an open-source dataset (e.g. EMBER). We focus on the latter, by optimizing attacks on the networks we consider for this work, and transferring them against all the others.

Query Attacks: The attacker can also target the victim system by exploiting access only to the prediction step on the target and without optimizing the attack on a surrogate model. By sending queries to the target model, the attacker can

[8] https://github.com/zangobot/secml_malware

Algorithm 3: Implementation of $h(z, \boldsymbol{t})$ for the Extend practical manipulation

```
Data: malware $\boldsymbol{z}$, vector of parameters $\boldsymbol{t}$
Result: $z'$
$align = \lceil \boldsymbol{t}_0 / z.\text{file\_alignment} \rceil * \boldsymbol{z}.\text{file\_alignment}$
if $|\boldsymbol{t}| < align$
    $\boldsymbol{t} = \boldsymbol{t} + \texttt{"0x00"} * (align - |\boldsymbol{t}|)$
$off = \boldsymbol{z}.\text{pe\_offset}$
$\boldsymbol{z'} = z_{0,\dots,off-1} + \texttt{"0x00"} * align + z_{off,\dots,|z|-1}$
$\boldsymbol{z'}.\text{pe\_offset} = off + align$
$S = \boldsymbol{z'}.\text{get\_sections()}$
for $s$ in $S$ do
    $s.\text{physical\_offset} = align + s.\text{physical\_offset}$
end
$\boldsymbol{z'}_{2,\dots,59} = \boldsymbol{t}_{0,\dots,57}$
$\boldsymbol{z'}_{64,\dots,|\boldsymbol{t}|-6} = \boldsymbol{t}_{58,\dots,|\boldsymbol{t}|-1}$
return $\boldsymbol{z'}$
```

Algorithm 4: Implementation of $h(z, \boldsymbol{t})$ for the Shift practical manipulation

```
Data: malware $z$, vector of parameters $\boldsymbol{t}$
Result: $z'$
$align = \lceil \boldsymbol{t}_0 / z.\text{file\_alignment} \rceil * z.\text{file\_alignment}$
if $|\boldsymbol{t}| < align$
    $\boldsymbol{t} = \boldsymbol{t} + \texttt{"0x00"} * (align - |\boldsymbol{t}|)$
$fs = z.\text{get\_first\_section\_offset()}$
$z' = z_{0,\dots,fs-1} + \texttt{"0x00"} * align + z_{fs,\dots,|z|-1}$
$z'.\text{pe\_offset} = z'.\text{pe\_offset} + align$
$S = \boldsymbol{z'}.\text{get\_sections()}$
for $s$ in $S$ do
    $s.\text{physical\_offset} = align + s.\text{physical\_offset}$
end
$\boldsymbol{z'}_{fs,\dots,fs+(align-1)} = \boldsymbol{t}$
return $\boldsymbol{z'}$
```

infer how the victim behaves locally around a particular set of samples they want to be misclassified. We implement RAMEn with the use of a genetic black-box optimizer, the same used by Demetrio et al. [8] for computing their attack. Since the genetic black-box optimizer works with real numbers, we encoded our vector of parameters as $\boldsymbol{t} \in [0, 1]^k$, where k is the number of values that will be perturbed. For instance, the *Partial DOS* attack sets k to 58. Before applying the practical manipulation h, we need to multiply by 255 and rounding to the nearest value the vector of parameter, since both Algorithm 3 and 4 consider each entry of vectors $\boldsymbol{t}$ as bytes to be placed inside the sample. We summarize the optimizer in Algorithm 5, where we have plugged the loss to minimize as dictated by RAMEn. The $\lceil \cdot \rfloor$ function rounds the argument's entries to the nearest integer. The pseudo-code follows the generic structure of genetic algorithms, where the initial N points are randomly generated. This population is modified by inserting new elements and keeping only the fittest, i.e. the one closest to the benign class, denoted by y. Both the *crossover* and the *mutate* functions apply perturbations on the population, by mixing and replacing entries of samples of this set.

Algorithm 5: Pseudo-code of the genetic black-box optimizer, adapted from previous work [8]

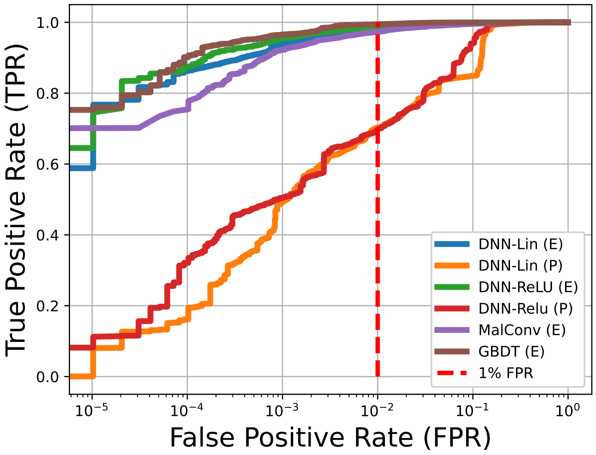
```
Data: population size $m$, generations $N$
Result: $\boldsymbol{t}^\star, z^\star$
P = $m$ random points
F = $(\mathbf{P}_i, L(f(\phi(h(z, \lceil 255\mathbf{P}_i \rfloor))), \mathrm{y}))_{i=1}^{m}$
i = 0
S = F
while $i < N$ do
    S = selection(S)
    S = crossover(S)
    S = mutate(S)
    S = $F \cup \bigcup_{\boldsymbol{t} \in S} (\boldsymbol{t}, L(f(\phi(h(z, \lceil 255\boldsymbol{t} \rfloor))), y))$
    i = i+1
end
$\boldsymbol{t}^\star = min(S)$
$z^\star = h(z, \boldsymbol{t}^\star)$
return $\boldsymbol{t}^\star, z^\star$
```

The only strategy that we omit from this black-box analysis is the FGSM proposed by Kreuk et al. [16] and Suciu et al. [24], since it is similar to the Padding strategy. We test Gamma [8], a regularized black-box strategy whose practical manipulations consist in appending content harvested from benign software to elude detection. Each entry of the vector of manipulations $\boldsymbol{t}_i$ specifies the amount of content that need to be taken from the chosen a-priori $i-$th section. After having built the adversarial payload, it is appended at the end of the input sample.

4 EXPERIMENTAL ANALYSIS

To perform our experiments, we used a Ubuntu 16.04.3 LTS server, with an Intel® Xeon® E5-2630 CPU, with 64 GB of RAM. We also used a Windows 10 virtual machine during the development of the practical manipulations described in Section 3.2 to validate that they indeed do not impact functionality. To highlight the performance of our strategy, we encoded other attacks proposed in the state of the art [16, 24] and we ran them against the chosen targets. We report all the implementation details of these techniques in Section 5. The network proposed by Johns[9] and Coull et al. [5] has been trained with two different datasets. The first one is EMBER [3], which is an open-source dataset of goodware and malware hashes, including a set of pre-extracted features, while the second is a proprietary production-quality dataset used for training malware classifiers. The first dataset is smaller, counting 1.1 M samples, while the second is larger, counting 16.3M files. MalConv has been trained on EMBER [3], like the GBDT model proposed by Anderson et al. [3].

[9] https://www.camlis.org/2017/jeffreyjohns



(a)

DNN-Lin (E)	0.5192
DNN-Lin (P)	0.5863
DNN-ReLU (E)	0.2357
DNN-ReLU (P)	0.5764
MalConv (E)	0.1955
GBDT (E)	0.0057

(b)

Fig. 4. On the left, the Receiver Operating Characteristic curve (ROC) of the classifiers under analysis, evaluated on the Ember test set [3]. The letter inside the parenthesis specifies the dataset used for training the classifier: *E* means Ember, while *P* implied the use of a larger proprietary dataset. The red dashed line highlights the performance of each classifier at 1% False Positive Rate. On the right, the detection thresholds of the classifiers, at 1% False Positive Rate.

The malware set we used for the empirical evaluation are the same as used by Demetrio et al. [7]. All the strategies are available online as an extension of SecML framework [18], named SecML Malware.[10]

Malware Detection Performance: Before delving inside the performance of the different attacks against the target classifiers, we first compute the Receiver Operating Characteristic (ROC) of the four models. The score has been computed on the test set of EMBER v1 dataset [3].

It is clear from Figure 4a that the GBDT model outmatches the convolutional networks, which aligns with previously reported results on this dataset [3]. This benefit might be connected to the manual feature engineering that is used by the GBDT model, instead of letting the network learn the relevant features itself. The non-linearity imposed by ReLU activation functions does not seem impact the overall score, implying that the majority of the examples in the dataset can be linearly separated. However, we do note that at lower false positive rates, the performance of the ReLU models does exceed that of the linear activation models, which may support the notion that there are some samples that are difficult to separate and where the non-linearity is useful. MalConv shows comparable though somewhat lagging results to both the DNN models trained on EMBER and the GBDT model. Clearly, the use of a larger and more diverse proprietary dataset does not necessarily improve the generalizability of the end-to-end models, which agrees with previous observations by Coull et al.[5] showing that overfitting may actually be beneficial in malware classification tasks. For each classifier, we compute the threshold such that the classifier has a 1% False Positive Rate (FPR). We use these thresholds to compute the Detection Rate (DR) for each attack in our experimental analysis. We report in Table 4b all the thresholds computed from using the results of the ROC.

[10] https://github.com/zangobot/secml_malware

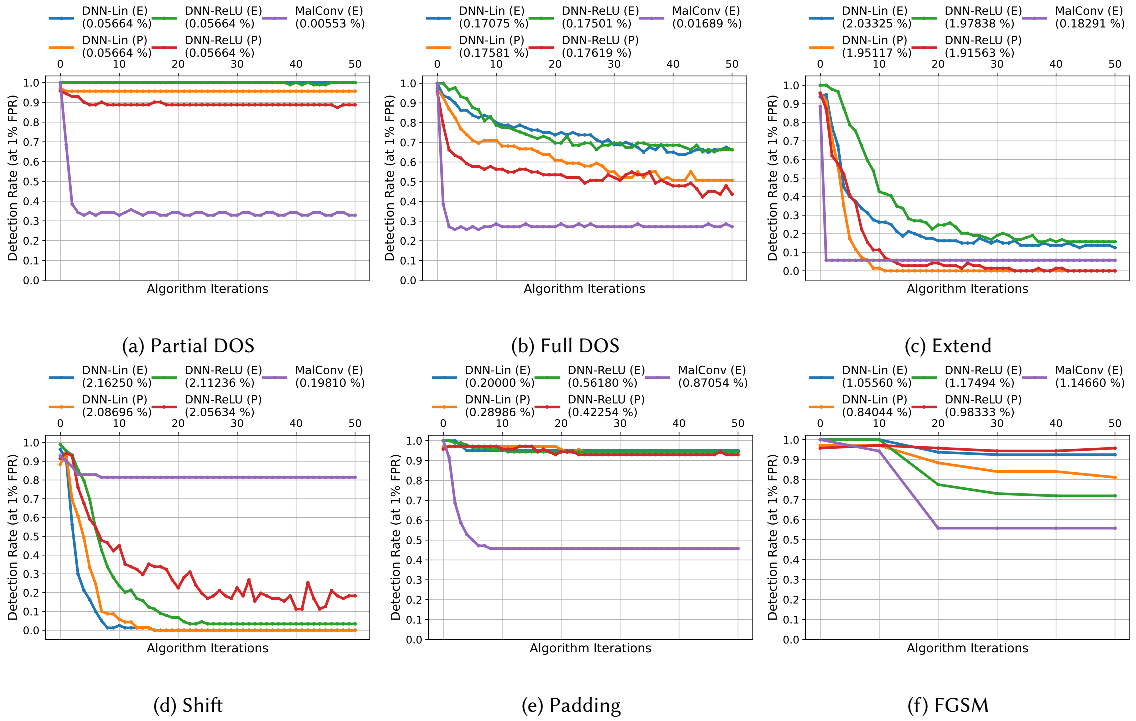
4.1 White-box Attacks Results

We tested all the differentiable models with the attacks formulated in Section 3.2 (*Full DOS*, *Extend* and *Shift*), the header attack [7], the padding attack [13], an iterative implementation of the fast gradient sign method (FGSM) that address both padding and slack space [16, 24]. The *Partial DOS* attack proposed by Demetrio et al. [7] alters only the first 58 bytes contained in the DOS header, ignoring the first two bytes containing the magic number MZ and the four-bytes-long offset located between 0x3c and 0x3f. The *Full DOS* attack searches for the PE signature inside the sample, and it marks as editable all the bytes in between except the one discussed in *Partial DOS*. This amount may vary from sample to sample: in our test dataset, it varies from 118 to 290 bytes. As already mentioned in Section 3.4, the *Extend* attack has a minimum shift of 512 rounded to the nearest multiple of the file alignment specified by the sample: in our test set, it varies between 512 and 4096 bytes, resulting in a payloads whose length varies between 630 and 4386. Similarly, the *Shift* attack adds 1024 bytes before the first section of the sample and aligns to the nearest multiple of the specified file alignment, resulting in adversarial payloads with a length between 1024 and 4096 bytes. The *Padding* attack appends bytes at the end of the file, with a default payload size of 10240 KB, motivated by the results obtained by Kolosnjaji et al. [13]. The iterative implementation of FGSM considers both padding and unused space between sections (slack space), and we set the FGSM free-parameter ϵ to 0.1. For all the attacks, we have chosen a step size of 256 bytes optimized at each iteration. We show the performance of each white-box technique we have described, targeting the different models we have considered for this analysis.

Discussions of the results: While the *Partial DOS* technique is generally ineffective against all classifiers except for MalConv, the *Full DOS* attack does substantially lower the detection rate of the networks proposed by Coull et al. [5]. This might be caused by spurious correlations learnt by the network, and altering these values cause the classifier to lose precision.

Both our novel strategies, *Extend* and *Shift* show great attack performance against all evaluated networks. Since these networks use different convolutional layers to learn local spatial information (especially DNN-Lin and DNN-ReLU [5]), it is possible that these models recognize adjacent characters and two-or-three byte instructions, and more. Our novel strategies replace a portion of the real header of the program, and it might be possible that the adversarial payload interferes with these local patterns learned by the networks. The *Extend* attack, for instance, covers the original position of the PE offset plus many fields of the Optional Header, like the *checksum* and the locations of directories such as the Import Table and Export table. This content is preserved, since it is shifted, but it is no longer present in the position the network believed them to be. The *Shift* attack does the same, but with the content of the first section, that is usually the one containing the code of the program. Surprisingly, the *Shift* attack against MalConv is not as effective as it was against the other networks. The reason might be once again the wrong feature importance that MalConv attributes to certain bytes. Analyzing the norm of the gradient computed on the location altered by the attack, we found that it is mostly zero, and the attack is unable to optimize the payload. If the attention is focused on the header, the rest of the file has a low impact on the final score. This can be glimpsed by looking at the *Extend* attack, which manipulates an extended portion of bytes starting from the DOS header.

The *Padding* attack proposed by Kolosnjaji et al. [13], and the *FGSM* attack proposed by Kreuk et al. [16] and Suciu et al. [24] fail to achieve evasion, since most of the manipulations applied are cut off by the limited window size of the network itself. For instance, if a sample is larger than 100 KB, it can not be padded, and all the strategies that rely on padding fail. To achieve evasion, these attacks can only leverage the perturbation of the slack space, but the number of bytes that can be safely manipulated is too few to have significant impact. Also, this strategy is incapacitated by



(a) Partial DOS

(b) Full DOS

(c) Extend

(d) Shift

(e) Padding

(f) FGSM

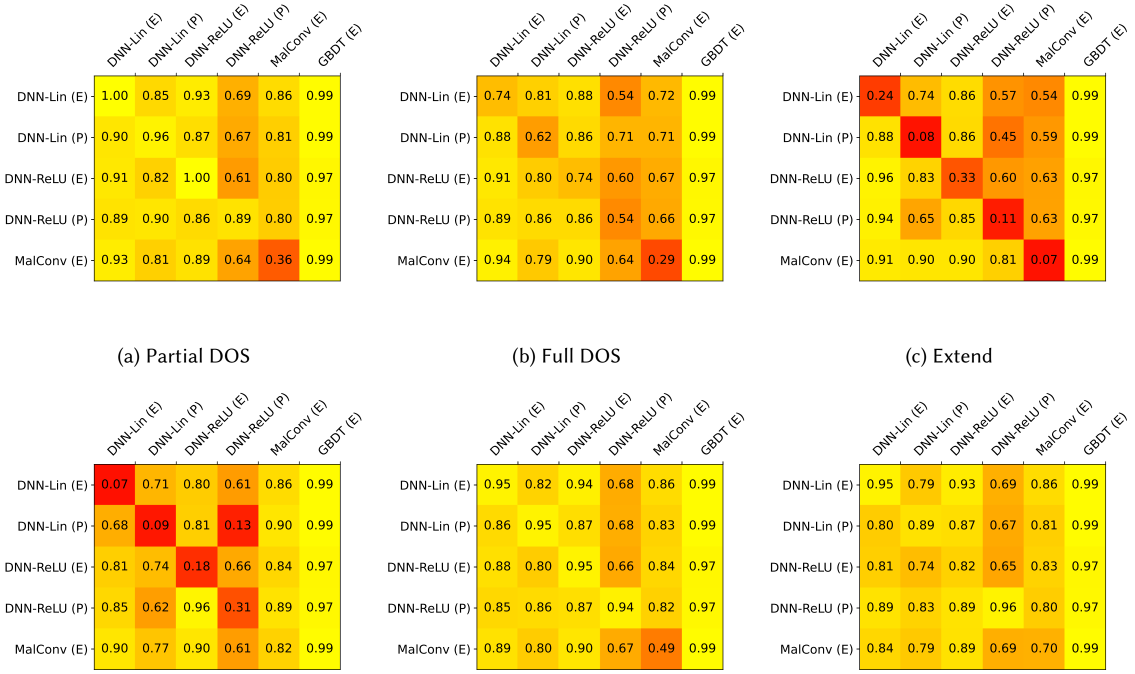
Fig. 5. The results of white-box attacks, expressed as the mean Detection Rate at each optimization step. Each plot sums up the degradation induced by a specific strategy against the classifiers we have considered, trained on different datasets (*E* for EMBER and *P* for proprietary). The number near the name of the classifier represents the size of the adversarial payload as a percentage of the input size.

the inverse-mapping problem: they compute the adversarial examples inside the feature space, and they project them back only at the end of the algorithm. This means that the attack might be successful inside the feature space, but not inside the input space, where there are a lot of constraints that are ignored by the attack itself. Against MalConv, the *Padding* attack proves to be quite effective, but it needs at most 10 KB to land successful attacks, as already highlighted by Kolosnjaji et al. [13]. The adversarial payload must include as many bytes as possible to counterbalance the high score carried by the ones contained inside the header.

Speaking of the the size of the adversarial payload of our novel strategies, we report the mean percentage size of the crafted noise w.r.t. to the input window of the target network near every label of the legend of Figure 5. This network has a window size of `100 KB`, and each attack alter, on average, 2% of that quantity (approximately, `3 KB`). Also, we can observe that both DNN-Lin and DNN-ReLU trained on the larger dataset are less robust w.r.t. to their counterparts trained on Ember, and this pattern can be observed in almost every white-box attack we have showed.

4.2 Black-box Transfer Attacks

We show how the classifiers under analysis behave against black-box transfer attacks. The latter is crucial since an attacker might optimize attacks against a model they own, and then they can try to evade other systems in the wild. To this extent, we use the adversarial EXEmples crafted for the white-box attacks, and we test them against all the



(a) Partial DOS

(b) Full DOS

(c) Extend

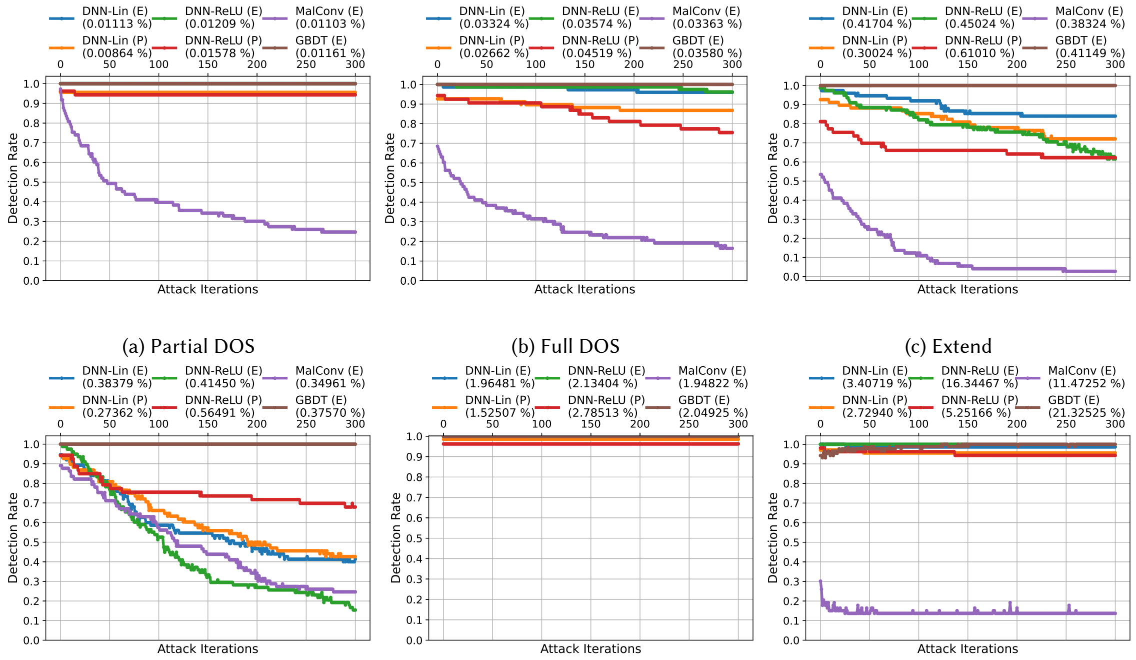
(d) Shift

(e) Padding

(f) FGSM

Fig. 6. Comparisons of transfer attacks, expressed using the detection rate: on the x-axis, the model used for computing the adversarial EXEmples. On the y-axis, the model used for testing the transfer. The diagonal shows the results of the white-box attack. For computing the Detection Rate, we used the threshold with 1% False Positive Rate.

other models. In general, these transfer attacks are not effective, as clearly highlighted by Figure 6, but still they pose interesting results for our analysis. Optimizing attacks on DNN-Lin and DNN-ReLU has an impact on the performance of MalConv, especially the *Extend* and the *Full DOS* attacks, while the contrary is not. Also, the DNN-ReLU model seems the model that suffer most from transfer attacks, in terms of shifted confidence. These attacks are not sufficient to subvert detection, but they highlight an interesting trends between models. This result might be explained by the non-linearity imposed by the ReLU activation functions, leading to many different local minima or maxima that are exploited by transfer samples. This effect is less evident with the DNN-Lin models. The GBDT model is not affected by any adversarial transfer attack. The byte-based features used by the decision-tree algorithm are only a small subset of all the characteristics considered by the classifier, such as the API imports, metadata and more. These attacks are not directly optimized against it, and the quantity of bytes that are altered is very little compared to the whole file size. For sure, the adversarial payload has a minimal effect on the byte-based features, but not enough to counterbalance all the other ones.



(a) Partial DOS

(b) Full DOS

(c) Extend

(d) Shift

(e) Padding

(f) Gamma

Fig. 7. The black-box attacks against all the models. We report the Detection Rate at each step of the black-box optimizer. For computing the Detection Rate, we used the threshold with 1% False Positive Rate. The number near the name of the classifier represents the size of the adversarial payload w.r.t. the mean file size of out malware test set.

4.3 Black-box query attacks

The query attacks behave worse than their white-box counterparts, as shown in Figure 7. We once more use the mean Detection Rate as metric for this analysis. We set the population size $N = 10$, and the number of generations $T = 300$. First, the *Extend* attack seems to have lost some of its potential. Recalling the strategy expressed in Section 3.5, the black-box optimizer use a genetic algorithm for finding the best bytes for lowering the confidence, and space is explored using local manipulations. An higher number of manipulations might lead to more exploration of such space, as seems to happen with the *Shift* attack. It is interesting to see that, since these mutations origin from random perturbations, the DNN-Lin and DNN-ReLU trained on the proprietary dataset show more robustness w.r.t. their counterparts, opposed to the white-box results. This might be explained by robustness to random noise induced by the high volume of data used for at training time. MalConv acts as a lower-bound for every other classifier of this experimental analysis, as its scores are successfully lowered by every black-box attacks (except the padding one). As already stated by Demetrio et al. [8], the black-box Gamma algorithm is effective against MalConv, but using such a low threshold for the GBDT, this attack is not effective against the tree model. Demetrio et al. [8] considered the standard detection, where the original threshold [3] is used (that is 0.871). We set the regularization parameter $\lambda = 10^{-5}$ to match the results shown by the authors. This strategy is ineffective against the other networks since it leverage content appending, and, as discussed in Section 4.1, it is useless against the network proposed by Coull et al. [5].

5 RELATED WORK

In this section we briefly review some concurrent work proposing a similar attack framework, provide additional details on attacks against learning-based malware detectors and other learning-based detection methods, and conclude by discussing relationships between the problem of adversarial malware to packing and obfuscation techniques.

5.1 Other attack frameworks

Recent concurrent work by Pierazzi et al. [19] propose a general formalization for defining the optimization of adversarial attacks inside the input space, spanning multiple domains like image and speech recognition and Android malware detection. They define sequences of practical manipulations that must preserve the original semantics. These manipulations need to be imperceptible to manual inspection, and they must be resilient to pre-processing techniques. The authors also explicit define the resulting side-effects generated by applying such mutations to the original sample, as a summation of vectors. The attacker optimizes the sequences of practical manipulations that satisfy all the constraints mentioned above by exploring the space of mutations imposed by such practical manipulations. Our formalization shares the use of practical manipulations applied inside the input space, and we also minimize the applications of practical manipulations to compute adversarial examples. Furthermore, our practical manipulations are semantics-invariant by design, including all the constraints proposed by Pierazzi et al, and removing the need of the side-effect vector. We do not enforce the robustness to the pre-processing step, as the defender needs to know these manipulations in advance. This might not be always possible: the two novel attacks we propose act as a zero-day, and only after the discovery of such techniques can they be patched properly. We generalize the objective function to optimize by including a loss function, and the attacker can choose how to implement it by using a function of their choice and adding constraints expressed as regularization parameters. As opposed to Pierazzi et al., we explicitly express the variables to optimize inside the optimization problem, since they are the parameters of the practical manipulations.

5.2 Attacks against malware detectors

Many of the white-box attacks in the space of malware classifiers have been previously introduced in this paper. For completeness, we revisit their respective contributions to the literature here. Kolosnjaji et al. [14] propose an attack against MalConv that leverage padding bytes at the end of the input sample and chooses the best local approximation of each padding value. Demetrio et al. [7] propose the *Partial DOS* practical manipulation against the MalConv classifier. Both strategies are easy to apply, but they are not as effective as the novel manipulations we have proposed, as shown in Figure 5 and Figure 7, in both white-box and black-box settings. Also, the number of bytes altered by padding and partial dos are either too few or placed in locations with zero gradient, hence useless during the optimization.

Kreuk et al. [16] propose an iterative variant of the Fast Gradient Sign Method (FGSM) [9] by manipulating malware inside the feature space imposed by the target network MalConv using both padding and slack space bytes. Similarly, Suciu et al. [24] apply the classic non-iterative FGSM in feature space by adding bytes to slack space between sections. Both strategies alter the sample inside the feature space, reconstructing a real adversarial EXEmple only at the end of the iteration, which might reduce the adversarial payload effectiveness.

Sharif et al. [22] propose semantics-preserving practical manipulations that alter the code of the input executable, and evaluate against MalConv and the network proposed by Krvcal et al. [15]. For instance, they alter math operations, registers, operand and they add instructions without side-effect on the original behavior of the program. They apply such manipulations at random to each function of the executable, keeping them if the resulting feature vector is aligned with the gradient. Our approach is entirely guided by the gradient of the target function, and it does not leverage randomness while searching for adversarial examples. Also, our practical manipulations target the structure of the executable rather than its code, minimizing the size of the perturbation.

5.3 Malware detection through machine learning

We review other techniques that have been produced in the literature to spot malware using machine learning technology. Saxe et al. [21] develop a deep neural network which is trained on top of a feature extraction phase. The authors consider type-agnostic features, such as imports, bytes and strings distributions along with metadata taken from the headers, for a total of 1024 input variables. Kolosnjaji et al. [14] propose to track which APIs are called by a malware, capturing the execution trace using the Cuckoo sandbox,[11] that is a dynamic analysis virtual environment for testing malware. Hardy et al. [10] statically extract which APIs are called by a program, and they train a deep network over this representation. David et al. [6] develop a network that learns new signatures from input malware, by posing the issue as a reconstruction problem. The network infers a new representation of the data, in an end-to-end fashion. These new signatures can be used as input for other machine learning algorithms. Incer et al. [11] try to tackle the issue of an adversarially robust classifier by imposing monotonic constraints over the features used for the classification tasks. Krčál et al.[15] propose a similar architecture as the one developed by Johns[12] and Coull et al. [5]: a deep convolutional neural network trained on raw bytes. Both architectures share a first embedding layer, followed by convolutional layers with ReLU activation functions. Krčál et al. use of more fully dense connected layers, with Scaled Exponential Linear Units (SeLU) [12] activation functions.

5.4 Lessons learned with packing

Another way to achieve evasion without applying any optimization is leveraging packing techniques. Initially designed to save on disk space and protect intellectual property, packing is often used to obfuscate the representation of input programs. This causes an increase in the difficulty of studying packed samples, both malign or benign, using static analysis techniques. In this scenario, machine-learning techniques are not a disruptive technology for detecting threats based only on static information, as described by Aghakhani et al. [1]. The authors of the work studied how packing decreases the meaningfulness of different static feature sets, by destroying the original representation. On the other hand, content obfuscation by packing leads to the creation of a new program itself, and it can be seen as a very intrusive way of hiding the malicious content from static analysis-based classifiers.

From the perspective of adversarial machine learning, we are interested in sizing the worst case and evaluate adversarial robustness of machine-learning models by showing that even very small, non-invasive perturbations of the input program can successfully break detection, without the need of packing or obfuscating the whole input program. The goal of our analyses is to demonstrate how brittle learning-based models can be in face of perturbations carefully optimized against them, rather than showing that static code analysis can be bypassed by packing the whole program. We do believe that this is really important, as similar issues may also be found for learning-based models trained on features extracted from dynamic code analysis. In particular, a learning-based model trained on such features may anyway learn to discriminate between legitimate and malware programs based only a small subset of (very discriminant) feature values. In this case, even a small change to such feature values may allowing evading malware detection. For this reason, we believe that understanding and quantifying adversarial robustness of learning-based malware detectors

[11] https://cuckoosandbox.org/

[12] https://www.camlis.org/2017/jeffreyjohns

may not only unveil different, novel vulnerabilities, but that it also constitutes a very important research direction to improve and design more robust models in this space.

6 CONCLUSIONS

We conclude the paper by discussing the limitations of our attacks and some mitigation strategies to prevent them, along with what we have learned and achieved in this work.

Limitations and Mitigations: Since our content-injection attacks must comply with specific constraints posed by the format, the attacker can not add adversarial payloads freely. The injected content must not interfere with the file alignment specified inside the header, or the sample will be corrupted and unable to execute. As we have shown how to attach an adversarial payload to a program, we now discuss possible countermeasures that can be considered to avoid these particular attacks. All header attacks can be easily patched before the classification step by filtering out all the content between the magic number `MZ` and offset `0x3c`, plus all the content between offset `0x40` and the header identifier `PE`. This will ensure the failure of these attacks, with no particular effort required for the defender. Also, content-shifting attacks can be sanitized by looking at whether the beginning of the first section and the end of the optional header match. The defender can get rid of these manipulations by either erasing the payload or shifting all the content backward and reverting altered section entries. In both cases, the adversarial payload is deleted, and the classification step can be applied without further complications. Since we are interested in finding minimal perturbations that alter the decision process, we focused on less invasive alterations, in contrast to the one proposed by Sharif et al. [22], whose application alters the code section of the input program.

Contributions and future work: We propose RAMEn, a lightweight formalization that encapsulates all the needs of the attacker, with all the practical manipulations applicable in the domain of choice and with all the constraints expressed as a penalty term inside the optimization process. We define and apply new practical manipulations, crafted for the domain of malware detection of Windows PE programs. We take into account state-of-the-art deep learning classifiers, presenting successfully evasive adversarial malware against them, in both white-box and black-box settings, since RAMEn can be implemented in both ways. The amount of noise added to the original malware samples is below 2% of the input size of the target network, and it can be considered imperceptible to the eyes of an expert analyst. We also test the performance of transfer attacks, showing how an attacker can take advantage of only using surrogate models they own. We show how all the attacks proposed in the state of the art literature can be reduced to our formalization without loss of generality, and implemented accordingly. Results show the effectiveness of the proposed strategy, in particular the ones that shift the content of the input samples. Since semantics have multiple syntactical representations, it can hardly be inferred by static features, as syntax can be easily twisted to shape adversarial malware that evades detection.

We believe that future lines of research should include the study of which feature sets are easier to perturb with practical manipulations to craft adversarial malware and which are not. The formalization we propose is general enough for including also attacks against dynamic and hybrid detectors, since RAMEn is highly modular, and it can comply with the attacker's needs. On a different note, it would be interesting to improve the robustness of machine learning malware classifiers by leveraging domain knowledge in the form of constraints and regularizers, e.g. , to capture invariant transformations known to domain experts that may modify the input program bytes but preserve its semantics and functionality [17]. This would enable learning robust algorithms against such transformations in a very efficient manner, without the need of performing *adversarial training*, i.e. , generating the actual adversairal EXEmples and retrain the model using them. In fact, such adversarial training procedure may anyway remain ineffective due to the high dimensionality of the input space and variability of transformations. For this reason, we believe that encoding domain knowledge directly into the learning process may substantially improve model robustness by bridging the gap between models that are learned entirely in a data-driven manner and the design of hand-crafted feature representations for them.

REFERENCES

[1] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel. When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

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

[3] H. S. Anderson and P. Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[4] R. L. Castro, C. Schmitt, and G. Dreo. Aimed: Evolving malware with genetic programming to evade detection. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 240–247. IEEE, 2019.

[5] S. E. Coull and C. Gardner. Activation analysis of a byte-based deep neural network for malware classification. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 21–27. IEEE, 2019.

[6] O. E. David and N. S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015.

[7] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. *Proceedings of the Third Italian Conference on CyberSecurity (ITASEC)*, 2019.

[8] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando. Efficient black-box optimization of adversarial windows malware with constrained manipulations, 2020.

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

[10] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li. Dl4md: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Mining (DMIN)*, page 61. The Steering Committee of The World Congress in Computer Science, Computer …, 2016.

[11] I. Incer, M. Theodorides, S. Afroz, and D. Wagner. Adversarially robust malware detection using monotonic classification. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, pages 54–63. ACM, 2018.

[12] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017.

[13] 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. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

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

[15] M. Krčál, O. Švec, M. Bálek, and O. Jašek. Deep convolutional malware classifiers can learn from raw executables and labels only. *Sixth International Conference on Learning Representations (ICLR) Workshop*, 2018.

[16] 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. *arXiv preprint arXiv:1802.04528*, 2018.

[17] S. Melacci, G. Ciravegna, A. Sotgiu, A. Demontis, B. Biggio, M. Gori, and F. Roli. Can domain knowledge alleviate adversarial attacks in multi-label classifiers?, 2020.

[18] M. Melis, A. Demontis, M. Pintor, A. Sotgiu, and B. Biggio. secml: A python library for secure and explainable machine learning, 2019.

[19] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. *arXiv preprint arXiv:1911.02142*, 2019.

[20] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[21] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, pages 11–20. IEEE, 2015.

[22] M. Sharif, K. Lucas, L. Bauer, M. K. Reiter, and S. Shintre. Optimization-guided binary diversification to mislead neural networks for malware detection. *arXiv preprint arXiv:1912.09064*, 2019.

[23] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin. Automatic generation of adversarial examples for interpreting malware classifiers. *arXiv preprint arXiv:2003.03100*, 2020.

[24] O. Suciu, S. E. Coull, and J. Johns. Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 8–14. IEEE, 2019.