Functionality-Preserving Black-Box Optimization of Adversarial Windows Malware

Luca Demetrio, Battista Biggio, *Senior Member, IEEE*, Giovanni Lagorio, Fabio Roli, *Fellow, IEEE*, and Alessandro Armando

***Abstract*— Windows malware detectors based on machine learning are vulnerable to adversarial examples, even if the attacker is only given black-box query access to the model. The main drawback of these attacks is that: (*i*) they are query-inefficient, as they rely on iteratively applying random transformations to the input malware; and (*ii*) they may also require executing the adversarial malware in a sandbox at each iteration of the optimization process, to ensure that its intrusive functionality is preserved. In this paper, we overcome these issues by presenting a novel family of black-box attacks that are both query-efficient and functionality-preserving, as they rely on the injection of benign content (which will never be executed) either at the end of the malicious file, or within some newly-created sections. Our attacks are formalized as a constrained minimization problem which also enables optimizing the trade-off between the probability of evading detection and the size of the injected payload. We empirically investigate this trade-off on two popular static Windows malware detectors, and show that our black-box attacks can bypass them with only few queries and small payloads, even when they only return the predicted labels. We also evaluate whether our attacks transfer to other commercial antivirus solutions, and surprisingly find that they can evade, on average, more than 12 commercial antivirus engines. We conclude by discussing the limitations of our approach, and its possible future extensions to target malware classifiers based on dynamic analysis.**

***Index Terms*— Adversarial examples, malware detection, evasion attacks, black-box optimization, machine learning.**

I. Introduction

MACHINE learning is becoming ubiquitous in the field of computer security. Both academia and industry are investing time, money and human resources to apply these statistical techniques to solve the daunting task of malware detection. In particular, Windows malware is still a threat in the wild, as thousands of malicious programs are uploaded to VirusTotal every day.[1] Modern approaches use machine learning to detect such threats at scale, leveraging many different learning algorithms and feature sets [1]–[7].

While these techniques have shown promising malware-detection capabilities, they have not been originally designed to deal with non-stationary, adversarial problems in which attackers can manipulate the input data to evade detection. This has been widely shown in the last decade in the area of *adversarial machine learning* [8], [9]. This research field studies the security aspects of machine-learning algorithms under attacks staged either at training or at test time. In particular, in the context of learning-based Windows malware detectors, it has been shown that it is possible to carefully optimize *adversarial malware* samples against the target system to bypass it [10]–[17]. Many of these attacks have been demonstrated in the black-box setting in which the attacker has only query access to the target model [14]–[17]. This really questions the security of such systems when deployed as *cloud services*, as they can be queried by external attackers who can in turn optimize their manipulations based on the feedback provided by the target system, until evasion is achieved.

These black-box attacks are however still not very efficient in terms of (*i*) the number of required queries, (*ii*) the complexity of their optimization process, and (*iii*) the amount of manipulations performed on the input sample, as detailed below. First, query efficiency is hindered by the fact that these attacks optimize the adversarial malware by iteratively applying transformations which are not specifically targeted to evade detection, like injection of *random* bytes after the end of the file. Second, the optimization process may be quite computationally demanding as some attacks require executing the adversarial malware sample in a sandbox at each iteration to ensure that its intrusive functionality is preserved. This verification step is required by attacks that either manipulate data in feature space (rather than considering realizable input modifications [18]), or consider input transformations that may break the functionality of the malware sample [14], [19]. While executing the malware sample once inside a sandbox may not significantly slow down the whole process, the problem becomes relevant when this step has to be repeated after each iteration of the optimization process, as it requires restoring the state of the virtual environment at the stage before infection. In addition, many malware samples can detect if they are run in a virtual environment and delay their execution to stay undetected [31]. This makes the problem of verifying that malware functionality is preserved even more complicated.

Manuscript received February 20, 2020; revised September 29, 2020, February 17, 2021, and April 30, 2021; accepted May 11, 2021. Date of publication May 20, 2021; date of current version June 4, 2021. This work was supported by the Progetti di Rilevante Interesse Nazionale (PRIN) 2017 Project RexLearn through the Italian Ministry of Education, University and Research, under Grant 2017TWNMH2. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Loukas Lazos. *(Corresponding author: Luca Demetrio.)*

Luca Demetrio is with the PRA Lab, Department of Electrical and Electronic Engineering, University of Cagliari, 09124 Cagliari, Italy (e-mail: luca.demetrio93@unica.it).

Battista Biggio and Fabio Roli are with the PRA Lab, Department of Electrical and Electronic Engineering, University of Cagliari, 09124 Cagliari, Italy, and also with Pluribus One, 09128 Cagliari, Italy.

Giovanni Lagorio and Alessandro Armando are with the Computer Security Laboratory (CSecLab), University of Genoa, 16126 Genoa, Italy.

Digital Object Identifier 10.1109/TIFS.2021.3082330

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

Third, all these attacks achieve evasion by significantly manipulating the content of the input malware, without considering additional constraints, e.g., on the resulting file size or number of injected sections. This may result in attack samples that are easily detected as anomalous by only looking at some trivial characteristics, like the file size or the number of sections.

In this paper, we aim to overcome the aforementioned limitations by proposing a novel family of black-box attacks (Sect. III) that can efficiently optimize adversarial malware samples. First, our attacks are *query-efficient*, as they rely upon injecting content specifically targeted to facilitate evasion, i.e., extracted from *benign* samples (instead of being randomly generated). Second, they are *functionality-preserving* by design, as they leverage a set of manipulations that only inject content into the malicious program by exploiting ambiguities of the file format used to store programs on disk, without altering its execution traces. While in this work we only focus on injecting content either at the end of the file (*padding*) or within some newly-created sections (*section injection*), our approach is general enough to encompass a wider range of functionality-preserving manipulations (as discussed in Sect. III-A). Finally, our attacks are *stealthier*. In particular, they are formalized as a constrained minimization problem which does not only optimize the probability of evading detection, but also penalizes the size of the injected adversarial payload via a specific regularization term.

We focus on two popular learning-based Windows malware detectors, built on features extracted from static code analysis (Sect. II). Our empirical evaluation (Sect. IV) investigates the trade-off between detection and size empirically, and shows that our black-box attacks are able to efficiently bypass the considered detectors after only few iterations and changes. Moreover, we show that our attacks succeed not only when the target models output a continuous probability (or confidence) score, but also when they only provide the predicted labels. We then evaluate whether our attacks transfer to other commercial antivirus solutions, and surprisingly find that they can evade, on average, more than 12 commercial antivirus engines. We discuss how related work differs from ours in Sect. V, and acknowledge the limitations of our work in Sect. VI. We conclude by discussing possible future extensions of this work (Sect. VII), including how to extend it to target malware classifiers based on dynamic analysis.

II. Programs and Malware Detection

In this section we first discuss the Windows *Portable Executable* (PE) format,[2] which describes how programs are stored on disk, and explains to the operating system (OS) how to load them in memory before execution. We then introduce the two popular learning-based Windows malware detectors used in the remainder of this work.

A. The Windows Portable Executable (PE) File Format

The Windows PE format consists of several components, as shown in Fig. 1 and described below.

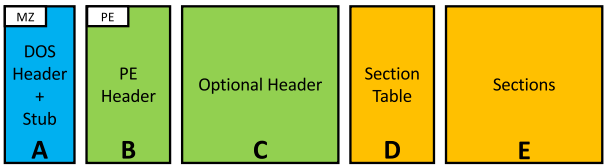


Fig. 1. The Windows PE file format. Each colored section describes a particular characteristic of the program.

1) DOS Header (A): It contains metadata for loading the executable inside a DOS environment, and the *DOS stub*, which prints "*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's operating system. From the perspective of a modern application, the only relevant portions present inside the DOS Header are: (i) the magic number `MZ`, a two-byte long signature for the file, and (ii) the four-byte long integer at offset `$0x3$c`, that works as a pointer to the real header. If one of these two values is scrambled for some reason, the program is considered corrupted, and it will not be executed by the OS.

2) PE Header (B): It contains the magic number `PE` along with other file characteristics, such as the target architecture, the header size, and the file attributes.

3) Optional Header (C): It contains the information needed by the OS to initialize the loading program. It also contains offsets that point to useful structures, like the Import Address Table (IAT), needed by the OS for resolving dependencies, and the Export Table offset, which indicates where to find functions that can be referenced by other programs.

4) Section Table (D): It is a list of entries that indicates the characteristics of each core component of the program, and where the OS loader should find them inside the file.

5) Sections (E): These contiguous chunks of bytes host the real content of the executable. To list a few: *.text* contains the code, *.data* contains global variables and *.rdata* contains read-only constants, and counting.

The structure of an executable program can be useful for statically inferring information about its behavior. Indeed, most antivirus vendors apply static analysis to detect threats in the wild, without executing suspicious programs inside a controlled environment. This approach saves time and resources since the antivirus programs do not execute the suspicious software inside the host OS. Static analysis serves as the first line of defense, and its performance is crucial for opposing the countless threats in the wild.

B. Learning-Based Windows Malware Detection

We focus on two popular, state-of-the-art machine learning-based detectors that have been coded, trained, and publicly released on GitHub by EndGame.[3] Both models are trained on the EMBER dataset built by the same company [6].

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

[3] https://www.endgame.com/

1) MalConv: The first detector is an end-to-end convolutional neural network (CNN) proposed by Raff *et al.* [7]. It takes as input the first 2 MB of an executable and returns the probability of being malware. If the input executable length exceeds this threshold, the file is truncated to the specified size, otherwise, the file is padded with the value `0`. Since the padding value should be unique, all values are shifted by one to maintain this distinction. Each byte is embedded into a representation space with eight dimensions, learned directly from the available data with the goal of defining a meaningful distance metric between bytes. The convolutional layers are then used to correlate spatially-distant bytes inside the input binary, e.g., jumps and function calls.

2) GBDT: The second detector is implemented using Gradient Boosting Decision Trees (GBDT) [6], [20]. Differently from MalConv, this detector uses a fixed representation consisting of 2,381 features, extracted from: (i) *general file information*, including the virtual size of the file, the number of imported and exported functions, the presence of debug sections, etc.; (ii) *header information*, accounting for the characteristics of the executable, the target architecture, the version, etc.; (iii) *byte histogram*, which counts the occurrences of each byte, divided by the total number of bytes; (iv) *byte-entropy histogram*, inspired from [1], which accounts for the entropy of the byte distribution of the file, applying a sliding window over the binary; (v) *information taken from strings*, which counts the number of occurrences of each string (considered as a sequence of at least five consecutive printable characters), and how many special markers they contain, such as `C:\`, `HKEY`, `http` and `https`; (vi) *section information*, which includes name, length, entropy, and virtual size of each section; (vii) *imported and exported functions* which tracks all the functions imported from libraries, and all the ones that are exposed to the other programs. Many of these feature sets are compressed inside a histogram by applying the *hashing trick* [21], to reduce the dimension of the problem to a smaller and manageable space.

III. Black-Box Optimization of Adversarial Windows Malware

In this section, we present our novel black-box attack framework, named GAMMA (Genetic Adversarial Machine learning Malware Attack). GAMMA can efficiently optimize adversarial malware samples while only requiring black-box access to the model, i.e., by only querying the target model and observing its output, without accessing its internal structure and parameters. Our attack relies upon a set of *functionality-preserving* manipulations that inject content into the malicious program by exploiting ambiguities of the PE format used to store programs on disk, without altering its execution traces. This allow us to get rid of the computationally-demanding validation steps required to ensure that the manipulated malware preserves its intended functionality. In particular, we consider here content manipulations specifically targeted to facilitate evasion, i.e., extracted from *benign* samples rather than being generated at random. While this makes our attack much more *query-efficient*, it is worth remarking that our framework is general enough to encompass many other different content manipulation techniques, as detailed in Sect. III-A. Finally, to make our attack *stealthier*, we formalize it as a constrained optimization problem which does not only minimize the probability of evading detection but also the size of the injected content via a specific penalty term.

Notation: In the following, we denote with $\boldsymbol{x} \in \mathcal{X} \subset \{0, \ldots, 255\}^*$ the (malicious) input program, described as a string of bytes of arbitrary length. We then define a set of k distinct functionality-preserving manipulations that can be applied to the input program $\boldsymbol{x}$ as a vector $\boldsymbol{s} \in \mathcal{S} \subset [0, 1]^k$. Each element of $\boldsymbol{s}$ corresponds to a different manipulation that can be applied to the input program. The manipulations are parameterized in [0, 1], to denote the extent to which they are applied. For example, if we assume that the i^{th} element s_i is associated to the injection of a given section in the input program, $s_i = 0.4$ may represent the fact that only 40% of the bytes present in that section will be injected. We can also consider injection of specific API functions, in which case s_i will be a binary variable denoting whether the given API is injected ($s_i = 1$) or not ($s_i = 0$). The function $\oplus : \mathcal{X} \times \mathcal{S} \rightarrow \mathcal{X}$ applies the manipulations described by $\boldsymbol{s}$ to the input program $\boldsymbol{x}$, preserving functionality, and returns the manipulated program. We use $f : \mathcal{X} \rightarrow \mathbb{R}$ to denote the output of the classification model on the input program. Without loss of generality, we consider here f to be the output of the model on the malicious class, i.e., the higher the value of $f(\boldsymbol{x})$ is, the more $\boldsymbol{x}$ is considered malicious. The value of $f(\boldsymbol{x})$ is eventually compared against a decision threshold θ to decide whether the input program is malicious, i.e., $f(\boldsymbol{x}) \geq \theta$, or not.

Attack Formulation: We can now formalize our attack as the following constrained minimization problem:

$$\begin{aligned} &\underset{\boldsymbol{s} \in \mathcal{S}}{\text{minimize}} \;\; F(\boldsymbol{s}) = f(\boldsymbol{x} \oplus \boldsymbol{s}) + \lambda \cdot \mathcal{C}(\boldsymbol{s}), \\ &\text{subject to } q \leq T. \end{aligned} \tag{1}$$

The objective function $F(\boldsymbol{s})$ consists of two conflicting terms: (i) $f(\boldsymbol{x} \oplus \boldsymbol{s})$, i.e., the classification output on the manipulated program, and (ii) $\mathcal{C}(\boldsymbol{s})$, i.e., a penalty function that evaluates the number of injected bytes into the input malware. The hyperparameter $\lambda > 0$ tunes the trade-off between these two terms, i.e., it promotes solutions with smaller number of injected bytes $\mathcal{C}(\boldsymbol{s})$ at the expense of reducing the probability that the sample is misclassified as benign (larger $f(\boldsymbol{x} \oplus \boldsymbol{s})$ values). Varying the hyperparameter λ allows us to evaluate how the attack effectiveness increases as a function of the size of the injected adversarial payload.

The objective F is minimized w.r.t. the choice of the applied manipulations $\boldsymbol{s}$. In this work, we restrict the available manipulations $\boldsymbol{s} = (s_1, \ldots, s_k)$ to the injection of content extracted from a predefined set of k benign sections, without optimizing the content-injection location. This means that s_i will represent the fraction of bytes extracted from the i^{th} benign section, and these bytes will be injected before those extracted from section s_j, for $j > i$. In this context, we define the penalty term $\mathcal{C}(\boldsymbol{s})$ as $\mathcal{C}(\boldsymbol{s}) = \boldsymbol{c}^T \boldsymbol{s}$, where $\boldsymbol{c} \in \mathbb{R}^k$ is a vector whose i^{th} element c_i is equal to the overall size of the

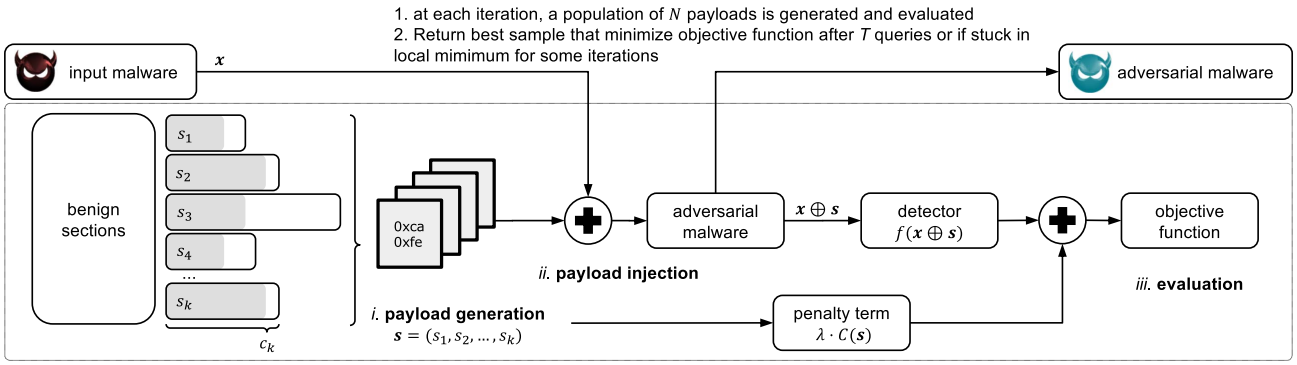


Fig. 2. Conceptual schema of GAMMA. The optimizer (i) generates different payload extracted from benign programs, (ii) injects such content inside the original malware, and (iii) computes the objective function, by combining the response of the detector and the size constraint controlled by λ. The process ends after T queries, or if it converges earlier, i.e., if the objective function does not significantly decrease after a given number of iterations.

i^{th} benign section. Accordingly, $\mathcal{C}(\boldsymbol{s})$ measures the size of the injected payload. As the elements of $\boldsymbol{c}$ and $\boldsymbol{s}$ are non-negative, this term penalizes a weighted version of the ℓ_1 norm of $\boldsymbol{s}$, thus promoting sparse solutions. This means that many elements of the optimal solution vector $\boldsymbol{s}^\star$ will be zero, i.e., only content from few benign sections will be injected.

As optimizing the objective in a black-box manner requires querying the target model f repeatedly, we use the constraint $q \leq T$ to upper bound the maximum number of queries q that can be performed by T. The query budget T is another hyperparameter in our approach, and increasing it allows our attack to better optimize the trade-off between misclassification confidence and payload size, at the expense of an increased computational complexity.

Solution Algorithm: We solve the given minimization problem using a black-box genetic optimizer, as detailed in Algorithm 1 and Fig. 2. The algorithm is initialized by randomly generating a matrix $\mathbf{S}' = (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_N) \in \mathcal{S}^N \subset [0,1]^{N \times k}$, which represents the initial population of N candidate manipulation vectors (line 2). Then, the genetic algorithm iterates over three steps that mimic the process of biological evolution: *selection*, *cross-over*, and *mutation*. The *selection* step (line 4) uses the objective function to evaluate the candidates in $\mathbf{S}'$, and selects the best N candidates between the current population $\mathbf{S}'$ and the population generated at the previous iteration $\mathbf{S}$. These are the candidate manipulation vectors associated with the lowest values of F. The *crossover* function (line 5) takes the selected candidates as input and returns a novel set of N candidates by mixing the values of pairs of randomly-chosen vector candidates. In particular, given a pair of candidate vectors from the previous population, a new candidate is generated by cloning the values $s_1, \ldots, s_j$ from the first parent and the remaining values $s_{j+1}, \ldots, s_k$ from the second parent, being $j \in \{1, \ldots, k\}$ an index selected at random. The *mutation* function (line 6) changes the elements of each input vector at random, with low probability. The combination of both *cross-over* and *mutation* ensures that the new population is sufficiently different from the previous one, allowing the algorithm to properly explore the space of feasible solutions.

In each iteration, the algorithm performs N new queries to the target model, to evaluate the objective F on the new candidates in $\mathbf{S}'$, and then retains the best candidate population $\mathbf{S}$. When either the maximum number of queries T or convergence is reached (e.g., if no further improvement in the value of F is observed across a given number of iterations), the algorithm returns the best manipulation vector $\boldsymbol{s}^\star$ from the current population $\mathbf{S}$. The corresponding optimal adversarial malware $\boldsymbol{x}^\star$ can be finally obtained by applying the optimal manipulation vector $\boldsymbol{s}^\star$ to the input sample $\boldsymbol{x}$ through the manipulation operator $\oplus$ as $\boldsymbol{x}^\star = \boldsymbol{x} \oplus \boldsymbol{s}^\star$.

Algorithm 1: Genetic Optimization of Adversarial Malware With GAMMA

Input : $\boldsymbol{x}$, the initial malware sample; λ, the regularization parameter; N, the population size; T, the query budget.
Output: $\boldsymbol{s}^\star$, the manipulations which minimize F.

1 $q \leftarrow 0$, $\mathbf{S} \leftarrow \emptyset$
2 $\mathbf{S}' \leftarrow (\boldsymbol{s}_1, \ldots, \boldsymbol{s}_N) \in \mathcal{S}^N$
3 **while** $q < T$ ***and*** *not converged* **do**
4 $\quad \mathbf{S} \leftarrow \text{selection}(\mathbf{S} \cup \mathbf{S}', F, \boldsymbol{x}, \lambda)$
5 $\quad \mathbf{S}' \leftarrow \text{crossover}(\mathbf{S})$
6 $\quad \mathbf{S}' \leftarrow \text{mutate}(\mathbf{S}')$
7 $\quad q \leftarrow q + N$
8 **return** $\boldsymbol{s}^\star$, best candidate from $\mathbf{S}$ with minimum F.

Hard-Label Attacks: In some cases, the target model may only provide the classification label assigned to the input sample, instead of a continuous confidence value $f(\boldsymbol{x})$. In this hard-label scenario, we adapt GAMMA by setting $f(\boldsymbol{x}) = 0$ if the input sample is classified as benign, and $f(\boldsymbol{x}) = \infty$ otherwise, to discard perturbed malware samples that do not evade detection. This basically amounts to performing random transformations to the input malware until an evasive variant is found, and reducing the injected payload size $\mathcal{C}(\boldsymbol{s})$ only afterwards, while trying to preserve misclassification. Notably, this random exploration does not substantially hinder the success of our algorithm, as evasive malware variants can be typically found by injecting a sufficiently-large amount of benign content into the initial malware.

A. Functionality-Preserving Manipulations

We discuss here the set of functionality-preserving manipulations that can be used in our attack framework. In the context of Windows PE file format, there are only a few transformations that can be applied without compromising the execution of the input program. We categorize them either as structural or behavioral, as detailed below.

1) Structural: This family of manipulations affects only the structure of the input program, by exploiting ambiguities inside the file format, without altering its behaviour.

(s.1) Perturb Header Fields [13]–[15]. This technique includes altering section names, breaking the checksum, and altering debug information. These are fine-grained manipulations that can be applied to the PE components *B*, *C* and *D* in Fig. 1.

(s.2) Filling Slack Space [12]–[15], [23]. This technique manipulates the slack space inserted by the compiler to maintain the alignments inside the file. The corresponding slack bytes (inside *E* in Fig. 1) are usually set to zero, and they are never referenced by the code of the executable.

(s.3) Padding [11], [12], [23]. This technique injects additional bytes at the end of the file (after *E* in Fig. 1).

(s.4) Manipulating DOS Header and Stub [10], [22]. This technique modifies some bytes in the DOS Header (*A* in Fig. 1) which are not used by modern programs (see Sect. II).

(s.5) Extend the DOS Header [22]. This technique extends the DOS header by injecting content before the actual header of the program (between *A* and *B* in Fig. 1).

(s.6) Content shifting [22]. This technique creates additional space before the beginning of a section, by shifting the content forward, and injects adversarial content in between (i.e., between *D* and *E* in Fig. 1).

(s.7) Import Function Injection [13]–[15]. This technique injects import functions by adding an appropriate entry to the Import Address Table, specifying which function from which library must be included during the loading process (this affects *C* and *E* in Fig. 1).

(s.8) Section Injection [13]–[15]. This technique injects new sections into the input file by creating an additional entry inside the section table (thus affecting *D* and *E* in Fig. 1). Each section entry is 40 bytes long, so all the content has to be shifted by that amount, without compromising file and section alignments as specified by the header.

2) Behavioral: This family of perturbations can change the program behavior and execution traces, but still preserving the intended functionality of the malware program. For example, these transformations encompass the binary rewriting techniques in [24], as discussed below.

(b.1) Packing [13]–[15]. This technique amounts to encrypting or encoding the content of the binary inside another binary and decoding it at run-time. The effect of a packer is invasive since the whole structure of the input sample is modified.

(b.2) Direct [24]. This approach rewrites specific portions of the code, like replacing assembly instructions with equivalent ones (e.g., additions and subtractions with opposed sign).

(b.3) Minimal Invasive [15], [24]. This technique sets the entry-point to a new executable section that jumps back to the original code.

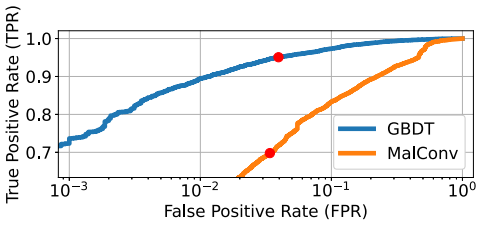


Fig. 3. Receiver Operating Characteristic (ROC) curve of both classifiers.

(b.4) Full Translation [24]. This approach lifts all the code to a higher representation, e.g., LLVM,[4] since it simplifies the application of perturbations, and it then translates the code back to the assembly language.

(b.5) Dropper [30]. This approach stores the code as a resource of another binary, which is then loaded at runtime.

3) Padding and Section-Injection Attacks: In this work, we implement GAMMA using two different structural manipulation techniques, i.e., *padding* and *section injection*, and refer to them respectively as *padding* and *section-injection* attacks. While GAMMA can support most of the aforementioned manipulations via their open-source implementations in [33], we only consider *padding* and *section-injection* attacks in this work as they provide two representative examples of injecting content inside the sample *with* and *without* requiring manipulating additional header components (e.g., the section table). In particular, similarly to $s.1-s.6$, padding injects content into the unused space of the executable, without altering any other header component. Section injection, instead, does not only allow injecting custom content like the other techniques, but it also manipulates the structure of the executable by adding section entries inside the section table.

IV. Experiments

In this section, we empirically evaluate the effectiveness of our attacks against both the GBDT and MalConv malware detectors. We ran our experiments on a workstation equipped with an Intel® Xeon® CPU E5-2670, with 48 CPU and 128 GB of RAM. The pre-trained version of MalConv presents a slightly different architecture w.r.t. the original formulation: 1 MB of input size and padding value of `256` to avoid the shifting pre-processing part. The network is implemented using PyTorch [25]. We developed the genetic optimizer of GAMMA using *DEAP* [26]. We tested the attack using a population size N of 10 elements, varying the query budget T from 10 to 510. If the optimizer stagnates in a local minimum for more than 5 iterations, we halt the process. We used values for the regularization parameter $\lambda \in \{10^{-i}\}_{i=3}^{9}$. Since the attack feature space $\mathcal{S}$ is parametric over the number of sections the attacker may add, we randomly extract 75 `.rdata` sections from our goodware dataset that will be used for adding content to the input malware, for a maximum of 2.5 MB, as discussed in Section III-A. We willingly set this number high, as the optimizer will find small payloads thanks to the sparsity imposed by the penalization term that behaves as a ℓ_1 norm.

[4]https://llvm.org/

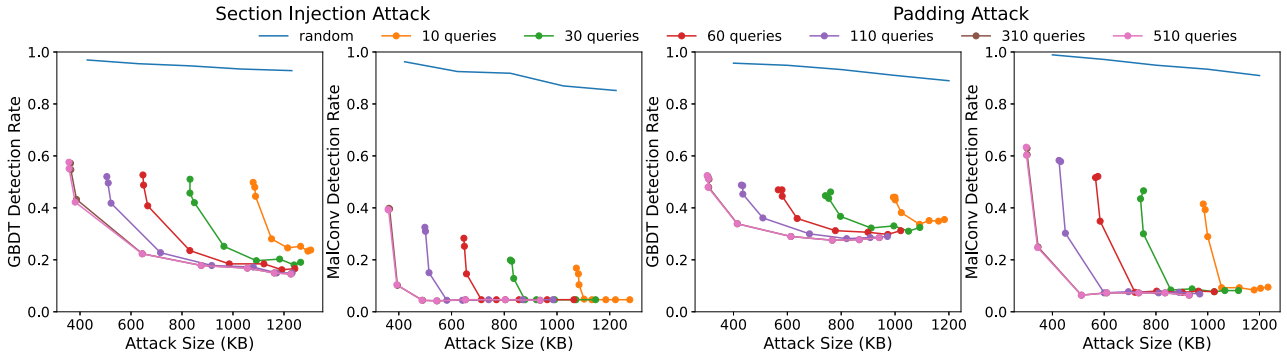


Fig. 4. Padding and section injection attack performances for $\lambda \in \{10^{-i}\}_{i=3}^{9}$, using 500 malware samples as input. The solid lines are computed as a regression over the point of a particular setting of the experiments.

We implemented and open-sourced the library we used for computing these attacks, named `secml-malware` [33].

4) Performance in the Absence of Attack: To evaluate the performance of both classifiers in the absence of attacks, we collected a set of 15,000 benign and 15,000 malware samples.

The malware samples were gathered from VirusTotal,[5] while the goodware samples were collected by downloading executable programs from GitHub. The results are shown in Figure 3. With the suggested classification threshold of 0.8336 [6], GBDT achieves a False Positive Rate (FPR) of 3.9% and a True Positive Rate (TPR) of 95%, while the default threshold of 0.5 suggested for MalConv [6], [7] corresponds to an FPR of 3.5% and a TPR of 69%. The red dots inside the plot show such values directly on the corresponding ROC curves. Results are comparable to the description given by the authors of GBDT [6], as both detectors achieve just a slightly lower score w.r.t. what reported in the paper. Still, they can be both used as a baseline for our analysis.

5) Attack Evaluation: We randomly sample 500 from our collection of 15K malware set to use during the adversarial attacks, and this set includes 5.3% ransomware, 29% downloaders, 18% viruses, 7% backdoors, 29% grayware, 8% worms, plus other families with lower percentage. Figure 4 shows how both the detection rate and adversarial payloads size vary w.r.t. the number of queries and the value of the regularization parameter. Each curve in the plot has been produced by computing the mean detection rate and mean size for each values of λ, repeated for different numbers of queries sent. As the value of λ decreases, the algorithm finds more evasive samples with bigger payloads, since the penalty term is negligible while computing the objective function. On the other hand, by increasing the value of λ the resulting attack feature vector become sparse, generating smaller but more detectable adversarial example. In this case, the penalty term engulfs the score computed by the classifier, which becomes irrelevant during the optimization. Another significant effect is posed by the number of total queries used by the genetic optimizer: the more are sent, the better the adversarial examples are in both detection rate and size. Intuitively, by sending more queries, GAMMA can explore more solutions that are stealthy and evasive at the same time, but such solutions could not be found at early stages of the optimization process. To prove the efficacy of our methodology, we report the results of the application of random byte sequences of increasing length. This experiment highlights a slight descending trend, but the optimized attack with benign content injection is way more effective than random perturbations. The detection rate of GBDT is decreased more by the section-injection attack than by padding. Since the first technique also introduces a section entry inside the section table, the adversarial payload perturbs more features than those modified by the padding attack.

6) Hard-Label Attacks: We show aggregate results in Table I, highlighting the comparisons between the performances of the soft-label and hard-label attacks. Each entry presents the mean detection rate and the mean adversarial payload size for each detector, given a pair of number of queries/regularization parameter used for computing the specified attack. We computed 4 different values of λ in the set $\{10^{-(2i+1)}\}_{i=1}^{4}$. Results suggest that, without the confidence score, once one evasive payload is found, then its size is optimized iteration after iteration of the genetic algorithm, regardless of the value of the regularization parameter λ. This is caused by the settings we impose for our experiments: we used an infinite value to discard each detected adversarial example; hence, all the remaining ones are used for optimizing only the size, acting as a constraint itself for the optimization. The effectiveness of our methodology in this setting is caused by the nature of the injected content, which mimics the benign class, as also confirmed by Figure 4 (injecting random byte sequences has no effect against the targets). Moreover, increasing the number of queries helps reducing the adversarial payload size while only slightly affecting misclassification confidence.

7) Injected Payload Size: We report here some additional insights on the size of the injected payload with respect to the average malware and benign program sizes. In particular, while the source malware samples have an average size of 343 KB (with standard deviation of 300 KB), which is roughly increased by 1-2× after manipulation, they remain comparable in size to goodware programs, for which the average size is 240 KB (with standard deviation of 1 MB).

8) Temporal Analysis: From a temporal point of view, the complexity of GAMMA is dominated by the time spent

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

TABLE I

COMPARISON OF SOFT-LABEL AND HARD-LABEL ATTACKS, WITH DIFFERENT NUMBER OF QUERIES SENT AND VALUES OF λ

	Soft-Label GAMMA (Q = 500)		**Hard-Label GAMMA (Q = 30)**		**Hard-Label GAMMA (Q = 500)**	
Padding	*GBDT*	*MalConv*	*GBDT*	*MalConv*	*GBDT*	*MalConv*
$\lambda = 10^{-9}$	28% / 941 KB	6% / 927 KB				
$\lambda = 10^{-6}$	33% / 413 KB	6% / 511KB	35% / 945 KB	6% / 835 KB	30% / 661 KB	5% / 473 KB
$\lambda = 10^{-3}$	52% / 302 KB	63% / 298 KB				
Section Injection	*GBDT*	*MalConv*	*GBDT*	*MalConv*	*GBDT*	*MalConv*
$\lambda = 10^{-9}$	14% / 1227 KB	4% / 935 KB				
$\lambda = 10^{-6}$	22% / 643 KB	10% 487 KB	31% / 1080 KB	4% / 880 KB	21% / 830 KB	4% / 490KB
$\lambda = 10^{-3}$	57% / 356 KB	39% / 359 KB				

TABLE II

MEAN ELAPSED TIME FOR EACH ATTACK AND TARGET

	GBDT	**MalConv**
Padding	0.60 $\pm$ 0.24s	0.93 $\pm$ 0.33s
Section injection	0.60 $\pm$ 0.30s	0.86 $\pm$ 0.20s

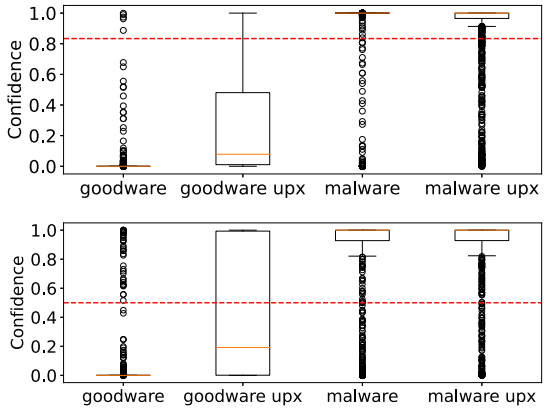


Fig. 5. Effect of UPX packing on GBDT (top) and MalConv (bottom). Each box-plot shows the distribution of the classifier confidence $f(\boldsymbol{x})$ on the malicious class, and the dashed red line is the decision threshold θ.

TABLE III

NUMBER OF ANTIVIRUS PROGRAMS FROM VIRUSTOTAL (VT) THAT DETECT (i) THE INITIAL MALWARE AND ITS MODIFIED VERSIONS WITH (ii) RANDOM AND (iii) SECTION-INJECTION ATTACKS, AVERAGED OVER 200 MALWARE SAMPLES (STANDARD DEVIATION IS ALSO REPORTED). WHILE RANDOM ATTACKS EVADE 5.76 DETECTORS, ON AVERAGE, SECTION-INJECTION ATTACKS EVADE UP TO 12.01 DETECTORS

	Malware	**Random**	**Sect. Injection**
Detections (VT)	46.56 $\pm$ 12.40	40.80 $\pm$ 12.40	34.50 $\pm$ 12.63

TABLE IV

DETECTION RATE OF 9 ANTIVIRUS PROGRAMS FROM VIRUSTOTAL COMPUTED ON (i) THE INITIAL SET OF 200 MALWARE SAMPLES, AND ON THE SAME SAMPLES MANIPULATED WITH (ii) RANDOM ATTACKS AND (iii) SECTION-INJECTION ATTACKS

	Malware	**Random**	**Sect. Injection**
AV1	93.5%	85.5%	30.5%
AV2	85.0%	78.0%	68.0%
AV3	85.0%	46.0%	43.5%
AV4	84.0%	83.5%	63.0%
AV5	83.5%	79.0%	73.0%
AV6	83.5%	82.5%	69.5%
AV7	83.5%	54.5%	52.5%
AV8	76.5%	71.5%	60.5%
AV9	67.0%	54.5%	16.5%

querying the detector. Table II shows the mean elapsed time needed to compute one single query, for each attack and target. Surprisingly, the sum of the time spent by the feature extraction phase and the prediction of GBDT is less than the time needed by the neural network to process all the bytes.

9) Packing Effect: Since these classifiers leverage only static features, it is reasonable to ask ourselves whether encrypting the program content is already sufficient to evade detection, without applying all the techniques we have introduced in Section III. *Packing* is a technique used to reduce the size of an executable, by applying a compression, encryption or encoding algorithm. As the effect of a packer completely changes the program representation on disk, it has been extensively used by malware vendors to hide their product to the analysts, increasing the difficulty of reverse-engineering analyses. In this context, we apply one famous technique, called UPX[6] to 1000 malware and 1000 goodware programs, and test the evasion rate for both MalConv and GBDT. The effectiveness of the UPX packer is shown in Figure 5. Both detectors attribute a malicious score when the sample is packed, and this is intuitive by looking at the box-plot of the packed goodware programs. Both detectors increase their score towards the malware class, while there is only a little change in terms of mean and variance for packed malware.

These results confirm the evidence reported by Aghakhani *et al.* [32], i.e., that the application of packing techniques tends to be learned as a malicious feature by the detector. This is caused by the potential abundance of packed malware inside the training set [6], opposed to the scarcity of packed goodware, along with the fact that packers may leave specific artifacts embedded into the packed samples. For example, the UPX packer creates two executable sections called *UPX0* and *UPX1*, which contain the extraction code and the original compressed program. As a result, models trained on such data might learn these spurious characteristics and tend to classify packed programs as malware.

10) Evaluation on Antivirus Programs (VirusTotal): We evaluate here the impact of our attack on commercial detectors. In this context, we are not interested in evading detection by these commercial programs, e.g., by packing the input

[6]https://upx.github.io

samples, but rather in assessing whether these methods can detect our attacks, given that our attacks only minimally modify the content of the input malware sample. In particular, the manipulations that we apply to our malware samples address only the syntactical structure of each program, and we aim to evaluate here if the application of such transformations can pose a threat to other antivirus programs. We expect that most of the commercial solutions should not be affected by such attacks. We rely on the response retrieved by VirusTotal,[7] which is an online interface for many threat detectors. The service offers an API that can be used for querying the system, by uploading samples from remote. We test the performance of our attack by sending 200 malware samples, before and after injecting the adversarial payload into the sample, optimized using the section-injection attack against the GBDT classifier. We also compare our attack against a baseline random attack that simply adds a random payload of 50 KB to each sample. Table III shows how many antivirus programs hosted on VirusTotal (70 in total) detect the submitted malware samples, on average. While the random attack only slightly decreases the number of detections per sample, the section-injection attack is able to bypass an average of more than 12 detectors per sample. To better evaluate the impact of our attack on individual antivirus programs, in Table IV we report the detection rates of 9 different antivirus products that appear on the 2019 Gartner Magic Quadrant for Endpoint Protection Platforms,[8] including many leading and visionary products, before and after executing the random and section-injection attacks. In many cases, our section-injection attack is able to drastically decrease the detection rate (see, e.g., AV1, AV3, AV7 and AV9), significantly outperforming the random attack (see, e.g., AV1 and AV9). The reason may be that some of these antivirus programs already use static machine learning-based detectors to implement a first line of defense when protecting end-point clients from malware, as also confirmed in their blog or website, and this makes them more vulnerable to our attacks. To conclude, our analysis highlights that these commercial products can be evaded with a transfer attack, and we believe that their detection rate could decrease even more with an optimized attack against them.

V. Related Work

Previous work is significantly different from GAMMA, as it considers different settings and solutions. In particular, related approaches explore the creation of adversarial examples for information-security detectors, leveraging both gradient-based and black-box algorithms, as detailed in the following.

A. Reinforcement Learning

Anderson *et al.* [15] propose a reinforcement learning approach to decide the best sequence of manipulation that leads to evasion. To test the effectiveness of the agent, they also test the application of manipulations picked at random. The model they used as a baseline is a primordial version of the GBDT classifier we have analyzed in this work, trained on fewer samples. To train the policy of the learning agent, they let the model explore the space of adversarial examples, by fixing a budget for the number of queries that can be used. The mean number of queries applied for training these policies is roughly 1600 [15]. The authors do not report the resulting file size of the adversarial malware: the reinforcement learning method contains actions that enlarge the representation on disk, but it is not clear how and how much. Differently, our approach does not need a training phase, as it can be deployed as-is against the remote detector. The transformation we use are functionality-invariant by design, and their application do not alter the execution flow of the program. Lastly, we take into account how much content is added to the input malware, by plugging a regularizer inside the optimization process. In this way, the amount of inserted noise is controlled, and the algorithm can find adversarial examples that not only evade the target classifier, but also they are limited in size.

B. Genetic Strategies

Castro *et al.* [13], [14] apply both a random and genetic algorithm to perturb the input malware, and they test the functionality of the samples at each iteration of the optimization process inside a sandbox. The mutations are the same proposed by Anderson *et al.* [15]. The authors of these work state that they need approximately almost 4 minutes for creating adversarial malware, using 100 queries. No architecture details have been unveiled. We do not need to validate the malware inside a sandbox, as we include domain knowledge inside the mutation process. For this reason, our methods performs 1,400 queries in the same time span. They also do not report which are the most influential mutations that lead to evasion: the latter is crucial, we are dealing with potential vulnerabilities that lies inside statistical algorithms, whose presence is less evident compared to other security breaches.

C. Generative Adversarial Networks

Hu and Tan [17] develop a Generative Adversarial Network (GAN) [27] whose aim is to craft adversarial malware that bypass a target classifier. The network learns which API imports should be added to the original sample, but no real malware is crafted, as that is attack only operates inside the feature space. In contrast, we create functioning malware, as real samples are generated each time.

A recap of the black-box attacks against Windows malware detectors can be found in Table V, where we compare the techniques we mentioned above with our method.

VI. Limitations and Open Issues

We discuss here which aspects of our work can be improved in the near future, by highlighting its current limitations.

A. Countermeasures

This work aims to show that learning-based detectors can be vulnerable to well-crafted attacks, even if they manipulate only a small fraction of the input program. We have however

[7] https://virustotal.com

[8] https://www.microsoft.com/security/blog/2019/08/23/gartner-names-microsoft-a-leader-in-2019-endpoint-protection-platforms-magic-quadrant/

TABLE V
BLACK-BOX ADVERSARIAL ATTACKS ON WINDOWS MALWARE DETECTORS. **FP:** FUNCTIONALITY-PRESERVING TRANSFORMATIONS; **NS:** NO SANDBOXING REQUIRED; **AO:** ATTACK OPTIMIZATION; **ST:** ATTACK STEALTHINESS (E.G., FILE SIZE OPTIMIZATION)

Detector	FP	NS	AO	ST
MalGAN [17]			✓	
ARMED [13]				
AIMED [14]			✓	
RL Agent [15]	✓	✓	✓	
GAMMA	✓	✓	✓	✓

not investigated any potential mitigation strategy against our attacks. One first line of defense may be to consider robust features against our attacks, e.g., features that are not affected by changes performed either at the byte or section level. For example, graph-based representations extracted from static analysis like Abstract Syntax Trees (ASTs) may be used to this end. However, extracting these features is typically much more computationally demanding and, at least in principle, practical transformations that can alter these features may also be derived and added to our optimization framework. A second line of defense may be to improve robustness of the learning algorithm [9], e.g., via adversarial re-training or by developing specific detection mechanisms for our attacks. For instance, it would be interesting to see if our attacks can be detected by identifying specific artifacts (e.g., an excessively large number of sections). It would also be interesting to study how a classifier could be hardened by embedding domain knowledge inside the training pipeline, defining loss functions that are invariant to the application of adversarial manipulations. Another interesting line of defense may be related to analyze the sequence of consecutive queries received from the same source, to detect whether a black-box attack performing correlated queries is taking place. We believe that all these defense strategies, especially if exploited in a complementary manner, may provide an interesting research direction towards designing more robust malware detectors.

B. Dynamic Classifiers

It is finally worth remarking that our approach is clearly not effective against systems that use features computed by dynamically executing the input program, since the manipulations we applied only focus on the structure of an executable without altering its execution. This issue may however be overcome by exploiting behavioral manipulations, like *binary rewriting techniques* [24], which may also alter the execution of the program and the corresponding dynamic feature representation. Considering techniques that may affect dynamic analysis thus constitutes another interesting avenue to extend the impact of our work in the near future.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel family of black-box attacks on learning-based Windows malware detectors that are both query-efficient and functionality-preserving, overcoming the limitations of previous work. Our attacks rely on the injection of benign content (which will never be executed) either at the end of the malicious file, or within newly-created sections, exploiting the ambiguities of the file format used to store programs on disk, without altering its execution traces. The proposed attacks are formalized as a constrained minimization problem which enables optimizing the trade-off between the probability of evading detection and the size of the injected payload. Our extensive empirical evaluation on two popular learning-based Windows malware detectors has shown that our black-box attacks can bypass them with only few queries and very small payloads, even when the target models only output the predicted labels. We have also shown that our attacks can successfully transfer to other commercial antivirus solutions, finding that they can evade, on average, up to 12 commercial antivirus engines available on VirusTotal. Nevertheless, we believe that a optimizing our attacks directly against these detectors might be even more effective.

Future Work: An interesting avenue for future work is related to investigating the applicability of suitable countermeasures against our attacks, as those discussed in Sect. VI, including the use of more robust feature representations (insensitive to byte-based or section-based manipulations) and learning paradigms (via adversarial re-training, specific attack detection mechanisms or the use of domain-knowledge constraints). Another promising research direction is to extend our attack beyond manipulations that only inject content either at the end of the file or within some newly-created sections. We firmly believe that this can be readily achieved, as our approach is already general enough to encompass a wider range of functionality-preserving manipulations, including those discussed in Sect. III-A. Extending our work to deal with manipulations that can also modify the dynamic execution of malware programs, such as altering their control flow while preserving their malicious intent, is definitely challenging. Nevertheless, this would certainly provide an important step towards improving both the evaluation and the adversarial robustness of malware detectors trained on features extracted from dynamic program analysis.

REFERENCES

[1] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2015, pp. 11–20.
[2] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Proc. Australas. Joint Conf. Artif. Intell.* Cham, Switzerland: Springer, 2016, pp. 137–149.
[3] W. Hardy, L. Chen, S. Hou, Y. Ye, and X. Li, "DL4MD: A deep learning framework for intelligent malware detection," in *Proc. Int. Conf. Data Mining (DMIN)*, 2016, pp. 61–67.
[4] O. E. David and N. S. Netanyahu, "DeepSign: Deep learning for automatic malware signature generation and classification," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2015, pp. 1–8.
[5] Ì. Ì. Romeo, M. Theodorides, S. Afroz, and D. Wagner, "Adversarially robust malware detection using monotonic classification," in *Proc. 4th ACM Int. Workshop Secur. Privacy Anal.*, Mar. 2018, pp. 54–63.
[6] H. S. Anderson and P. Roth, "EMBER: An open dataset for training static PE malware machine learning models," 2018, *arXiv:1804.04637*. [Online]. Available: http://arxiv.org/abs/1804.04637
[7] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole EXE," in *Proc. 32nd AAAI Workshops*, New Orleans, LA, USA, Feb. 2018, pp. 268–276.
[8] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *Proc. 4th AISec*, 2011, pp. 43–58.

[9] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," *Pattern Recognit.*, vol. 84, pp. 317–331, Dec. 2018.

[10] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," in *Proc. 3rd Italian Conf. Cyber Secur. (ITASEC)*, 2019, pp. 1–13.

[11] B. Kolosnjaji *et al.*, "Adversarial malware binaries: Evading deep learning for malware detection in executables," in *Proc. 26th Eur. Signal Process. Conf. (EUSIPCO)*, Sep. 2018, pp. 533–537.

[12] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Adversarial examples on discrete sequences for beating whole-binary malware detection," 2018, *arXiv:1802.04528v1*. [Online]. Available: https://arxiv.org/abs/1802.04528v1

[13] R. L. Castro, C. Schmitt, and G. D. Rodosek, "ARMED: How automatic malware modifications can evade static detection?" in *Proc. 5th Int. Conf. Inf. Manage. (ICIM)*, Mar. 2019, pp. 20–27.

[14] R. L. Castro, C. Schmitt, and G. D. Rodosek, "Aimed: Evolving malware with genetic programming to evade detection," in *Proc. 18th Int. Conf. TrustCom*, 2019, pp. 240–247.

[15] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading machine learning malware detection," in *Proc. BlackHat*, 2017.

[16] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici, "Generic black-box end-to-end attack against state of the art API call based malware classifiers," in *Proc. RAID*. Cham, Switzerland: Springer, 2018, pp. 490–510.

[17] W. Hu and Y. Tan, "Generating adversarial malware examples for black-box attacks based on GAN," 2017, *arXiv:1702.05983*. [Online]. Available: http://arxiv.org/abs/1702.05983

[18] L. Tong, B. Li, C. Hajaj, C. Xiao, N. Zhang, and Y. Vorobeychik, "Improving robustness of ML classifiers against realizable evasion attacks using conserved features," in *Proc. USENIX Secur.*, 2019, pp. 285–302.

[19] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers," in *Proc. NDSS*, 2016, pp. 21–24.

[20] G. Ke *et al.*, "LightGBM: A highly efficient gradient boosting decision tree," in *Proc. NIPS*, 2017, pp. 3146–3154.

[21] J. Moody, "Fast learning in multi-resolution hierarchies," in *Proc. NIPS*, 1989, pp. 29–39.

[22] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, and F. Roli, "Adversarial EXEmples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection," 2020, *arXiv:2008.07125*. [Online]. Available: http://arxiv.org/abs/2008.07125

[23] O. Suciu, S. E. Coull, and J. Johns, "Exploring adversarial examples in malware detection," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2019, pp. 8–14.

[24] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, "From hack to elaborate technique—A survey on binary rewriting," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 1–37, Jul. 2019.

[25] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. NIPS*, 2019, pp. 8026–8037.

[26] F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *J. Mach. Lang. Res.*, vol. 13, pp. 2171–2175, Jul. 2012.

[27] I. Goodfellow *et al.*, "Generative adversarial nets," in *Proc. NIPS*, 2014, pp. 2672–2680.

[28] N. Rndic and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 197–211.

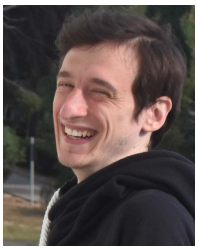
[29] C. Smutz and A. Stavrou, "Malicious PDF detection using metadata and structural features," in *Proc. 28th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2012, pp. 239–248.

[30] F. Ceschin, M. Botacin, H. M. Gomes, L. Oliveira, and A. Grégio, "Shallow security: On the creation of adversarial variants to evade machine learning-based malware detectors," in *Proc. 3rd Reversing Offensive-Oriented Trends Symp.*, 2019, pp. 1–9.

[31] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware dynamic analysis evasion techniques: A survey," *ACM Comput. Surv.*, vol. 52, no. 6, pp. 1–28, Jan. 2020.

[32] H. Aghakhani *et al.*, "When malware is Packin' heat; limits of machine learning classifiers based on static analysis features," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–20.

[33] L. Demetrio and B. Biggio, "Secml-malware: A Python library for adversarial robustness evaluation of windows malware classifiers," 2021, *arXiv:2104.12848*. [Online]. Available: https://arxiv.org/abs/2104.12848 and https://github.com/zangobot/secml_malware



Luca Demetrio received the M.Sc. (Hons.) and Ph.D. degrees in computer science from the University of Genoa, Italy, in 2017 and 2021, respectively. He is currently studying the weaknesses of threat detectors implemented with machine learning techniques and how to exploit such vulnerabilities. His research interests cover the area of adversarial machine learning, with strong focus on its application in the cyber-security domain.

Battista Biggio (Senior Member, IEEE) received the M.Sc. degree (Hons.) in electronic engineering and the Ph.D. degree in electronic engineering and computer science from the University of Cagliari, Italy, in 2006 and 2010, respectively. Since 2007, he has been with the Department of Electrical and Electronic Engineering, University of Cagliari, where he is currently an Assistant Professor. In 2011, he visited the University of Tuebingen, Germany, and worked on the security of machine learning to training data poisoning. His research interests include secure machine learning, multiple classifier systems, kernel methods, biometrics, and computer security. He is a member of IAPR. He also serves as a reviewer for several international conferences and journals.

Giovanni Lagorio received the M.Sc. (Hons.) and Ph.D. degrees in computer science from the University of Genoa, Italy, in 1999 and 2004, respectively. In 2000, he joined DIBRIS, where he started working on the design of object-oriented languages and systems. He is currently an Assistant Professor with the University of Genoa. His research interests have shifted more towards the security aspects of programs and systems, binary analysis and exploitation, adversarial machine learning, and ethical hacking.

Fabio Roli (Fellow, IEEE) received the Ph.D. degree in electronic engineering from the University of Genoa, Italy. He was a Research Group Member with the University of Genoa from 1988 to 1994 and an Adjunct Professor with the University of Trento from 1993 to 1994. In 1995, he joined the Department of Electrical and Electronic Engineering, University of Cagliari, where he is currently a Professor of computer engineering and the Head of the research laboratory on pattern recognition and applications. He was a very active organizer of international conferences and workshops and established the popular workshop series on multiple classifier systems. His research activity is focused on the design of pattern recognition systems and their applications. He is a Fellow of IAPR.

Alessandro Armando received the Ph.D. degree in computer engineering from the University of Genoa. In 2011, he founded (and led until 2016) the Security and Trust Research Unit, Bruno Kessler Foundation, Trento. His appointments include a position as a Research Fellow at The University of Edinburgh and INRIA-Lorraine, France. He is currently a Professor with the University of Genoa, where he teaches computer security and has founded and coordinated the master in cybersecurity and data protection. He is also serving as the Vice Director of the CINI National Cybersecurity Laboratory. He has been a coordinator and the team leader in several national and EU research projects, including AVISPA, AVANTSSAR, SpaCIoS, and SECENTIS. He contributed to the discovery of an authentication flaw in the SAML 2.0 Web-browser SSO Profile and a serious man-in-the-middle attack on the SAML-based SSO for Google Apps.