Dipartimento di Informatica, Bioingegneria,
Robotica ed Ingegneria dei Sistemi

Formalizing evasion attacks against machine learning security detectors

by

Luca Demetrio

Theses Series **DIBRIS-TH-2020-XXXIII**

DIBRIS, Università di Genova
Via Opera Pia, 13 16145 Genova, Italy `http://www.dibris.unige.it/`

Università degli Studi di Genova

Dipartimento di Informatica, Bioingegneria,

Robotica ed Ingegneria dei Sistemi

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum

Formalizing evasion attacks against machine learning security detectors

by

Luca Demetrio

October, 2020

Dottorato di Ricerca in Informatica ed Ingegneria dei Sistemi
Indirizzo Informatica
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università degli Studi di Genova

DIBRIS, Univ. di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
http://www.dibris.unige.it/

Ph.D. Thesis in Computer Science and Systems Engineering
Computer Science Curriculum
(S.S.D. INF/01)

Submitted by Luca Demetrio
DIBRIS, Univ. di Genova
. . . .

Date of submission: October 2020

Title: Formalizing evasion attacks against
machine learning security detectors

Advisor: **Giovanni Lagorio**
Dipartimento di Informatica, Bioingegneria, Robotica ed Ingegneria dei Sistemi
Università di Genova

External Reviewers:

Annalisa Appice,
Professor at Università degli Studi di Bari
Lorenzo Cavallaro,
Professor at King's College London
Bruno Crispo,
Professor at Università degli Studi di Trento

Abstract

Recent work has shown that adversarial examples can bypass machine learning-based threat detectors relying on static analysis by applying minimal perturbations. To preserve malicious functionality, previous attacks either apply trivial manipulations (e.g. padding), potentially limiting their effectiveness, or require running computationally-demanding validation steps to discard adversarial variants that do not correctly execute in sandbox environments. While machine learning systems for detecting *SQL injections* have been proposed in the literature, no attacks have been tested against the proposed solutions to assess the effectiveness and robustness of these methods. In this thesis, we overcome these limitations by developing RAMEn, a unifying framework that (*i*) can express attacks for different domains, (*ii*) generalizes previous attacks against machine learning models, and (*iii*) uses functions that preserve the functionality of manipulated objects. We provide new attacks for both Windows malware and SQL injection detection scenarios by exploiting the format used for representing these objects. To show the efficacy of RAMEn, we provide experimental results of our strategies in both white-box and black-box settings. The white-box attacks against Windows malware detectors show that it takes only the 2% of the input size of the target to evade detection with ease. To further speed up the black-box attacks, we overcome the issues mentioned before 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, encoded in an algorithm called GAMMA. We also evaluate whether GAMMA transfers to other commercial antivirus solutions, and surprisingly find that it can evade many commercial antivirus engines. For evading SQLi detectors, we create WAF-A-MoLE, a mutational fuzzer that that exploits random mutations of the input samples, keeping alive only the most promising ones. WAF-A-MoLE is capable of defeating detectors built with different architectures by using the novel practical manipulations we have proposed. To facilitate reproducibility and future work, we open-source our framework and 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.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Notation

Math symbols notation

$\mathcal{Z}$: input space
$\boldsymbol{z}$: sample inside the input space
$\boldsymbol{z}_i$: i-th element of vector or string
$\boldsymbol{z}^{(i)}$: i-th row of matrix $\mathbf{Z}$ or i-th version of $\boldsymbol{z}$
$\mathcal{X}$: feature space
$\boldsymbol{x}$: sample inside the feature space
$\mathbf{A}$: (capital letter) matrix
$\mathbf{A}_{i,j}$: entry of the matrix located at row i and column j
$\mathbf{A}^T$: Transpose of a matrix
∇ : gradient
$\nabla_{\boldsymbol{x}} f$: gradient of function f w.r.t. $\boldsymbol{x}$
$\phi(\cdot)$: feature extraction function
$\| \cdot \|_p$: p-norm of the argument
$|\boldsymbol{z}|$: length of the vector or string $\boldsymbol{z}$
$\lceil \cdot \rceil$: integer ceil function
$\lceil \cdot \rfloor$: integer round function

Publications

This thesis work is based on the following papers:

- "*Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries*", Demetrio Luca, Biggio Battista, Lagorio Giovanni, Roli Fabio and Armando Alessandro. Proceedings of the Third Italian Conference on Cyber Security (ITASEC), 2019.
- "*WAF-A-MoLE: Evading Web Application Firewalls through Adversarial Machine Learning*", Demetrio Luca, Valenza Andrea, Costa Gabriele, Lagorio Giovanni. Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC) 2020.
- "*Functionality-preserving Black-box Optimization of Adversarial Windows Malware*". Demetrio Luca, Biggio Battista, Lagorio Giovanni, Roli Fabio and Armando Alessandro. Submitted to IEEE Transactions of Forensics and Security (TIFS), under review.
- "*Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection*", Demetrio Luca, Coull Scott E., Biggio Battista, Lagorio Giovanni, Armando Alessandro, Roli Fabio. Submitted to ACM Transactions of Privacy and Security (TOPS), under review.

Chapter 1

Introduction

MACHINE learning techniques are 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, both Windows malware and SQL injections are significant threats in the wild, as witnessed by thousands of malicious programs uploaded to VirusTotal every day [1], and the OWASP top-10 document [2]. Modern approaches use machine learning to detect such threats at scale, leveraging many different learning algorithms and feature sets [3–11].

While these techniques have shown promising threat-detection capabilities, they were not originally designed to deal with non-stationary, *adversarial* problems in which attackers can manipulate the input data to evade detection. The weakness of such approaches has been widely shown in the last decade, in the active area of *adversarial machine learning* [12, 13]. This research field studies the security aspects of machine learning algorithms under attacks, either staged at training or test time. In particular, in the context of learning-based threat detectors, it has been shown that it is possible to optimize both *adversarial malware* and *adversarial SQL payloads* carefully, against a target system to bypass it [14–24]. Many of these attacks take place in black-box settings, where attackers have only query access to the target model [18–21]. These attacks question the security of such systems, especially 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.

Unlike adversarial evasion attacks in other areas, such as image classification, manipulating byte sequences corresponding to executable code, while simultaneously preserving their behavior, is more challenging. In particular, each perturbation may lead to changes in semantics, or the corruption of the underlying syntax/format, preventing the code from executing altogether or achieving its intended goal. To address this problem, attackers can

either choose to:

- apply invasive perturbations and use dynamic analysis methods, for instance emulation, to ensure that functionality of the binary is not compromised [25, 26], or
- focus their perturbation on areas of the files that do not impact functionality [14, 15, 23, 27, 28]; e.g. by appending bytes or injecting new sections.

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 Section 1.1 we discuss the limitations of the state of the art. Then, in Section 1.2, we highlight the reasons that sprung us to work on new techniques and a unifying formalization. Finally, in 1.3 we summarize our contributions by outlining the structure of this thesis.

1.1 Limitations of the state of the art

A formalization for generating adversarial examples has been proposed by Pierazzi et al. [29]. They construct a strategy as a minimization problem, by imposing different kind of constraints that shape the adversarial examples in terms of detectability, robustness to preprocessing, possible side-effects, and more. Such a formalization, while timely and effective, presents some issues. In particular, the authors impose a *plausability* constraint that should test if an adversarial sample would be or not spotted by a domain expert. While this constraint can be included by design in many domains, like images or audios where the attack has bounds, it is not clear how to port the same concept to security objects, and how to include it inside the optimization process. Moreover, their framework proposes the use an "oracle" that allows attackers to test the functionality of their adversarial examples. This implies that, at each iteration, the sample must be validated. While this is easy to achieve in some domains, for instance, in the image or audio domains, where perturbations are bounded to a certain level of additive noise, it can be troublesome in others. E.g., in the malware domain this step is equivalent to observe the behavior of adversarial examples inside a sandbox at each iteration of the optimization. The usage of a sandbox is shared by different strategies proposed in the state of the art, both white-box and black-box [17, 25, 26, 30]. Not only this step is time-consuming and resource hungry, but it might be skipped entirely with the use of *practical* manipulations, that is, manipulations that are semantics-invariant *by design*.

Many attacks rely on *padding* [15, 16, 19, 26, 28] to generate adversarial malware, whose application is trivial, and it is significant only if the payload consists of thousands of bytes. Other attacks also slightly alter the structure of the file [19] using a Python library

called *LIEF* [31] by adding sections and imports to the sample. Furthermore, the payload injected with such transformations is not optimized, and no strategy in the state of the art apply a constraint on the resulting file size. So, the adversarial payload might grow unbounded, and it could be easily detected due to its giant file size. From an adversarial machine learning perspective, the trade-off between detection rate and file size is essential, since one of the goals of these studies is showing how little noise does it take to disrupt the performance and preciseness of a victim classifier. It is indeed useful to state if an attack is successful without considering any other metrics, e.g. the size of the perturbation, but this would not help to understand the problem in its complexity. By finding such minimal patterns, researchers can foretell not only possible attacking strategies but also propose new defence mechanisms to protect such systems.

1.2 Motivations

As described above, applying manipulations to binary programs is more challenging than altering, say, images, since even altering a single bit inside a program can corrupt its file format, preventing the execution altogether, or entirely change the semantics.

With this background in mind, we started digging the details of the PE file format, and we noticed that it contains many different places that can host an adversarial payload. Once we found locations where adversaries can write, without disrupting the executables, we applied a simple strategy to decrease the confidence attributed from the victim network [14].

Since we alter bytes, we report the result of the attack, in Figure 1.1, by showing how much the modification of each byte alters the score. Each dashed red line represents a single iteration of our algorithm, that is concluded after having modified the maximum number of bytes (in this case, 58, as detailed in Section 3.2.1.1, page 28). The x-axis specifies the total number of modification that are applied to the bytes inside the editable region. So, after three iterations, the algorithms starts to consistently decrease the probability of being recognized as malware, and during the fourth iteration the score is dropped under the 0.1% probability.

We also have tried to understand why this target network behaves in this weird way, by applying a technique used for explaining the output of machine learning algorithms, called *integrated gradients* [32]. Without diving inside all mathematical details, this technique use the gradient to estimate how much a particular feature had impacted the overall score produced by the network. The results are shown in Figure 1.2. The blue color is used for highlighting a contribution towards the benign class, while the red color is used for the malicious class. Even though the DOS header does not contain any useful code or meaningful information regarding the nature of a program, the network is attributing false contribution to such locations. This results has two important take-away points: (i) this

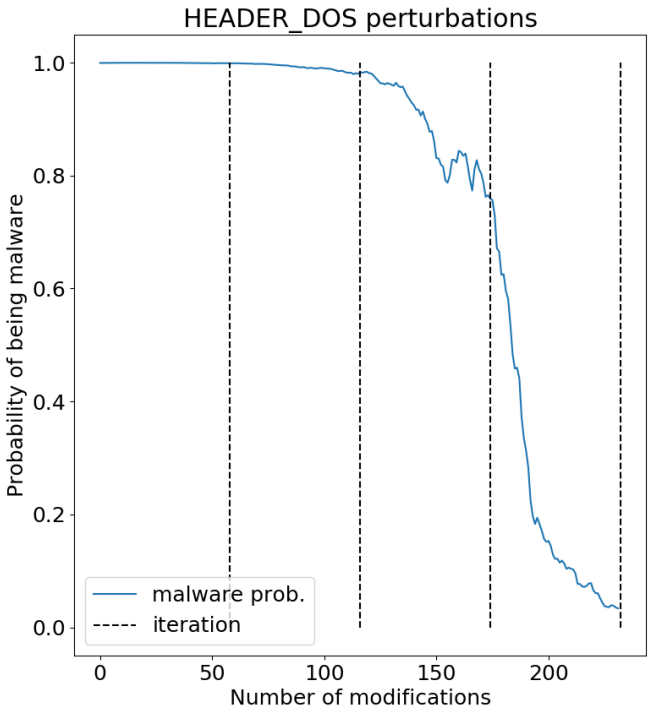


Figure 1.1: The attributed confidence to a input malware, by optimizing only 58 bytes inside the DOS header.

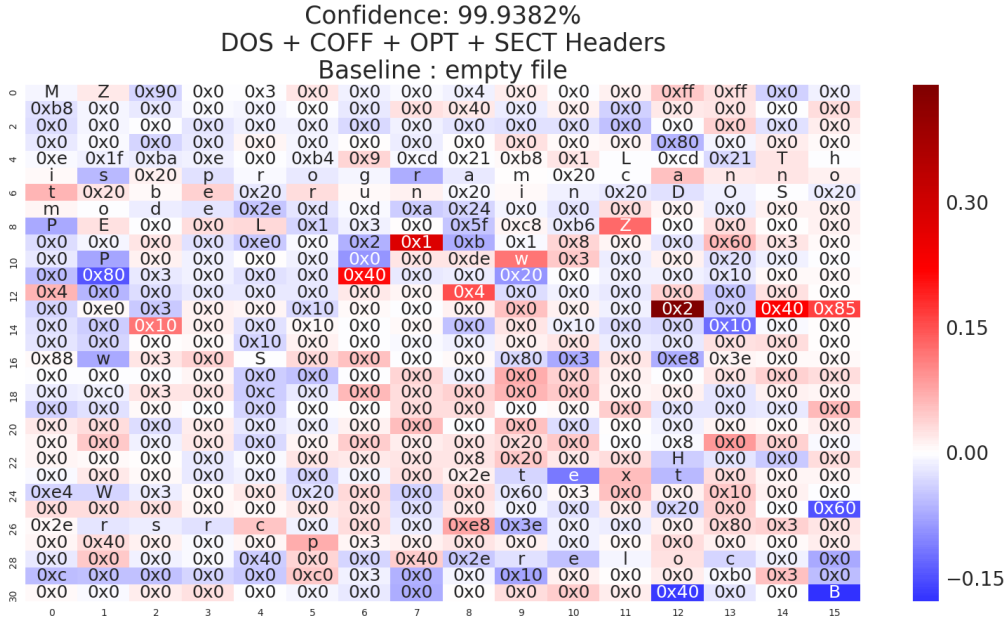


Figure 1.2: The feature importances attributed by MalConv to the initial part of the header of a sample program.

perturbation is infinitesimally small w.r.t. the whole file size, comparable to the invisible perturbation that is applied to images, and (ii) the semantics of the original sample is preserved.

Motivated by the results of this preliminary work, we started investigating approaches that would help attackers gain the upper hand in these scenarios.

1.3 Outline of this thesis

The next chapter, Chapter 2, gives the reader some background information, by briefly discussing the central intuition behind machine learning and its use in threat detection, and sets the stage for the following chapters, followed by a discussion of the techniques already proposed in the state of the art.

Chapter 3 presents our new lightweight formalization, called RAMEn (Realizable Adversarial Malware Examples), that can encode multiple evasion attacks without loss of generality. Our framework leverages practical manipulations, i.e. perturbations that do not impact the original functionality of input samples. The usage of such manipulations avoid the imposition of further constraints, like the one discussed in Section 1.1, and they can be

tuned during the optimization process. Since the application of such manipulations has a cost, RAMEn includes the usage of a loss function of choice, allowing attackers to choose how to penalize the injected noise. We showcase the expressiveness of RAMEn by creating adversarial examples against Windows antivirus programs and SQL injection detectors.

We test the efficacy of these transformations in Chapter 4, considering both white-box and black-box attacks against the detectors under analysis. For the white-box attacks, we highlight that attackers only need, on average, a 2% of the window length of the target end-to-end detector to compute fully functional and evasive adversarial malware. These attacks also highlight how detectors trained with different dataset size and architecture are still vulnerable to minimal perturbations. For the black-box attack, we develop two different optimizers: a genetic algorithm and a mutational fuzzer. The first one is used (*i*) inside GAMMA, that is, an algorithm that generates adversarial examples by injecting content harvested from benign software inside the input malware, and (*ii*) for optimizing each byte singularly inside the adversarial payload. The mutational fuzzer is used for creating adversarial SQL injections, by randomly mutating them and keeping the best one for the next round of mutations.

We conclude by explaining our final remarks and possibile future work in Chapter 5.

Chapter 2

Background

NOWADAYS, we are assisting to the necessity of analyzing massive corpora of data in less time, since systems are scaling up faster than before. For instance, in cybersecurity, there is a need for filtering, detecting and dissecting new threats and techniques that circulate on the Internet. These activities require instruments to help human analysts to recognize malicious patterns into data. This is why, in the last few years, the broad area of *machine learning* is becoming so relevant also in the security domain.

In this chapter we introduce the mechanics behind machine learning (Section 2.1, page 7). Then we describe the application of machine learning in detecting threats (Section 2.2, page 9). Finally, we describe the limits and dangers of using such a technology in adversarial contexts (Section 2.3, page 14).

2.1 Machine Learning

Machine learning is a research field, across computer science and statistics, that studies algorithms that learn from past observations, to foretell future outcomes. Such past knowledge is expressed as a matrix of vectors representing each observation of the phenomenon under study. In *supervised learning*, each observation is associated with a value. Such a value is the output of the unknown input-output relation that ties it to the observation. By stacking multiple observations, it is possible to compute a function that approximates such a relation. Creating a machine learning algorithm is indeed useful when the function to be learnt relies on patterns inside the data that are difficult to spot. For instance, creating a function that recognizes faces in photos can be done by telling the algorithm where the faces are, in each input picture. Then, after the training phase, the function's parameters are computed, and the approximate detector can be tested on unseen data.

Let be $\mathcal{X} \subseteq \mathbb{R}^d$ the set containing the observations in the form of a generic d-dimensional vectors, and let be $\mathcal{Y}$ the set containing the values of the output. Each value $y \in \mathcal{Y}$ can be a single real number in the case of *regression problems*, it can be either 0 or 1 in *binary classification problems*, or even a vector for *multi-classification problems* where the goal is to discriminate between multiple decisions, like animals or letters. By gathering all observations and their outputs, we form the *training set*, that can be expressed as a set of pairs $(\boldsymbol{x}^{(i)}, y^{(i)})_{i=1}^{N}$, where N is the total number of gathered samples. The aim of the learning process is finding an approximation of the real function that given an observation $\boldsymbol{x}^{(i)}$ returns $y^{(i)}$. If we would possess all infinite points that define such a function, we would be able to compute it. Since it is impossible to collect all these points, we need to rely on a finite set of observations that are representative enough for the function we want to learn. A way for producing such approximation function is minimizing the error scored by the candidate approximation on the input samples. This can be done by introducing a distance function that quantifies the error committed by the approximation w.r.t. the real output, the *loss function* $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$. Usually, the loss function is a differentiable function, like the square loss $L(f(\boldsymbol{x}), y) = (f(\boldsymbol{x}) - y)^2$, or the logistic loss $L(f(\boldsymbol{x}), y) = -ylog(f(\boldsymbol{x})) - (1 - y)log(f(\boldsymbol{x}))$. Now that we have introduced a notion of distance between the approximation and the real values of the function, given observations inside a d-dimensional space expressed as a matrix $\mathcal{X} = \mathbb{R}^d, \boldsymbol{X} \in \mathbb{R}^{N \times d}$, and their output expressed as a vector $\boldsymbol{y} = (y^{(0)}, ..., y^{(N)})$, we can compute the minimization as:

$$\min_{f \in \mathcal{H}} \sum_{i=0}^{N-1} L(f(\boldsymbol{x}^{(i)}), \boldsymbol{y}_i) \tag{2.1}$$

where $\mathcal{H}$ is the space of functions we want to search (hyperplanes, polynomials, gaussians, etc.). By minimizing this quantity, the function f adapts itself to the past observations. This can be achieved by differentiating w.r.t. the parameters of f, and put the whole quantity to 0. For instance, if the function we want to learn is a hyperplane with coefficients $\boldsymbol{w} \in \mathbb{R}^d$, and the loss is the squared loss, we would have:

$$\nabla_{\boldsymbol{w}} \sum_{i=1}^{N} (\boldsymbol{w}^T \boldsymbol{x}^{(i)} - \boldsymbol{y}_i)^2 = 0 \tag{2.2}$$

The problem can be solved in closed form, and the best coefficient $\boldsymbol{w}^*$ are computed. These learning algorithms are useful for solving many difficult tasks, such as *object detection*, *face recognition*, *text to speech* problems and more. Also, they can serve as a fundamental tool for aiding security analysts in their job. For this reason, studying how to carve information from objects related to cybersecurity is the first step for training clever classifiers that stop threats in the wild.

2.2 Detecting threats with Machine Learning

With the aid of statistics and pattern recognition, malware can be fought before it causes harm. Machine learning algorithms can be trained with large corpora of data to create models capable of understanding what is malicious and what is not. Since these techniques find patterns in data, they might represent a key for understanding how threats evolve and how to spot them without risking to infect millions of devices. In addition, these tools are indeed helpful in filtering results or recognizing variants of the same threat without wasting hours of human labour. There are already many companies that apply such techniques, including statistical algorithm inside their products [33–36]. In this thesis, we consider detectors that are trained to recognize two different families of threats: Windows malware and SQL injections. Since the goal of this thesis is to show the effectiveness of our formalization (described in detail in Chapter 3), the choice of these two not-so-overlapping families of threats highlights how much our work can express different domains, and we use them as a showcase of the generality of our approach. Nevertheless, other threats can be included in the future too, like Android malware applications, or traffic data analyzers.

2.2.1 Windows malware

Threat detectors, that leverage static analysis, work on the on-disk representation of Windows programs. These are just a sequence of bytes following the *Windows Portable Executable* (PE) [37]. This format is interpreted by the operating system, that takes care of creating a process and loading various parts of the program at different virtual addresses. Figure 2.1 shows various components of a program file:

DOS Header and Stub contain metadata for loading the executable inside a DOS environment, and a few instructions that print "*This program cannot be run in DOS mode*". These two components have been kept to maintain compatibility with older Microsoft's 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, 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 cannot be executed.

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 is not really *optional*, since it contains the information needed by the operating system for loading the program. It also contains offsets that point to useful

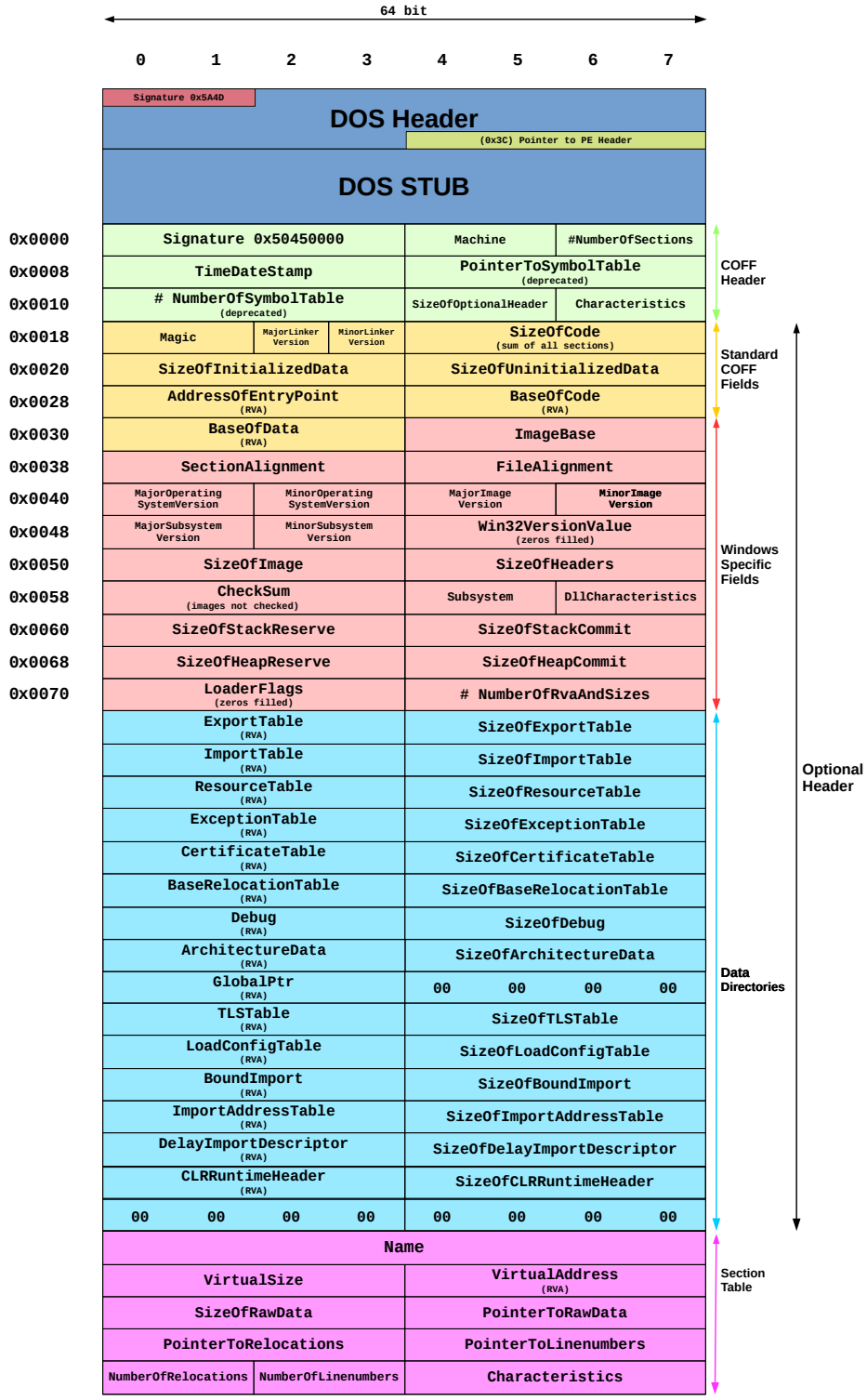


Figure 2.1: The Windows PE file format.

structures, like the *Import Table* needed for resolving dependencies, and the *Export Table* to make functions accessible 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.

All programs that run natively on Windows are stored on disk by following this format, *malware* included. This kind of software is created to accomplish not-ethical, or even criminal, purposes, including stealing or destroying data, opening hidden accesses, farming cryptocurrencies and more. For instance, *ransomware* encrypts all content of the victim hard-drive, and then asks the user to pay for restoring their files back.

Two recent byte-based convolutional neural network models for malware detection use an embedding layer to project the bytes in PE format into a higher-dimensional space. Then, they apply one or more convolutional layers to learn relevant features that are fed to a fully-connected layer for classification with a sigmoid function. 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.

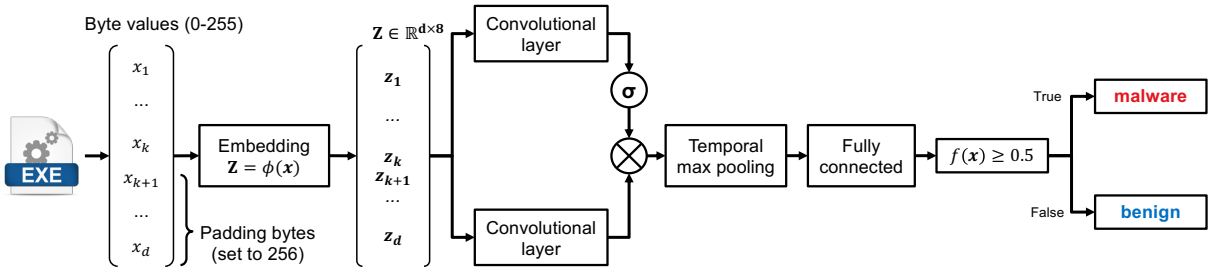


Figure 2.2: The architecture of MalConv.

MalConv: Proposed by Raff et al. [9], *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. The results of these operations are passed as input to a fully-connected layer for classification. The architecture is depicted in Figure 2.2.

DNN with Linear (DNN-Lin) and ReLU (DNN-ReLU) activations: Jeffrey Johns [38] and Coull et al. [39] 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. These are 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. Like MalConv, files exceeding this length are truncated and shorter files are padded with a distinguished padding token. Several variations of this architecture are evaluated in Section 4 (on page 46), 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. [39] 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): Anderson et al. [8] train a GBDT [40] on top of their open-source dataset called *EMBER* [8]. 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 to compare the two above byte-based models against each other, and help demonstrating the gap between the features learned by byte-based neural networks and those created by subject-matter experts.

2.2.2 SQL Injections

Structured Query Language (SQL) is a language for interacting with database servers. That is, the SQL standard describes commands that allow users to insert, update and delete information from databases. An example of SQL query is shown in Listing 2.1. In this example, we want to extract all names, surnames and birthdays of all users whose role is *admin*.

```
SELECT name, surname, birthday
FROM users
WHERE role = "admin"
```

Listing 2.1: An example of SQL query

When SQL queries are hazardously built by concatenating user inputs and literal strings, an attacker can inject commands inside the resulting query. An example of a such bad code is depicted in Listing 2.2.

```
$query = 'SELECT name, surname, birthday FROM USERS WHERE role = "admin" ' .
         'AND pwd = "' . $_POST['pwd'] . '"';
$result = query($connection, $query);
```

Listing 2.2: An example of buggy PHP code, containing a SQL injection vulnerability

These two PHP statements should construct a SQL query to search for admin users whose password match the user input `$_POST['pwd']` (that is, a specific POST parameter). One problem resides in how the SQL query is constructed: the input value is directly used inside the command, without any sort of escaping/sanification (the other, *huge*, problem is storing passwords in cleartext, but that is out of scope for this discussion). Hence, by using an ad-hoc input, like `1" OR 1=1 #`, as value for `$_POST['pwd']`, the SQL query will return private data of every user, since the right side of the `OR` is always true.

These kinds of attacks are the so-called *SQL injections*, and even if there are trivial ways for avoiding them, they still appear in the wild [2]. One simple way for avoiding SQL injections is to sanitize every client input, for instance, by employing *prepared statements*. Inside a prepared statement, wildcard symbols let the engine know where parameters are. Such placeholders are then replaced with automatically escaped values. An example is shown in Listing 2.3. The `bind_param` function replaces the symbol `?` with the value passed as second argument (that is, `$_POST['pwd']`), whose type is specified by the first argument (in this case, `s` corresponds to the string type).

```
$stmt = $conn->prepare('SELECT name, surname, birthday FROM USERS
                        WHERE role = "admin" AND pwd = ?');
$stmt->bind_param("s", $_POST['pwd']);
$stmt->execute();
```

Listing 2.3: An example PHP code that uses prepared statement

To mitigate this kind of vulnerability it is possible to interpose a program, which can check whether a query has been injected, and stops the malicious ones from causing harm when passed to the DBMS. These tools are called *Web Application Firewalls*. Usually, these tools work on top of deny-lists of malicious payloads that lead to data stealing, manipulation or else. Since these lists are static, the attacker can slightly change the syntax for producing a new unknown injection that evades the WAF. To gain advantage of such dynamical environment, the use of machine learning algorithms can help in modeling functions that adapt to this domain.

WAF-Brain: available on GitHub, and developed by a private company, WAF-Brain [41] is a recurrent neural network that looks five continuous character in a string and it tries

to guess the sixth. Since it has been trained on SQL injections, it tries to guess if such substring is part or not of a malicious injection payload. For clarity, given the following string `OR '1` it might guess that, after the digit, there will be the missing `'` character that closes the string. Correctly predicting the following character means to have found a match with the learned malicious payloads. This process is repeated for every block of five characters forming the target query, and a final mean score is returned. The input layer is a Gated Recurrent Unit (GRU) [42] made of five neurons, followed by two fully-connected layers, i.e., a dropout layer followed by another fully connected layer. Finally, WAF-Brain computes the average of all prediction errors over the input query and scores it as malicious if the result is above a fixed threshold chosen a priori by the user. Since the threshold is not given by the classifier itself, as all other details of the training and cross-validation phases, we set it to 0.5, which is the standard threshold for classification tasks.

Token-based Classifiers: Token-based classifiers represent input queries as histograms of symbols, namely tokens. A token is a portion of the input that corresponds to some syntactic group, e.g., a keyword, comparison operators or literal values. We took inspiration from the review written by Komiya et al. [43] and Joshi et al [11] and we developed a tokenizer for producing the features vector to be used by these models. On top of that, we implemented different models: (*i*) a Naive Bayes (NB) classifier, (*ii*) a random forest (RF) classifier with an ensemble of 25 trees, (*iii*) a linear SVM (L-SVM), and a gaussian SVM (G-SVM).

Graph-based Classifiers: Kar et al. [10] developed SQLiGoT, an SQL injection detector that represents a SQL query as a graph, both directed and undirected. Each node in this graph is a token of the SQL language, plus all system reserved and user defined table names, variables, procedures, views and column names. Moreover, the edges are weighted uniformly or proportionally to the distances in terms of adjacency. The model is based again on SVMs, and Kar et al. released the hyper-parameter they found on their dataset, but since both C and γ depend on data, we had to train these models from scratch.

2.3 Adversarial Machine Learning

Since learning algorithms might be used in critical contexts, there is the necessity of studying the limits and weaknesses of such detectors. Let us imagine a program whose aim is recognizing the face of its owner, like the newest face detectors built in modern smartphones. Once verified, the smartphone operating system unlocks the phone, allowing the user a total control over the device. In this very common and real-life scenario, there is the need of a error-proof algorithm, since the user does not want their device to be accessed by unknown strangers in the wild. The field of *Adversarial Machine Learning* [12, 44] formalizes the presence of an attacker that wants to take advantages of weaknesses of learning

algorithms to reach a particular accomplishment. We can divide these goals in five main categories: *poisoning*, *backdoors*, *model stealing*, *model inversion*, *membership*, and *evasion* attack. The latter one will be deeply discussed, since all work contained in this thesis are extensions of such strategy.

Poisoning Attacks: If the attacker can manipulate the training set of a particular model, they can inject data that alter the decision boundary itself [13,45,46]. For instance, an *online learning algorithm* updates its knowledge w.r.t. the new input that it has to classify, like *anomaly detectors*. These classifiers do not retain all past knowledge, since they need to adapt to the future data that will be sent to the detector. Each time a sample is misclassified, it is added inside the training set, while the oldest point is removed. Lastly, the model is re-trained on this new set of points. Hence, the attacker may send samples that lies near the decision boundary of the target, shifting it bit by bit. The ultimate goal is the degradation of the target model, by spoiling its original functionality, by also including the evasion of the system itself.

Backdoors Attacks: instead of messing with the decision boundary of the target model, the attacker wants to implant a particular behavior inside the victim model, without degrading the whole classifier [47–49]. This is achieved by attaching a pattern to the input samples, and using poisoning techniques modify the decision boundary to be compliant to the attacker's need. An example is given by image detectors, whose backdoors are small pattern of pixel inserted inside the image. The presence of such pattern is enough for piloting the decision towards the desired class or score.

Model Stealing Attacks: training a classifier requires time and resources: depending from the chosen architecture and the volume of data, the training phase might last days or weeks, with extensive usage of expensive devices and GPUs. Just think about the translation model offered by Google: it cannot be trained on standard machines, but it would be indeed useful to be used locally. Also, by using this local model the users would avoid paying for such a service. Under some preconditions, the attacker can compute a replica of a remote detector, by only using the results of the queries they send to it [50].

Model Inversion and Membership Attacks: depending on the task, machine learning models might infer knowledge from sensitive data, like medical records [51, 52]. Also, the presence of a particular point inside the corpora of data used at training time could be another potential privacy leak, such as which faces have been used for training a face detector, or else. In this scenario, the attacker wants to acquire such information, by either reconstructing values of interests (i.e. medical records) or acquiring knowledge of what has been used for training.

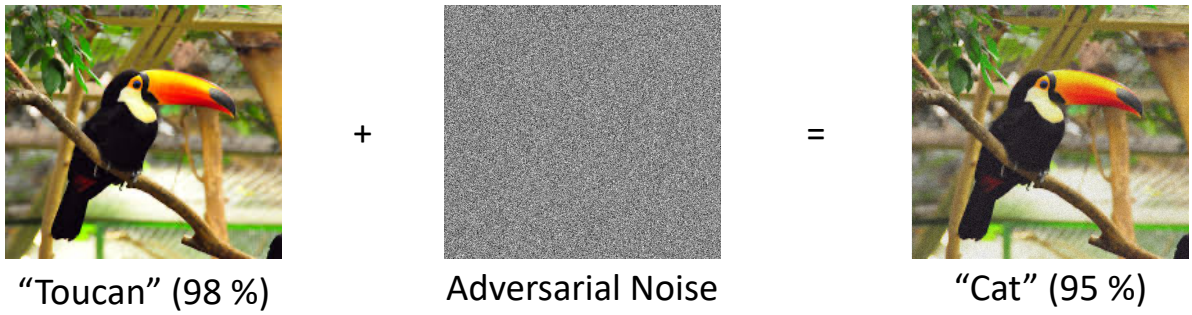


Figure 2.3: An example of evasion attack.

2.3.1 Evasion Attacks

The attacker wants to create samples that are misclassified into a specific or generic class, different from the real one. As an example, an image classifier that is fooled to believe that a picture of a toucan is in fact a cat [13, 53–58], as shown by Figure 2.3, or creating vocal commands that are interpreted erroneously by the victim model [59–61]. If human beings would stare at such pictures or listen to these sounds, they will not complain about the presence of such noise, since they will not notice it. The problem of evasion attacks arise from the domain itself. For instance, perturbing the color of a few pixels by small amount introduce a pattern inside the original image that it is very hard to spot. This is also true for black and white images, with very low amount of noise. The semantics that we attribute to such image is the same: we see a picture, that is a matrix of pixels arranged in a particular way, and we match it with the most similar image we have in our mind: in this case, a toucan. A machine learning algorithm works in a different way: it takes in input the values of each pixel, it passes them through different matrix-multiplication operations, and it finally gives an answer to the query we asked. The ad-hoc adversarial noise, which has been carefully crafted, aims to alter the results of such operations to accomplish the goal of the attacker. The process of creating such noise can be expressed mathematically as an optimization problem:

$$\begin{aligned} \max_{\boldsymbol{\delta}} \quad & f_l(\boldsymbol{x}+\boldsymbol{\delta}) - f_k(\boldsymbol{x}) \\ & s.t. \parallel \boldsymbol{\delta} \parallel_p < \epsilon, l \neq k \end{aligned} \tag{2.3}$$

where $\boldsymbol{\delta}$ is the noise added to the image, f is the target model, f_l and f_k express the probability of belonging to a generic class l and the true class k respectively, and $\boldsymbol{x}$ is the image we want to perturb, k is the true class of such sample. By maximizing this quantity, we are asking to find a perturbation $\boldsymbol{\delta}$ that, applied to the sample $\boldsymbol{x}$, will maximize the difference between the probability of a generic class $l \neq k$ and the probability of the real class k. As a consequence, since it is a difference, the maximum must be a positive number,

and this can only be given by an higher score attributed to a different class. The strength of the attack is controlled by a parameter, ϵ, that serves as an upper-bound to the p-norm of the noise. Depending on the norm that is chosen for computing the attack, the computed noise has different properties. The most powerful norm to use should be the ℓ_0 norm, that counts how many pixels have been changed after the application of the noise. Bounding this quantity means computing an adversarial example with the fewest changes possible. The problem of such norm is posed by its non-differentiability, hence it complicates the way samples can be generated.

Another example is the usage of the ℓ_∞, that returns the maximum value of the input vector. This implies that, when bounded, the attacker can modify every pixel of the image up to the bounding value ϵ. The attack perturbs basically each pixel of the image, but the process is easier to deal with. The optimization problem expressed in Equation 2.3 is an *untargeted attack*, since every class that satisfies the constraint fits the goal of the attacker. If they want to produce an example that is classified as a specific class, the attacker carries on a *targeted attack*, where we want to maximize the output of the chosen-a-priori class l, different from the real class k. In this way, the output of the l-th class will be higher that the score attributed for class k, successfully achieving the evasion with the desired class, without changing the problem formalized in Equation 2.3.

2.3.2 Attacker Knowledge

Once the attacker has chosen the goal to pursue, they also need to devise an effective strategy, depending on the knowledge they possess regarding the victim.

2.3.2.1 White-box attacks

The worst case scenario is posed by *white-box* attacks, where the attacker knows everything about the target. They can compute adversarial examples by using fast and effective algorithms that rely on the gradient of the model function [52,53,58,62]. The gradient of a function ∇f is a direction that points to the maximum ascent of the function itself, when evaluated on a point. By following that direction, the attacker can navigate the input space to find the ones that suits the attacker's needs.

2.3.2.2 Black-box attacks

White-box strategies can be applied if and only if the model function is differentiable as well. This might not be true, and the attacker needs to switch to another approach that is similar to techniques that involves less knowledge of the target. In general, the

assumption of having perfect knowledge of the target is strong and sometimes difficult to achieve. Many commercial machine learning models are closed-source, and they can be only accessed by sending samples to them, and collecting a reply. In this scenario, the attacker does not know the parameters of the model, so they can only leverage *black-box* attacks. Since these models are closed-source, there is no certainty regarding which model have been used, which information have been extracted from data and so on. Here, the attacker can choose different paths, depending on the limited knowledge they have.

Surrogate Models: if the attacker knows a subset of the data that have been used for training the target, they can re-create an approximation of such detector [62]. Since the victim can be queried, the attacker labels their dataset with the answer of the remote classifier. Once they have all answers, the attacker can train a differentiable model with these data. With this surrogate model, the attacker can apply white-box attacks to evade their own classifiers, and then they try to send the same adversarial examples to the target. The intuition behind this attack is that, with similar data, models behave in a similar way. Hence, adversarial examples that evade the surrogate, might evade the target as well.

Transfer Attacks: since training a surrogate is time consuming, and its performances rely on the quality of the input data, the attacker can choose an open-source public model that solves the same problem, compute white-box attacks on it, and test the adversarial examples against the target [63–65]. Intuitively, a model that has been trained to compute a similar answer might be also vulnerable to the same attacks, as if these models share similar blind-spots, since they have been designed in a similar way.

Query Attacks: instead of constructing the whole strategy on top of other classifiers or surrogates, the attacker directly exploits the knowledge embedded inside the response of the query they send to the target [66, 67]. By keeping interrogating the target, the attacker infers knowledge of the shape of the remote decision function. Each locally crafted perturbation is evaluated by the remote detector, and the attacker keeps exploring the space of adversarial examples in this way.

While it is easy to perturb an image or a sound, the real challenge lays in the perturbation of more "fragile" objects. In particular, altering commands or programs is not trivial at all, since the applied perturbations might break the sample. However, it would be indeed useful to an attacker that wants to bypass security-related targets, such as antivirus programs and threat detectors.

2.4 Related Work

After having introduced the key elements that characterize this thesis, we continue with an overview of the state of the art that can shed some light over the research that is carried

on in such fields of study.

Similar formalization: Pierazzi et al. [29] 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 both imperceptible to manual inspection and resilient to pre-processing techniques. The authors also make explicit 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 constraints mentioned above by exploring the space of mutations, imposed by such practical manipulations.

White-box adversarial attacks against malware detectors: Kolosnjaji et al. [15] apply a gradient-based attack against MalConv, by appending bytes to the overlay. Since MalConv is not fully differentiable, the attack takes place inside the embedding space. The gradient allows the attacker to append bytes that most reduce the confidence of the input malware. Experimental results show that the number of padding bytes needed to evade ranges from 2 KB to 10 KB. Demetrio et al. [14] fine-tune the attack proposed by Kolosnjaji et al. by generalizing the algorithm w.r.t. the location to apply the perturbations, and showing that the same network can be bypassed by manipulating only 58 bytes inside the DOS header of an executable.

Similarly, Kreuk et al. [16] applied the Fast Gradient Sign Method (FGSM) [68] to alter not only padding but also slack bytes, that is, bytes inserted between sections to maintain alignment. However, they show that slack locations are not enough for crafting an adversarial executable malware, and they need to include padding as well. The attack is formulated again in the embedding space, but the real bytes are only computed after the algorithm has found an evading sample.

Rosenberg et al. [20] attack a Recurrent Neural Network (RNN) using a black-box strategy [62] by reconstructing the target classifier under attack. They fool the proposed RNN by injecting fake API calls at run-time, wrapping the input malware inside another program with the correct sequence of API that needs to be called for producing the sample. While the wrapper they developed is interesting for proposing a different way for mutating malware, neither code nor results have been publicly released by the authors.

Suciu et al. [28] explored a similar approach to Kreuk et al., by applying the FGSM, and compute adversarial payloads to be inserted between sections (if there is available space) and as padding. No code has been released yet for testing the attack.

Sharif et al. [69] apply random manipulations that replace 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, then the sample is kept for the next iteration, otherwise it is discarded. Unfortunately, no code has been released yet for testing their attack.

Black-box adversarial attacks against malware detectors: Castro et al. [17, 25] propose two black-box optimizers to compute adversarial attacks. The former [17] applies random mutations to the input sample, while the latter [25] uses a genetic algorithm for searching the best sequences of mutations to apply. Both techniques leverage a sandbox environment where, at each iteration, the adversarial malware are executed to ensure that their functionality is preserved.

Anderson et al. [19] 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. Results show that the learned policy perform slightly better than the random one. 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. The model they used as a baseline is a primordial version of the EMBER classifier we have analyzed in this work (see Chapter 4 for our experimental evaluation), trained on fewer samples.

Hu et al. [21] develop a Generative Adversarial Network (GAN) [70] 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. It is interesting that the algorithm ignores the semantics of all objects that considers and it can create points that evade the target classifier nonetheless. However, no real malware is crafted, as that attack only operates inside the feature space. The result of the GAN serves only as a trace for the attacker, to understand what should be changed inside its malware. In contrast, we create functioning malware, as real samples are generated each time.

White-box attacks against other security-related detectors: Grosse et al. [71] propose a white-box attack against DREBIN, an Android malware detector [72], by applying a Jacobian-based approach [55]. Using the gradient, the authors understand which features needs to be added inside the application, in terms of modification of meta-data of the program.

Chen et al. [73] propose a gradient attack against two Android malware detectors [72, 74], by introducing API calls and new pre-written code to dilute the contributions of other features toward the negative class.

Pierazzi et al. [29] proposes a gradient-based attack against Android malware detectors [75] that injects fractions of benign Android byte-code inside the malware program.

Black-box attacks against other security-related detectors: Xu et al. [30] propose

a fully black-box attack scheme that relies on a genetic algorithm to produce adversarial examples. They show that this approach is suitable for security application, as they successfully create functioning adversarial PDF malware that evade state-of-the-art PDF malware detectors. On the other hand, since this method ignores the internal technical details of the PDF format, the black-box optimization produces both functioning and also broken PDF malware that are lately discarded by the evolution. As shown by the authors, this is very time consuming, as the algorithm needs to learn which manipulations break the original semantics and which not.

Laskov et al. [76] show a black-box attack against PDFRate [77], which is a PDF malware detector. The authors simulate an adversary that have only partial information regarding the classifier under attack. They reconstruct the victim model and they perform gradient attacks against it, transferring them to the real one. The attack injects content inside the PDF malware, successfully fooling both the surrogate and the target.

White-box attacks against in other domains: Biggio et al. [53] propose an iterative gradient technique for computing adversarial examples against any kind of classifier. Since the attacker must comply with certain constraints, at each iteration of the algorithm, the partial results are projected inside the feasible region, that represents the domain of valid adversarial examples.

Goodfellow et al. [54] propose the Fast Gradient Sign Method (FGSM), originally formulated for computing adversarial examples against deep image classifiers. The technique requires the computation of the gradient of the network w.r.t. the input, and its magnitude is bounded to the amount of perturbation that can be added by the adversary. To this extent, the attacker only takes the sign of these values, multiplied by a parameter for tuning the efficacy of such noise.

Papernot et al. [55] propose the Jacobian-based Saliency Map Attack (JSMA), that modifies one pixel at the time, maximizing the error at each step. Each pixel of the image is weighted based on the importance computed from the gradient in that point, highlighting which one must be modified to decrease the confidence.

Carlini and Wagner [58] propose an iterative approach whose core is a reconstruction problem, with bounds imposed on the maximum amount of perturbations. They use of different norms, whose application leads to different formalizations and results, as norms have different effects on the adversarial perturbations.

Generic Black-box attacks in other domains: Papernot et al. [62] show that it is possible to evade an unknown classifier by reconstructing a local surrogate model. The attacker need to query the target classifier to construct a surrogate dataset that will be used for training the model. The adversary computes the attack against the trained local classifier and it tries to transfer the samples to the target as well. This is time consuming as the attacker needs to send many queries to the target system, as all samples inside the

surrogate dataset need to be labeled.

Chen et al. [78] propose the so-called Zeroth Order Optimization (ZOO), that estimate the gradient of the victim classifier by sampling points around the input. This strategy reduce the number of queries, as the search is local and only around a particular point.

Ilyas et al. [66] apply the Natural Evolution Strategy (NES) [79] to reduce the number of queries that are needed to compute the black-box attack, by imposing a distribution over the manipulation and use the ones that allow the algorithm to explore more.

Other malware machine-learning detectors: Saxe et al. [3] 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. [4] propose to track which APIs are called by a malware, capturing the execution trace using the Cuckoo sandbox,[1] that is a dynamic analysis virtual environment for testing malware.

Hardy et al. [5] 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. [7] 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. [80] propose a similar architecture as the one developed by Johns[2] and Coull et al. [39]: 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) [81] activation functions.

Other SQL Injections machine-learning detectors: Ceccato et al. [**?**] propose a clustering method for detecting SQL injection attacks against a victim service. The algorithm learns from the queries that are processed inside the web application under analysis, using an unsupervised one-class learning approach, namely K-medoids [82]. New samples are compared to the closest medoid and flagged as malicious if their edit distance w.r.t. the chosen medoid is higher than the diameter of the cluster.

Kar et al. [10] develop SQLiGoT, a support vector machine classifier (SVM) that expresses

[1] https://cuckoosandbox.org/

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

queries as graphs of tokens, whose edges represent the adjacency of SQL-tokens. This is the classifier we used in our analysis.

Pinzon et al. [83] explore two directions: visualization and detection, achieved by a multi-agent system called *idMAS-SQL*. To tackle the task of detecting SQL injection attacks, the authors set up two different classifiers, namely a Neural Network and an SVM.

Makiou et al. [84] developed an hybrid approach that uses both machine learning techniques and pattern matching against a known dataset of attacks. The learning algorithm used for detecting injections is a Naive Bayes [85]. They look for 45 different tokens inside the input query, chosen by domain experts.

Similarly, Joshi et al. [11] use a Naive Bayes classifier that, given a SQL query as input, extracts syntactic tokens using spaces as separator. The algorithm produces a feature vector that counts how many instances of a particular word occurs in the input query. The vocabulary of all the possible observable tokens is set a priori.

Komiya et al. [43] propose a survey of different machine learning algorithms for SQL injection attack detection.

Chapter 3

Formalization and Attacking Strategies

Attackers' goal is the creation of adversarial examples, by adding well-crafted noise to each sample and achieve evasion with high confidence. This chapter will introduce (i) the mathematical framework used to encode all attacks (Section 3.1, page 24), (ii) the mutations that are applied to the samples to produce adversarial examples (Section 3.2, page 27), (iii) and how to implement the framework for white-box (Section 3.3, page 32) and (iv) black-box (Section 3.4, page 34) attacks.

3.1 Formalization

Most classifiers process data through a *feature extraction phase* and the attack takes place inside such a feature space. Since machine code and SQL commands are both encoded as arbitrarily long strings of bytes, we define the set of all possible functioning samples as $\mathcal{Z} \subset [0, 255]^*$. We denote the prediction function with $f : \mathcal{X} \to \mathcal{Y}$, and the feature extractor with $\phi : \mathcal{Z} \to \mathcal{X}$, where $\mathcal{X}$ is the feature space, and $\mathcal{Y}$ is the output space of the model.

The attacker can apply *practical manipulations*, i.e., transformations that alter the representation of the input sample without disrupting the original functionality, by exploiting redundancies offered by the format. We encode these manipulations as a function $h : \mathcal{Z} \times \mathcal{T} \to \mathcal{Z}$ whose output is an object compliant to the format specifications and with the same behavior of the input one, but with a different representation. Function h takes in input a sample $\boldsymbol{z} \in \mathcal{Z}$ and a vector $\boldsymbol{t} \in \mathcal{T}$, that represents the parameters of the transformation. By optimizing the entries of this vector, attackers can tune the application

of practical manipulations on a particular sample.

We denote $\mathcal{H}_z = \{h(\boldsymbol{z}, \boldsymbol{t}), \boldsymbol{t} \in \mathcal{T}\}$ the set of objects, e.g. programs or SQL queries, that can be created by applying practical manipulations on $\boldsymbol{z}$, the original sample. To measure the distance from the benign class, we use a *loss function* $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. This function depends on the output of the prediction on a point $f(\phi(\boldsymbol{z}))$, where we want to compute the error, and the label y of the class the attacker wants to approach: in our case, y represents the benign class. The goal of the attacker is to minimize $L(f(\phi(h(\boldsymbol{z}, \boldsymbol{t}))), y)$. This amounts to diminishing the probability of the modified program being recognized as malware.

We present *RAMEn* (Realizable Adversarial Malware Examples), a general framework that reduces the problem of computing adversarial examples of malware to optimization problems of the form:

$$\min_{\boldsymbol{t} \in \mathcal{T}} \quad L(f(\phi(h(\boldsymbol{z}, \boldsymbol{t})), y) \tag{3.1}$$

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

Every function is differentiable: If all functions are differentiable, then the optimization can be carried out inside the space of the parameters $\mathcal{T}$ by using the gradient descent algorithm:

$$\boldsymbol{t}_{(i+1)} = \boldsymbol{t}_{(i)} + \gamma \frac{\partial L}{\partial \boldsymbol{t}} \tag{3.2}$$

where γ is the step-size of the gradient descent algorithm, that modulates how much the space is explored. A use-case for Equation 3.2 is the production of adversarial examples against end-to-end image detectors, where the target end-to-end model is differentiable, and the practical manipulations consist of differentiable functions, like rotations.

Non-differentiable manipulations: If all 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:

$$\boldsymbol{z}' = \boldsymbol{z} + \gamma \frac{\partial L}{\partial h} \tag{3.3}$$

$$\boldsymbol{t}_{(i+1)} = \arg\min_{\boldsymbol{t} \in \mathcal{T}} \parallel z' - h(\boldsymbol{z}, \boldsymbol{t}) \parallel^2 \tag{3.4}$$

Non-differentiable feature extraction: Feature extractors are often non-differentiable and/or not-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 3.1. Set $\mathcal{V}_z = \{\phi(z') : z' \in \mathcal{H}_z\}$, then the problem inside the feature space can be posed as:

$$\boldsymbol{x}^\star \in \arg\min_{\boldsymbol{x} \in \mathcal{V}_z} L(f(\boldsymbol{x}), y) \tag{3.5}$$

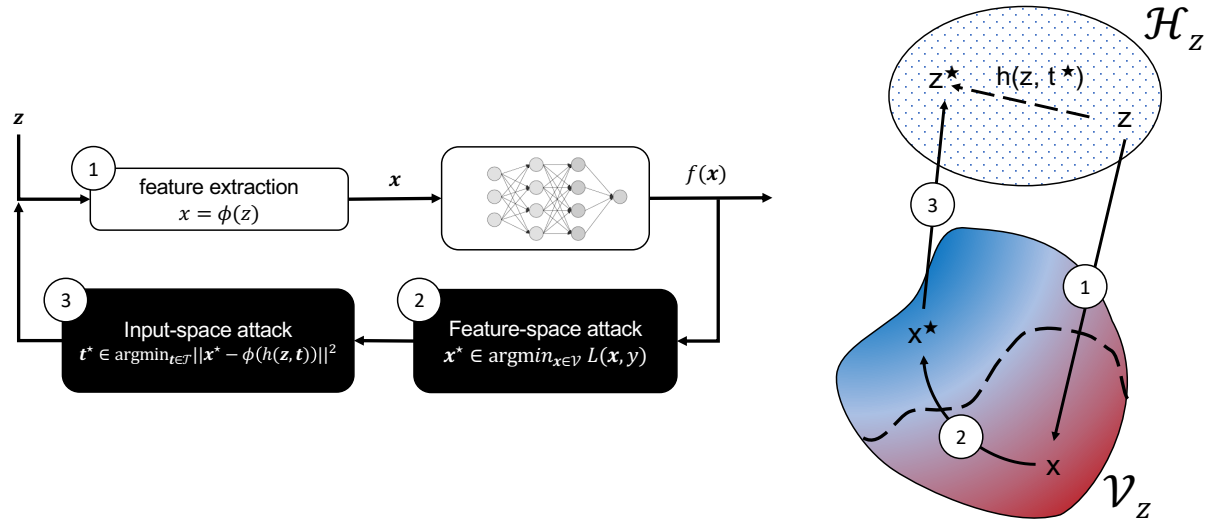


Figure 3.1: The steps performed by RAMEn to produce adversarial examples.

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, applied to $\boldsymbol{z}$, generates $\boldsymbol{x}^\star$. This can be expressed as a reconstruction problem, in the form of a minimization:

$$\boldsymbol{t}^\star \in \arg\min_{\boldsymbol{t}\in\mathcal{T}} \|\boldsymbol{x}^\star - \phi(h(\boldsymbol{z},\boldsymbol{t}))\|^2 \tag{3.6}$$

Non-differentiable model: If all functions are non-differentiable including the model itself, e.g. random forest classifiers, the attacker can only consider the use of black-box optimizer algorithms, since no gradient information is available.

Although our work share some common traits with the formalization proposed by Pierazzi et al. [29], we highlight the main differences between the two approaches. They explicit the effects of the application of manipulations to a sample, by decoupling the transformation into two components: the translation inside the feature space and the projection inside the set of feature space points that satisfy all constraints imposed by an attacker. Since we apply practical manipulations that satisfy all constraints by design, we do not need to project the points inside that set.

The use of non-differentiable feature extractor is a recurring pattern in security detectors, also in the case of end-to-end models. By truncating and embedding, these networks 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 3.1. At the beginning of each iteration, the algorithm applies the practical manipulations to the input sample, and

it maps them inside the feature space by applying ϕ, computing $\boldsymbol{x}$ (line 3). Such vector is used as initialization for the minimization problem (line 4). The optimization is carried on by searching for suitable candidates inside $\mathcal{V}_z$, that is the set representing all possible manipulated versions of the original sample. After having computed the best feature representation for the candidate adversarial example, we invert both the feature mapping and the application of practical manipulations by searching for a vector of parameters $\boldsymbol{t} \in \mathcal{T}$ that, applied to the original sample, is as close as possible to the optimized feature vector $\boldsymbol{x}^\star$ (line 5). The whole algorithm is iterative, allowing maximum freedom to the

Algorithm 1: General version of RAMEn algorithm

Data: malware $\boldsymbol{z}$, iterations N, target class label y
Result: $\boldsymbol{t}^\star, \boldsymbol{z}^\star$

1 $\boldsymbol{t}^{(0)} \in \mathcal{T}$
2 **for** i **in** $[0, N-1]$
3 $\quad \boldsymbol{x} = \phi(h(\boldsymbol{z}, \boldsymbol{t}^{(i)}))$
4 $\quad \boldsymbol{x}^\star = \arg\min_{\boldsymbol{x}' \in \mathcal{V}_z} L(f(\boldsymbol{x}'), y)$ # *initialize* $\boldsymbol{x}'$ *with* $\boldsymbol{x}$ *at beginning*
5 $\quad \boldsymbol{t}^{(i+1)} = \arg\min_{\boldsymbol{t} \in \mathcal{T}} \| \boldsymbol{x}^\star - \phi(h(\boldsymbol{z}, \boldsymbol{t})) \|^2$
6 $\boldsymbol{t}^\star = \boldsymbol{t}^{(N)}$
7 $\boldsymbol{z}^\star = h(\boldsymbol{z}, \boldsymbol{t}^\star)$
8 **return** $\boldsymbol{t}^\star, \boldsymbol{z}^\star$

attacker. By specifying different strategies for solving the two minimization problems, the attacker can customize RAMEn to their need.

3.2 Practical Manipulations

Now that we have presented the formalization, we focus our attention of transformations that alter the representation of samples without affecting their original semantics. So, we need to explore structural mutations for the Windows PE file format, and syntactical manipulations for the SQL language.

3.2.1 Windows PE format manipulations

Since attackers want to camouflage malicious content without compromising its functionality, they need an effective way of perturbing the on-disk representation of a program. As explained in Section 2.2.1 (on page 9), the operating system loads the executables, mapping the sections, with their respective permissions, into the process address space. To do so, the Windows loader must find the components of the program inside the executable.

Since attackers want to produce space for injecting an adversarial payload, they can alter the representation to their advantage, for instance, by shifting content at their will. These transformations can be used to create space inside the input binary, and hide all malicious code from the target network. The payload is then optimized using the implementation of RAMEn expressed in Algorithm 2 (on page 33), by specifying inside h how the content is shifted or extended. A simplified graphical representation of the following strategies is depicted in Figure 3.2. 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.

3.2.1.1 DOS Header editing

The DOS header is kept for compatibility with older Microsoft's operating systems. Since the only two important fields are the magic number `MZ`, and the 4 byte-long integer at offset `0x3c`, all other bytes can be used by the attacker as a space for storing an adversarial payload. Demetrio et al. [14] 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. As already said, all bytes inside the range [`0x02`, `0x3c`) are freely editable. We extend this range by adding also all bytes in the range [`0x40`, PE-header location). The position of the PE header may vary from executable to executable, but its offset is written inside the DOS header.

3.2.1.2 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 "ancient" DOS systems. The attacker can abuse this pointer to their will, 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 content of the file to match the introduced gap, keeping the alignment as expressed by 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.

3.2.1.3 Content Shifting

Each section is retrieved by the Windows loader by using the *physical offset*, that is a four byte-long 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

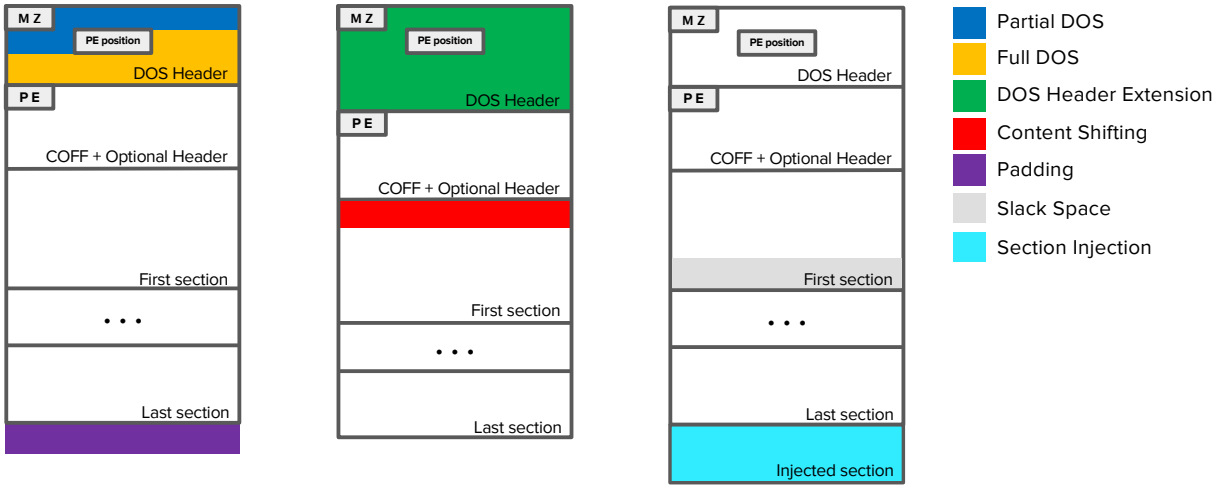


Figure 3.2: Graphical representation of the location perturbed by the different strategies.

executable will not be executed, and considered corrupted. While loading the program, the operating system 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, specified by a 4 byte-long unsigned integer inside the section entry. The attacker can edit the physical offsets of each section, by shuffling the content of the file, with the only constraint imposed by section alignment. They can also add content at their will, without breaking the functionality of the program. The latter is a way, for the attacker, to carve space inside the executable, and insert adversarial noise there.

3.2.1.4 Section injections

The attacker can add new sections inside the binary, by adding a new section table entry with a name, a set of characteristics that tells the loader how to deal with this new element of the binary, and the content to inject.

3.2.1.5 Slack Space

To maintain the alignments inside the file structure, the compiler and the linker might insert padding regions between sections. These bytes can be freely modified, since the program ignores their existence, and small adversarial payloads can be inserted in such space.

3.2.1.6 Padding

Since all adversarial content is appended at the end of the file, no structure is altered. The content that can be added can be extremely long, and it will not affect the functionality of the program.

3.2.2 SQL queries manipulations

SQL queries are commands executed by Database Management Systems (DBMS). The SQL standard provides many syntactical variations, with the same semantics, that attackers can leverage. The following sections detail some of these semantic-preserving transformations.

3.2.2.1 Case Swapping

Since SQL is case insensitive, the semantics of a query is not affected by alterations in the letter case, unless such characters are inside string literals. Hence, for instance, the keyword `Select` and the keyword `SeLEct` have the same meaning.

3.2.2.2 Whitespace Substitution

This transformation relies on the equivalence between several alternative characters that act as separators (whitespaces) between the query tokens. For instance, whitespaces include `\n` (line feed), `\r` (carriage return) and `\t` (horizontal tab). Each of these characters can be replaced by an arbitrary, non-empty sequence of the others without altering the semantics of the query.

3.2.2.3 Inline comments

Comments of the form `/* ... */` can be arbitrarily inserted between tokens of a query. For instance, `SELECT * from users` and `SELECT * from /* whatever */ users` are equivalent queries.

3.2.2.4 Comment Rewriting

Following the practical manipulation described before, any combination of characters that are added inside a comment, with the exception of the sequence `*/` or any characters that form that sequence, are ignore by the DBMS. Hence, inline comments can be filled

Operator	Example
Case Swapping	CS(`admin' OR 1=1#`) → `ADmIn' oR 1=1#`
Whitespace Substitution	WS(`admin' OR 1=1#`) → `admin'\n OR \t 1=1#`
Comment Injection	CI(`admin' OR 1=1#`) → `admin'/**/OR 1=1#`
Comment Rewriting	CR(`admin'/**/OR 1=1#`) → `admin'/*abc*/OR 1=1#xyz`
Integer Encoding	IE(`admin' OR 1=1#`) → `admin' OR 0x1=1#`
Operator Swapping	OS(`admin' OR 1=1#`) → `admin' OR 1 LIKE 1#`
Logical Invariant	LI(`admin' OR 1=1#`) → `admin' OR 1=1 AND 2<>3#`

Table 3.1: Examples of SQL practical manipulations.

with (almost) arbitrary strings of any length, without worrying about side-effects. For instance, `SELECT * from users /* return every user */` and `SELECT * from users /*alala*/` will produce the same results.

3.2.2.5 Integer Encoding

Numbers can be expressed in multiple ways, including alternative base representations. For instance, the number `42` can be expressed in its hexadecimal representation `CONV('2A', 16, 10)`. Also, numbers can be expressed as nested queries: `(SELECT 42)` is equivalent to `42`.

3.2.2.6 Operator Swapping

Some operators can be replaced by others that behave in the same way. For instance, the behavior of `=` (equality check) can be simulated by `LIKE` (pattern matching).

3.2.2.7 Logical Invariant

Each condition can be "expanded" by adding boolean expressions that are always evaluated to true or false, called *opaque predicates*.
For instance, `SELECT * from users WHERE a=1 AND 1=1` produces the very same result of `SELECT * from users WHERE a=1`.

Table 3.1 provides a compact list, including a short, mnemonic definition, of the operators described above. An example of perturbed SQL injection is shown in Figure 3.3. The original query is twisted to a long sequence of tokens, but there is the certainty that the original semantics is preserved when evaluated.

```
1 SELECT * from users where usr='admin' OR 1=1#
2 SELECT * from users where usr='admin' OR 0X1=1 or 0x726!=0x726 OR 0x1Dd
     not IN/*(seleCt 0X0)>c^Bj>N]*/ ((SeLeCT 476),(SELECT (SElEct 477)),0
    X1de) oR 8308 noT lIkE  8308\x0c AnD truE OR 'FZ6/q' LiKE 'fz6/qI'
    anD TRUE anD '>U' != '>uz'#t'%'03;Nd
3
```

Figure 3.3: Two different SQL queries that, when evaluated return the same result.

3.3 White box byte-optimization attacks

We start by implementing white-box attacks against malware classifiers. To provide an end-to-end algorithm, we extend the algorithm proposed by Demetrio et al. [14] and Kolosnjaji et al. [15], 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 metric over the input bytes, 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 programs longer than a specific amount will be cropped before passing to the embedding layer. Hence, we can encode ϕ as the composition of the cropping 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: $L(f(\phi(\boldsymbol{z})), y) = f(\phi(\boldsymbol{z}))$.

Algorithm 2 implements the resolution of both problems stated in Equation 3.5 and 3.6, by using a slightly modified version of the solver proposed by Kolosnjaji et al. [15]. We use $\hat{\phi}$ to denote the function that encodes a single byte inside the feature space (used in line 1), with also the addition of the padding symbol.

Firstly, 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 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 γ 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 256 embedded byte values on this direction, computing the distance point-to-line and the alignment with such direction (line 9). The padding value is not projected, as it can not be used as a replacement value.

The best byte replacement value is the one with the least non-zero distance, with positive alignment to the defined direction (line 10).

Algorithm 2: Implementation of RAMEn used for optimizing and reconstructing single bytes inside the input program.

Data: malware $\boldsymbol{z}$, number of bytes to optimise γ, iterations N
Result: $\boldsymbol{t}^\star, \boldsymbol{z}^\star$

1 $\mathbf{E}_i = \hat{\phi}(i), \forall i \in [0, 256]$
2 $\boldsymbol{t}^{(0)} \in \mathcal{T}$
3 **for** i **in** $[0, N-1]$
4 $\quad \mathbf{X} = \phi(h(\boldsymbol{z}, \boldsymbol{t}^{(i)}))$
5 $\quad \mathbf{G} = -\nabla_{\mathbf{X}} f(\mathbf{X}) \odot \boldsymbol{m}$
6 $\quad$ **for** k **in** argsort $(\parallel \boldsymbol{G} \parallel)_{0,\ldots,\gamma-1} \wedge \mathbf{G}_k \neq \mathbf{0}$
7 $\quad\quad$ **for** j **in** $[0, \ldots, 255]$
8 $\quad\quad\quad \mathbf{S}_{k,j} = \mathbf{G}_k^T \cdot (\mathbf{E}_j - \mathbf{X}_k)$
9 $\quad\quad\quad \widetilde{\mathbf{X}}_{k,j} = \parallel \mathbf{E}_j - (\mathbf{X}_k + \mathbf{G}_k \mathbf{S}_{k,j}) \parallel_2$
10 $\quad\quad \boldsymbol{t}_k^{(i+1)} = \arg\min_{j:\mathbf{S}_{k,j}>0} \widetilde{\mathbf{X}}_{k,j}$
11 $\boldsymbol{t}^\star = t^{(N)}$
12 $\boldsymbol{z}^\star = h(\boldsymbol{z}, \boldsymbol{t}^\star)$
13 **return** $\boldsymbol{t}^\star, \boldsymbol{z}^\star$

3.3.1 Implementing Byte Attacks within RAMEn

We formalize, by means of pseudo-code, both our novel practical manipulations, described in Section 3.2 (on page 27). Algorithm 3 shows the implementation of the Partial DOS attack, proposed by Demetrio el al. [14]. It takes the vector of parameters $\boldsymbol{t}$ and it copies its content at the desired position inside the sample (line 2). The Full DOS variant, shown in Algorithm 4 ensures that the parameters are enough for the application of the transformation, the algorithm checks its length and fixes it to match the desired length (line 4). After that, the algorithm proceeds with the rewriting of the initial bytes of the DOS header (line 5), and the also the bytes that comes after the pointer to the PE header and the PE header itself (line 6) The standard padding technique is expressed in Algorithm

7, often used in the state of the art [15,16,28]. Algorithm 5 shows how to enlarge the DOS header to add the adversarial payload, while Algorithm 6 shows how to incorporate the payload before the first section.

For both the *Extend* and *Shift* algorithms, we need to insert a string of bytes in the desired position (line 5 of both Algorithms 5 and 6), The first entry of the vector of parameters specifies how many bytes the attacker would like to insert inside the sample (line 1 of both Algorithms 5 and 6). Since such content must be a multiple of the alignment, this quantity is accordingly increased by rounding towards the nearest multiple of the alignment itself. 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 byte `0x00` is concatenated with itself *align* times, and we use the symbol `+` for indicating the concatenation of byte strings and byte values (line 5 of both Algorithms 5 and 6). After the insertion, we fix the constraints imposed by the format, including fixing the offset of the PE header for the *Extend* manipulation (line 6 of Algorithm 5), and the offset of the content of each section (from line 7 to line 10 of both Algorithms 5 and 6). Lastly, the content of the vector of parameters $\boldsymbol{t}$ is copied inside the sample (line 11 for Algorithm 6, and lines 11, 12 for Algorithm 5). In our experimental analysis, we set the desired payload length to 512 bytes for the *Extend* attack, and 1024 for the *Shift* attack. During the attack, such length is adjusted to match the file alignment of the sample to be perturbed. The vector of parameters is padded with zeros if it is shorter than 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.

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

Data: malware $\boldsymbol{z}$, vector of parameters $\boldsymbol{t}$
Result: $\boldsymbol{z}'$
1 $\boldsymbol{z}' = \boldsymbol{z}$
2 $\boldsymbol{z}'_{2,\dots,59} = \boldsymbol{t}$
3 return $\boldsymbol{z}'$

3.4 Black-box Attacks with Practical Manipulations

We investigate also the application of our practical manipulation in black-box cases, where the attacker do not know the model to attack. We describe the strategies we have implemented for testing this scenario, focusing on transfer attacks and query attacks.

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

Data: malware $\boldsymbol{z}$, vector of parameters $\boldsymbol{t}$
Result: $\boldsymbol{z}'$

$off = \boldsymbol{z}$.pe_offset
$\boldsymbol{z}' = \boldsymbol{z}$
if $|\boldsymbol{t}| < off + 58$
    $\boldsymbol{t} = \boldsymbol{t} + \texttt{0x00} * (off + 58 - |\boldsymbol{t}|)$
$\boldsymbol{z}'_{2,\dots,59} = \boldsymbol{t}_{0,\dots,57}$
$\boldsymbol{z}'_{64,\dots off-1} = \boldsymbol{t}_{58,\dots,|\boldsymbol{t}|-1}$
return $\boldsymbol{z}'$

Algorithm 5: Implementation of $h(\boldsymbol{z}, \boldsymbol{t})$ for the Extend practical manipulation

Data: malware $\boldsymbol{z}$, vector of parameters $\boldsymbol{t}$
Result: z'

$align = \lceil \boldsymbol{t}_0 / \boldsymbol{z}$.file_alignment $\rceil * \boldsymbol{z}$.file_alignment
if $|\boldsymbol{t}| < align$
    $\boldsymbol{t} = \boldsymbol{t} + \texttt{0x00} * (align - |\boldsymbol{t}|)$
$off = \boldsymbol{z}$.pe_offset
$\boldsymbol{z}' = \boldsymbol{z}_{0,\dots,off-1} + \texttt{0x00} * align + \boldsymbol{z}_{off,\dots,|\boldsymbol{z}|-1}$
$\boldsymbol{z}'$.pe_offset $= off + align$
$S = \boldsymbol{z}'$.get_sections()
for s ***in*** S **do**
    s.physical_offset $= align + s$.physical_offset
end
$\boldsymbol{z}'_{2,\dots,59} = \boldsymbol{t}_{0,\dots,57}$
$\boldsymbol{z}'_{64,\dots off-1} = \boldsymbol{t}_{58,\dots,|\boldsymbol{t}|-1}$
return $\boldsymbol{z}'$

Algorithm 6: Implementation of $h(\boldsymbol{z}, \boldsymbol{t})$ for the Shift practical manipulation

Data: malware $\boldsymbol{z}$, vector of parameters $\boldsymbol{t}$
Result: z'

$align = \lceil \boldsymbol{t}_0 / \boldsymbol{z}.\text{file\_alignment} \rceil * \boldsymbol{z}.\text{file\_alignment}$
if $|\boldsymbol{t}| < align$
    $\boldsymbol{t} = \boldsymbol{t} + \texttt{0x00} * (align - |\boldsymbol{t}|)$
$fs = \boldsymbol{z}.\text{get\_first\_section\_offset}()$
$\boldsymbol{z}' = \boldsymbol{z}_{0,\ldots,fs-1} + \texttt{0x00} * align + \boldsymbol{z}_{fs,\ldots,|\boldsymbol{z}|-1}$
$\boldsymbol{z}'.\text{pe\_offset} = z'.\text{pe\_offset} + align$
$S = \boldsymbol{z}'.\text{get\_sections}()$
for s ***in*** S **do**
    $s.\text{physical\_offset} = align + s.\text{physical\_offset}$
end
$\boldsymbol{z}'_{fs,\ldots,fs+(align-1)} = \boldsymbol{t}$
return $\boldsymbol{z}'$

Algorithm 7: Implementation of $h(\boldsymbol{z}, \boldsymbol{t})$ for the Padding practical manipulation

Data: malware $\boldsymbol{z}$, vector of parameters $\boldsymbol{t}$
Result: $\boldsymbol{z}'$

$\boldsymbol{z}' = \boldsymbol{z} + \boldsymbol{t}$
return $\boldsymbol{z}'$

3.4.1 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. We focus on the latter, by optimizing attacks on the networks we consider for this work, and transferring them against all others.

3.4.2 Query Attacks

The attacker can also target the victim system by exploiting the result of the prediction step, without optimizing the attack on a surrogate model. By sending queries to the target model, the attacker can infer how the victim behaves locally around a particular set of samples they want to be misclassified.

3.4.2.1 Genetic algorithm strategy

We implement RAMEn with the use of a genetic black-box optimizer, the same used by Demetrio et al. [23] 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 parameters, since both Algorithm 5 and 6 consider each entry of vectors $\boldsymbol{t}$ as bytes to be placed inside the sample.

We summarize the optimizer in Algorithm 8, 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. The best solution is the one with minimal score. To explore the attack feature space, the genetic algorithm performs three steps, mimicking the process of biological evolution: *selection*, *cross-over*, and *mutation*.

Function *selection* extracts attack feature vectors, whose value of the objective function is minimal, from the population that has been generated so far. Intuitively, these vectors are the most promising for computing an adversarial example.

Function *crossover* breeds two selected vectors together, by mixing their values as they were chromosomes. To clarify, a random point is chosen inside these vectors, and all values

Algorithm 8: Pseudo-code of the genetic black-box optimizer.

Data: population size m, generations N
Result: $\boldsymbol{t}^\star, \boldsymbol{z}^\star$

$\mathbf{P} = m$ random points
F $= \{(\mathbf{P}_i, L(f(\phi(h(\boldsymbol{z}, \lceil 255\mathbf{P}_i \rfloor))), \text{y}))\}_{i=1}^{m}$
i $= 0$
S $=$ F
while $i < N$ ***and*** *not stagnating* **do**
    S $=$ *selection*(S)
    S $=$ *crossover*(S)
    S $=$ *mutate*(S)
    S $= F \cup \bigcup_{\boldsymbol{t} \in S} \{(\boldsymbol{t}, L(f(\phi(h(\boldsymbol{z}, \lceil 255\boldsymbol{t} \rfloor))), y))\}$
    i $=$ i+1
end
$\boldsymbol{t}^\star = min(S)$
$\boldsymbol{z}^\star = h(\boldsymbol{z}, \boldsymbol{t}^\star)$
return $\boldsymbol{t}^\star, \boldsymbol{z}^\star$

from that point on are exchanged between the two chosen vectors. This ensures the creation of new elements, helping the algorithm exploring the space of solutions.

Lastly, function *mutation* applies a random mutation inside a selected feature vector, providing the altered vector with a unique trait that might help the evolution towards successful adversarial examples. At each round, a set of new candidates is produced and evaluated. Before moving to the next round, all new offsprings are added to the population. In this way, the genetic process starts again, selecting the fittest candidate for survival, eventually reaching a local minimum for the problem.

The algorithm terminates after N iterations, and the detector is queried exactly $N \cdot (m+1)$ times. Our strategy sets the constraint T over the whole amount of queries sent to the detector, that is $T = N(m+1)$, so the attacker must choose N accordingly. Also to avoid waste of computations, if the value of the fitness stagnates, i.e. no other local minima are found, for at least 5 generations, the process is halted.

Implementing GAMMA: As proposed in previous work [23], we implement GAMMA (Genetic Adversarial Machine learning Malware Attack), a black-box attack strategy that adds content taken from goodware samples. We decided to apply two different strategies: *padding* and *section injection*. The content that is added to input malware samples is directly taken from sections belonging to legit executables. Intuitively speaking, we believe that the addition of content taken from benign programs would move the samples closer to the benign class. Each dimension $\mathcal{T}$ is the content of a benign section, hence the attacker

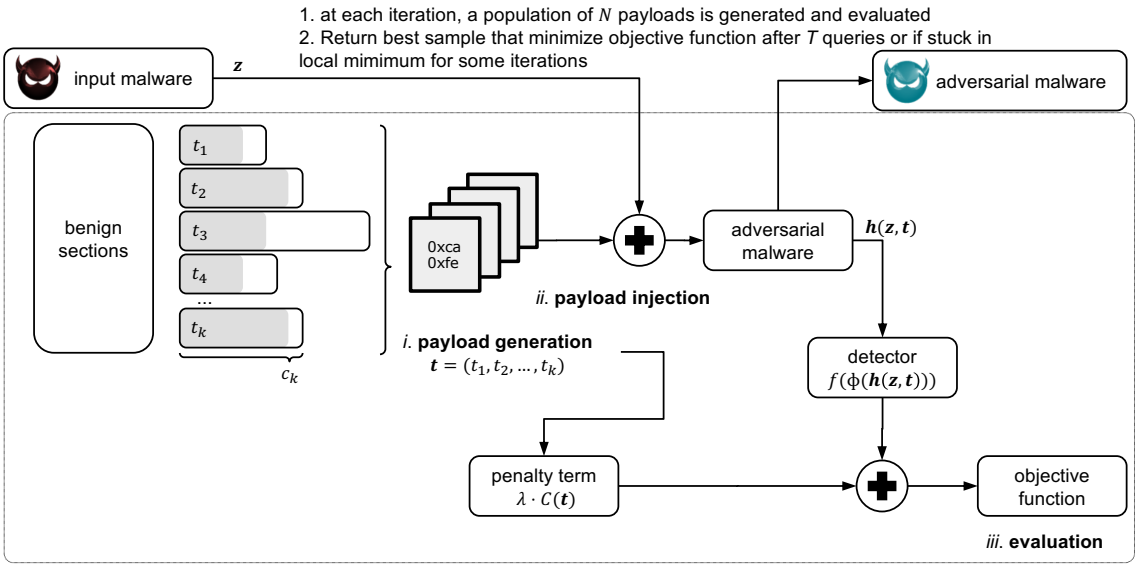


Figure 3.4: Scheme of GAMMA, by summing up the major steps of the process.

needs to decide how many pieces of goodware use during the optimization. The benign content can be added in chunks of variable size, which helps relax the problem to the continuous domain. We limit the usage of GAMMA to these two manipulations, but all other practical manipulations could have been used.

For each payload, GAMMA generates adversarial malware that is passed to the input detector, whose output is a probability of being malicious. The objective function is the sum of the score mentioned before and a penalty term, controlled by an input regularization parameter. After having sent T queries, GAMMA outputs the best adversarial malware found so far. Since we are adding bytes, we crop the content of the *i-th* section by taking only a fraction expressed by $\boldsymbol{t}_i$ of the original content when we add the section to the input malware. To clarify this concept, if $\boldsymbol{t}_i = 0.4$, the algorithm takes the 40% of the content from the i-th section. This product is needed since not all sections share the same length, and they must be penalized accordingly. The practical manipulation functions's implementation are shown in Algorithm 9 and Algorithm 10, as we rely on both *padding* and *section injection*. Figure 3.4 shows the overall GAMMA strategy, highlighting all its ingredients.

3.4.2.2 Genetic optimization of bytes

We take the attacks proposed in Section 3.3 (page 32) and instead of optimizing the values using the gradient technique, we use the genetic optimizer described in Section 3.4.2.1

Algorithm 9: Implementation of $h(\boldsymbol{z}, \boldsymbol{t})$ for GAMMA - padding

Data: malware $\boldsymbol{z}$, vector of parameters $\boldsymbol{t}$
Result: $\boldsymbol{z}'$

$\boldsymbol{z}' = \boldsymbol{z}$
for i ***in*** $0, ...|\boldsymbol{t}| - 1$ **do**
    $\boldsymbol{s} = \text{get\_section\_content}(i)$
    $\boldsymbol{s} = \text{crop}(\boldsymbol{s}, \lceil \boldsymbol{t}_i | \boldsymbol{s} | \rfloor)$
    $\boldsymbol{z}' = \boldsymbol{z}' + \boldsymbol{s}$
end
return $\boldsymbol{z}'$

Algorithm 10: Implementation of $h(\boldsymbol{z}, \boldsymbol{t})$ for GAMMA - section injection

Data: malware $\boldsymbol{z}$, vector of parameters $\boldsymbol{t}$
Result: $\boldsymbol{z}'$

$\boldsymbol{z}' = \boldsymbol{z}$
for i ***in*** $0, ...|\boldsymbol{t}| - 1$ **do**
    $\boldsymbol{s} = \text{get\_section\_content}(i)$
    $\boldsymbol{s} = \text{crop}(\boldsymbol{s}, \lceil \boldsymbol{t}_i | \boldsymbol{s} | \rfloor)$
    $\boldsymbol{z}' = \text{add\_section}(\boldsymbol{z}', \boldsymbol{s})$
end
return $\boldsymbol{z}'$

(page 37). In this way, the optimizer chooses a byte value for each entry of the vector of manipulations $\boldsymbol{t}$ used for creating adversarial malware.

3.4.2.3 Mutational fuzzing

Another technique for computing adversarial examples in a black-box setting is *fuzzing* the input and scoring the produced output. A *fuzzer*, in its simplest form, is a tool that takes in input an object, and alters it by applying random manipulations. This technique is useful for testing software with random inputs, trying to enhance the coverage of the corresponding test suite.

To create better inputs, smarter fuzzers look at the response of the target to test. Depending on the success or the failure of a test, a fuzzer may decide to apply different strategies for mutating the input. In this way, the randomness of the fuzzing creates multiple variants of the original input object. Usually, a fuzzer is used for finding bugs in a program, and once a bug is found the input is saved for further analysis.

Here, a fuzzer can be used as a black-box optimizer: it randomly mutates the sample, and each mutated sample is scored with an objective function. Depending on the achieved score, the fuzzer decides to discard or keep the input, and it proceeds until either it finds a local minimum or maximum or it runs out of iterations. This methodology belongs to the class of *guided mutational fuzz testing* [86, 87]. Figure 3.5 schematically depicts this approach.

We use a fuzzer to generate adversarial examples, by applying practical manipulations to the original sample until evasion is achieved. In particular, we compute adversarial examples against Web Application Firewalls (WAFs) [22], using the manipulations we have described in Section 3.2.2 (on page 30). Web Application Firewalls (WAFs) are a prominent family of IDS, widely adopted [88] to protect ICT infrastructures. Their detection algorithm applies to HTTP requests, where they look for possible exploitation patterns, e.g., payloads carrying a SQL injection.

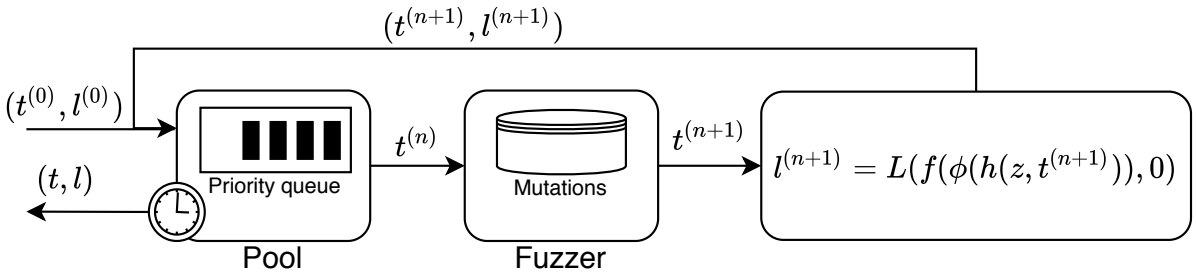


Figure 3.5: An outline of the mutational fuzz testing approach.

Since WAFs work at application-level, they have to deal with highly expressive languages such as SQL and HTML. Figure 3.3 shows two queries injected with two different payloads, both starting with `'admin' OR ...`. Notice that the two payloads are semantically equivalent. As a matter of fact, both reduce the above query to `SELECT * FROM users WHERE name='admin' OR` $\top$ `#...` where $\top$ is a tautology and ... is a trail of commented characters. Ideally a WAF should reject both these payloads. However, when classification is based on a mere syntactical analysis, this might not happen. Hence, the goal of an attacker amounts to looking for some malicious payload that is undetected by the WAF.

Algorithm 11: Core algorithm of WAF-A-MoLE.

Data: initial sample z, threshold T
Result: $\boldsymbol{t}^*$, $\boldsymbol{z}^*$

```
1  Q ← create_priority_queue()
2  l ← L(f(φ(z)), 0)
3  enqueue(Q, 0, l)
4  while y ≥ T do
5  |   t ← mutate(head(Q))
6  |   l ← L(f(φ(h(z, t))), 0)
7  |   enqueue(Q, t, l)
8  end
9  t* ← head(Q)
10 z* ← h(z, t*)
11 return t*, z*
```

A pseudo code implementation of the core evasion algorithm is shown in Algorithm 11. The objective function to minimize is a loss function that computes the distance between the computed score and the benign class, explicitly expressed with the value 0. We use a priority queue Q (line 1) to store the intermediate results of the optimization process. At the beginning of the procedure, the queue only contains $\mathbf{0}$, that is the vector that applies no practical manipulations on the initial sample (line 3). At each iteration, the fuzzer uses random practical manipulations to modify the head element of Q, i.e. the vector of coefficients that generates the adversarial example with the lowest probability of being detected (line 5). The mutated vector is inserted again inside the queue Q , along with the new score computed by the detector after the application of the manipulations (line 7). After the last iteration, we extract the first element of the priority queue that is the vector $\boldsymbol{t}$ containing the best mutations to apply to produce an adversarial example (line 9).

Mutation tree: The priority queue of Algorithm 11 contains a sequential representation of a mutation tree. Starting from a root element, i.e. the initial payload ($\boldsymbol{z}$ in Algorithm 11), a mutation tree contains elements obtained through the application of

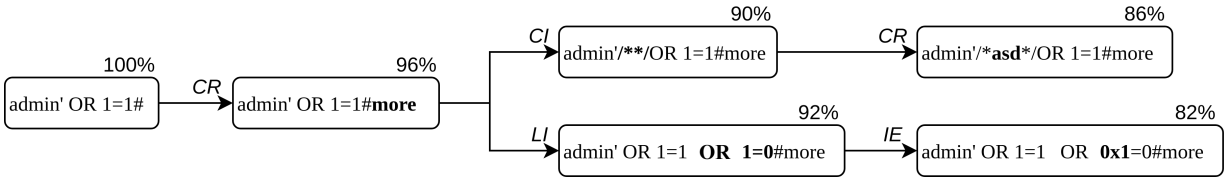


Figure 3.6: A possible mutation tree of an initial payload.

some mutation operator. A possible instance of a mutation tree is shown in Figure 3.6. Each edge is labeled with an identifier of the applied mutation operator. Also, each node is labeled with a possible classification value (in percentage). The corresponding queue is given by the sequence of the nodes in the mutation tree ordered by the associated classification value. After applying a mutation (actually after a full *mutation round*), the payload is evaluated and added to the priority queue, along with information about the payload that generated it. Keeping all individuals in the initial population helps avoiding local minima: when a payload is unable to create better payloads, the algorithm tries to backtrack on old payloads to create a new branch on the mutation tree.

Efficiency: the main bottleneck of our algorithm is the classification step, since the detector needs to apply its own feature extraction technique every time it scans a sample. While the computing the probability score alone is fast, the feature extraction process may require non-negligible string parsing operations. All mutation operators described in Section 3.2.2 (on page 30) rely on efficient string parsing, based on regular expressions, hence they do not contribute to a slow-down of the approach.

To mitigate the slowness of the feature extraction process, we precompute a subset of possible mutations chosen at random, and we forward them to the classifier. We call *mutation round* each of these batch generated candidates. Then, we run all classification steps in parallel and we discharge the mutants that increase the classification value of their parent. In this way we take advantage of the parallelization support of modern CPUs.

For memory efficiency, we only enqueue a mutant if it improves the classification value of its parent. In this way we mitigate the potential, exponential blow-up of the mutation tree. On the negative side, each branch of the mutation tree only evolves monotonically which might result in the algorithm stagnating on local minima.

3.5 Implementing existing white-box attacks within RAMEn

Table 3.2 contains all specifications used for implementing the techniques proposed in the previous literature within the RAMEn framework. Every strategy must define, how to

1. use the gradient,
2. move inside the space, and
3. reconstruct the sample in the original input space.

Both Kolosnjaji et al. [15] and Demetrio et al. [14] share the same methodology and same algorithm, but applied to different locations. 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. [15] append content at the end of the input program, while Demetrio et al. [14] fill a portion of the DOS header of the input program. Both strategies solve the problem in Equation 3.5 by searching the closest embedding value parallel to the gradient (*closest positive* in Table 3.2). This is done for each value that needs to be modified. The problem in Equation 3.6 is solved by inverting the look-up operation performed by the embedding layer (*inverse look-up* in Table 3.2). This operation can not fail, since the

Attack	**Proposed in**	**Loss function**	**Practical manipulations**	**Feature space opt.**	**Reconstruction**
Extend	This thesis	malware score	extend DOS header	closest positive	inverse look-up
Shift	This thesis	malware score	shift section content	closest positive	inverse look-up
Padding	Kolosnjaji et al. [15]	malware score	padding	closest positive	inverse look-up
Partial DOS	Demetrio et al. [14]	malware score	partial DOS header	closest positive	inverse look-up
FGSM (iterative)	Kreuk et al. [27]	classification loss	padding + slack space	FGSM	closest
FGSM (1 iteration)	Suciu et al. [28]	classification loss	padding + slack space	FGSM	closest
Equivalent instructions	Sharif et al. [69]	C&W loss	equivalent instructions	random	aligned mutation

Table 3.2: Implementing the white-box attacks of the state of the art, using RAMEn.

search at the step before chooses only values that correspond to bytes inside the input space.

Kreuk et al. [27] and Suciu et al. [28] apply the fast gradient sign method (FGSM) [54] inside slack space, that is the unused space between sections, and within appended padding. Then, at the end of all 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 3.2).

Sharif et al. [69] 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 each sample is constructed by applying random transformation.

Except for the manipulations proposed by Sharif et al. [69], all strategies describe so far can be found in the SecML library released alongside our paper [89].

Chapter 4

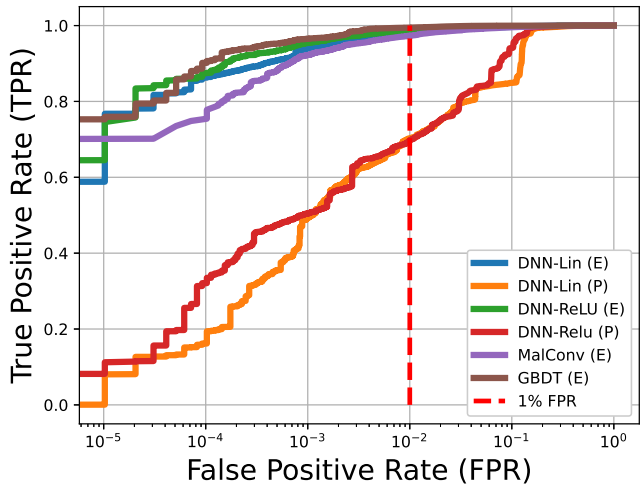
Experimental Analysis

In this chapter we analyze the results of the attacks proposed in Section 3.3 and 3.4. For each setting, we briefly introduce the hardware setup we used for carrying on the experiments.

However, before delving inside the performances of the different attacks against the target classifiers, we compute the *Receiver Operating Characteristic (ROC)* of the four models. The score has been computed on the first version of the Ember dataset [8].

The left plot of Figure 4.1a highlights that the tree model outmatches the convolutional networks. The letter inside the parenthesis of the legend specifies the dataset used for training the classifier: e means Ember, while p implies 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. This benefit might be connected to the manual feature engineering that is present inside the tree model, instead of letting the network learn the relevant locations by itself. The non-linearity imposed by ReLU activation functions does not impact the overall score, implying that the problem might be linear after having applied convolution functions. MalConv shows comparable results to the network with linear activation functions, and this might be caused by over-fitting, since the dataset used for testing has a non-null intersection with the one used for training the classifier. 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 4.1b all thresholds computed from using the results of the ROC. Since both MalConv and GBDT were originally trained on the dataset used for computing the ROC, they expose very low values of their detection thresholds.

In the following sections we will show all the results of the attacks we have developed and tested. We first address the performance of white-box attacks in Section 4.1, by presenting



(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)

Figure 4.1: The Receiver Operating Characteristic curve (ROC) of the classifiers under analysis, and the detection threshold at 1% FPR.

the efficacy of Algorithm 1 with all the practical manipulations described in Section 3.2. We continue with black-box attacks in Section 4.2, by showing the effectiveness of GAMMA (Section 4.2), where we (i) evaluate the models in presence and absence of the attack (Section 4.2.1.2), (ii) offer a comparison of the performances in the presence or absence of the scores computed by the victims (Section 4.2.1.3), (iii) analyze the time elapsed by the strategy (Section 4.2.1.4), (iv) analyze how techniques like packing behave w.r.t. our technique (Section 4.2.1.5), (v) conclude by presenting how GAMMA can deal also with online remote antivirus programs (Section 4.2.1.6). After, we discuss the efficacy of the attacks used in the white-box scenario, but applied with a genetic black-box optimizer (Section 4.2.2), along with transfer attacks (Section 4.2.3). We conclude our experimental analysis by presenting the results achieved by WAF-A-MoLE (Section 4.2.4.1), discussing why it is able to generate adversarial examples and which model is less likely to be evaded.

4.1 White-box attack results

We computed the experiments on 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 (on page 27). To highlight the performance of our strategy, we encoded other attacks proposed in the state of the art [27, 28] and we ran them against the chosen targets.

The network proposed by Coull et al. [39] has been trained with two different datasets. The first one is Ember [8], which is an open-source dataset of already extracted features and malware hashes, while the second is proprietary. The first dataset is smaller, counting 1.1 M samples, while the second is larger, counting 16.3 M files.

MalConv has been trained on Ember [8], like the tree model proposed by Anderson et al. [8]. The malware set we used for the empirical evaluation are the same used by Demetrio et al. [14].

All strategies are available online as an extension of SecML [90], named *SecML Malware* [89].

We tested all differentiable models in our possession with the attacks formulated in Section 3.2 (*Full DOS*, *Extend* and *Shift*), the header attack [14], the padding attack [15], and an iterative implementation of the fast gradient sign method (FGSM) that addresses both padding and inter-section content [27].

Also, Suciu et al. [28] use the FGSM to compute adversarial payloads, and it can be expressed by the first iteration of the attack proposed by Kreuk et al. [27].

The *Partial DOS* attack proposed by Demetrio et al. [14] 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 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 said in Section 3.3.1, 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. Summing up all ingredients, this strategy considers payloads whose length varies between `630` and `4386` bytes.

Same treatment for the *Shift* attack, by adding `1024` bytes before the first section of the sample, again aligned to the nearest multiple of the specified file alignment. This causes the adversarial payload to have a length between `1024` and `4096` bytes.

The *Padding* attack appends bytes at the end of the file. We set a default payload size of `10240` bytes, motivated by the results obtained by Kolosnjaji et al. [15]. The iterative implementation of FGSM considers both padding and unused space between sections (slack space). While the latter might changes from file to file due to alignment, we set the padding size to `10240` bytes. We set the FGSM free-parameter ϵ to 0.1.

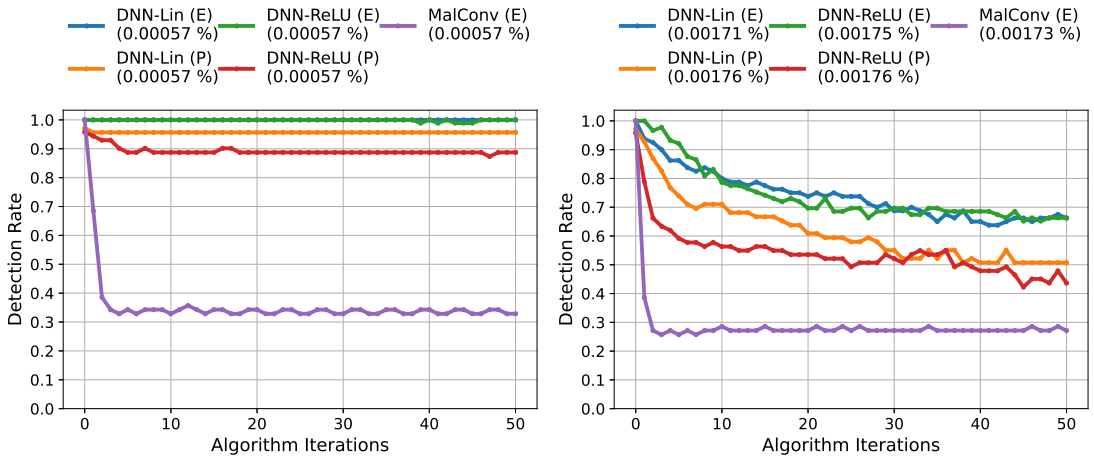
For all attacks, we chose a step size of `256` bytes optimized at each iteration. We show the performance of each white-box techniques we described, targeting the different considered models.

4.1.1 Discussions of the results

Figure 4.2 shows the efficacy of white-box attacks. Each plot sums up the degradation induced by a single specific strategy against the considered classifiers, trained on different datasets (E is Ember, while P is proprietary). 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 input window size. While the *Partial DOS* technique is ineffective, the *Full DOS* attack contribute to lower the detection rate of the networks proposed by Coull et al. [39]. This might be caused by spurious correlations learnt by the network, and altering these values cause the classifier to lose precision. As already pointed out by Demetrio et al. [14], MalConv has a blind spot regarding all header attacks.

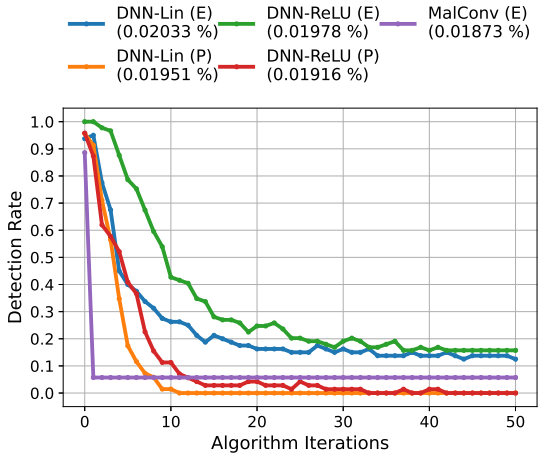
Both our novel strategies, *Extend* and *Shift*, show great performances against all networks. Since these networks use different convolutional layers to learn local spatial information (especially DNN-Lin and DNN-ReLU [39]), it is possible that these models recognize adjacent characters and two-or-three 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 field 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 more 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. [15], and the *FGSM* attack proposed by Kreuk et al. [27] and Suciu et al. [28] 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, then it can not be padded, and all 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 is safe to be manipulated is too small. The strategy proposed by Kreuk et al. performs better than the *Partial DOS*; this is due to the editing of the slack space. Also, this strategy is incapacitated by 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

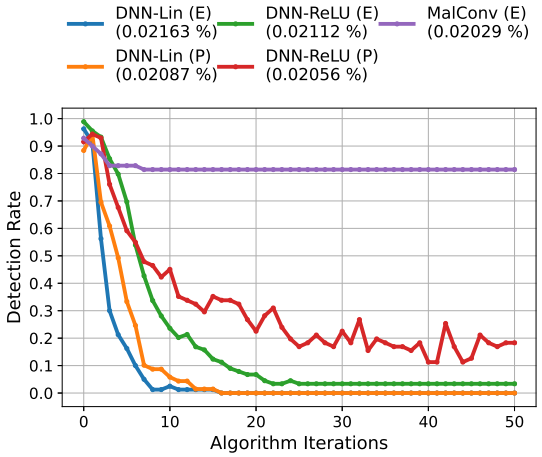


(a) Partial DOS

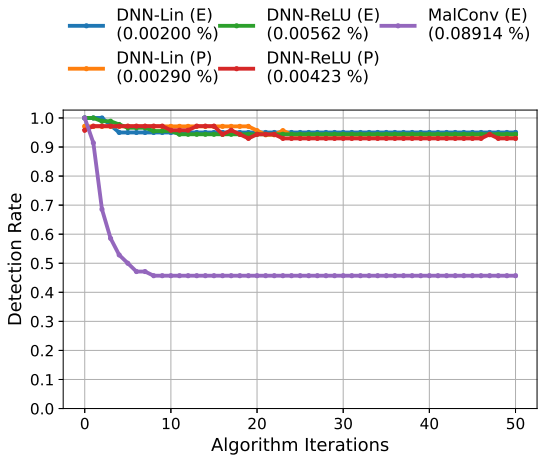
(b) Full DOS



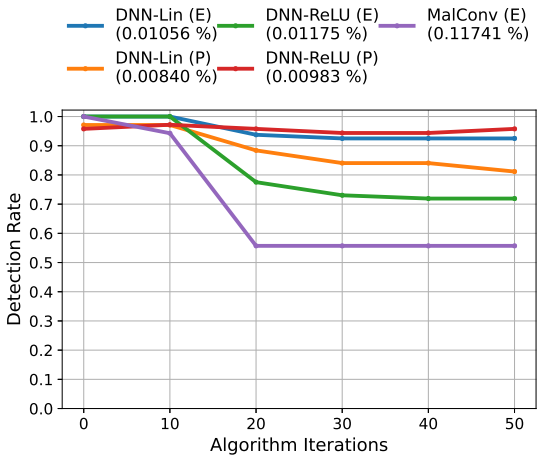
(c) Extend



(d) Shift



(e) Padding



(f) FGSM

Figure 4.2: The results of white-box attacks, expressed as the mean Detection Rate at each optimization step.

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. [15]. 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 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 4.2. This network has a window size of `100 KB`, and each attack alter, on average, less than the 0.03% 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 showed.

4.2 Black-box attack results

We firstly cover the experiments for the genetic and GAMMA attack (discussed in Section 3.4.2.1, on page 37) in Section 4.2.1, then we proceed by showing the results of both query and transfer attacks based on black-box strategies in Section 4.2.2 and Section 4.2.3, respectively. We conclude the analysis by showing the results of the fuzzing technique (presented in Section 3.4.2.3, on page 41) in Section 4.2.4.

4.2.1 GAMMA black-box attack results

We ran our experiments against both EMBER and MalConv 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 [91]. We developed the genetic optimizer of GAMMA using *DEAP* [92]. We tested the attack using a population size N of 10 elements, varying the number of generations G from 1 to 50. We used values for the regularization parameter $\lambda \in \{10^{-i}\}_{i=3}^{9}$. 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 3.4.2.1. 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. All attacks have been implemented with *SecML Malware* [89].

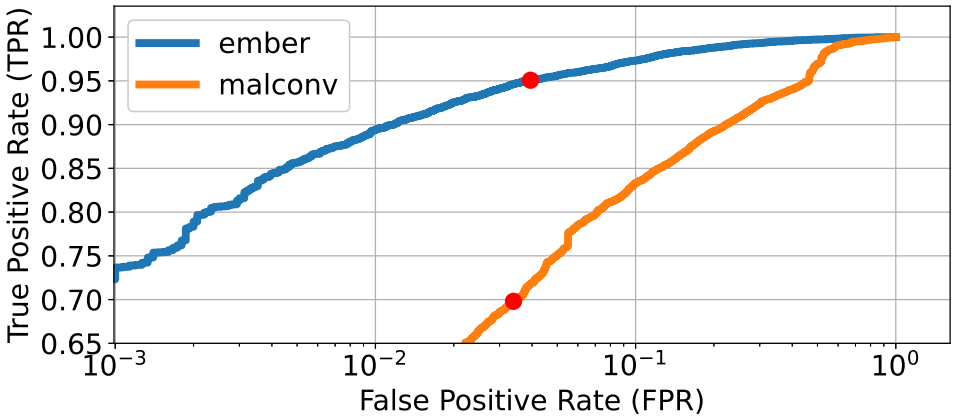


Figure 4.3: Receiver Operating Characteristic (ROC) curve of EMBER and MalConv on 30K samples.

4.2.1.1 Performance on test dataset in 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 [1]. The goodware samples were collected by downloading executable programs from GitHub. The results are shown in Figure 4.3: the threshold chosen for EMBER is 0.8336, which corresponds to an False Positive Rate (FPR) of 0.039 and a True Positive Rate (TPR) of 0.95. The threshold used for MalConv is 0.5, that lead to a FPR of 0.035 and a TPR of 0.69. The red dots inside the plot shows such values directly on the curve. These results are comparable to the description given by the authors of EMBER [8], as both detectors achieve just a slightly lower score w.r.t. what is reported in the paper. Still, they can be both used as a baseline for our analysis.

4.2.1.2 Attack performances

We randomly sample 500 from the 15K malware set introduced in Section 4.2.1.1 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 4.4 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

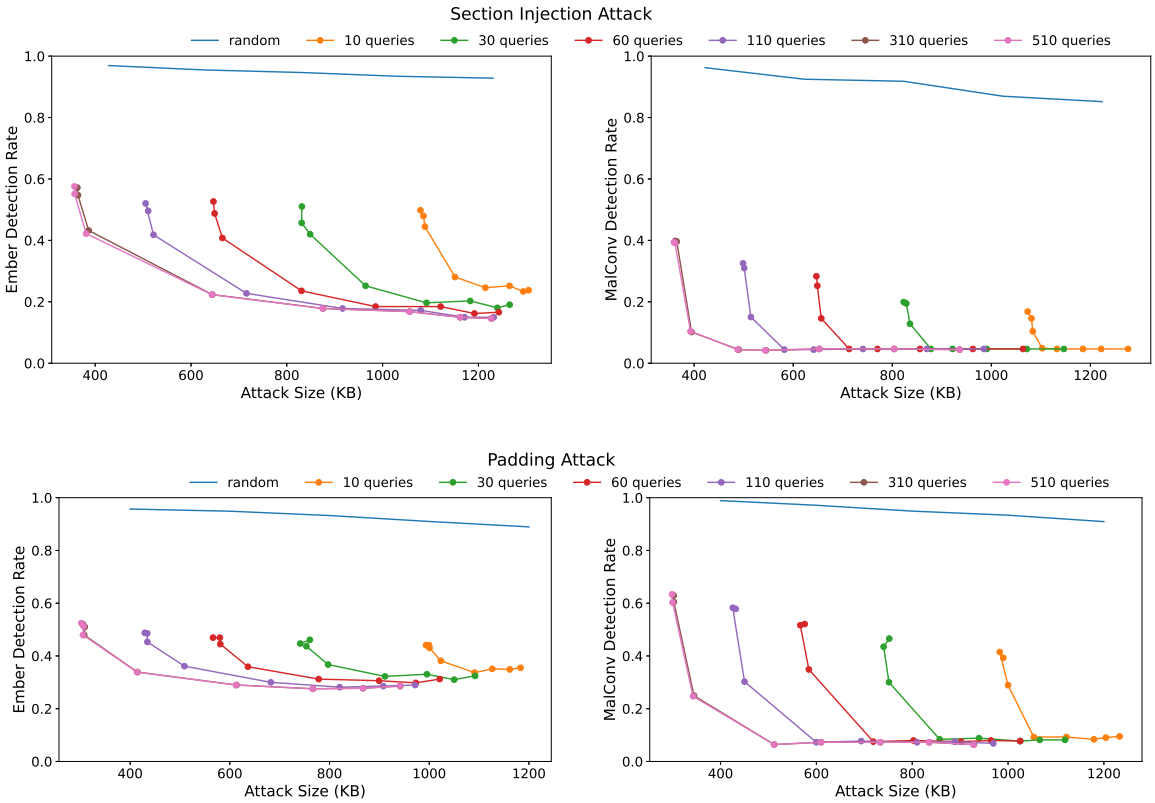


Figure 4.4: Padding and section injection attack performances for $\lambda \in \{10^{-i}\}_{i=3}^{9}$, using 500 malware samples as input.

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 EMBER 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

	Soft-Label GAMMA Q = 500		**Hard-Label GAMMA Q = 30**		**Hard-Label GAMMA Q = 500**	
Padding	*EMBER*	*MalConv*	*EMBER*	*MalConv*	*EMBER*	*MalConv*
$\lambda = 10^{-9}$	28%/941 KB	6%/927 KB	35% 945 KB	6% 835 KB	6% 835 KB	6% 835 KB
$\lambda = 10^{-6}$	33%/413 KB	6%/511KB				
$\lambda = 10^{-3}$	52%/302 KB	63%/298 KB				
Section Injection	*EMBER*	*MalConv*	*EMBER*	*MalConv*	*EMBER*	*MalConv*
$\lambda = 10^{-9}$	14%/1227 KB	4%/935 KB	6% 835 KB	6% 835 KB	6% 835 KB	6% 835 KB
$\lambda = 10^{-6}$	22%/643 KB	10% 487 KB				
$\lambda = 10^{-3}$	57%/356 KB	39%/359 KB				

Table 4.1: Comparison of soft-label and hard-label attacks, with different number of queries sent and values of λ.

padding attack.

4.2.1.3 Hard-label Attacks

Since GAMMA utilizes the score given by the classifier, it might not be suitable for an *hard-label attack*, that is an attacker setting where the target only replies with binary scores, hiding the real confidence. To guide the genetic algorithm, we penalize each detection by using *infinity* as value, to discard samples that are still seen as malware. Otherwise, the computed confidence is set to zero, and the regularization parameter is computed as usual. Optimizing this new quantity means finding evasive samples as small as possible.

We show aggregate results in Table 4.1, 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 four 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 remaining ones are used for optimizing only the size. On the contrary, the number of queries serves itself as a regularizer, since too few queries lead to larger adversarial payloads with low confidence, and numerous queries led to small payload whose score is higher.

4.2.1.4 Temporal analysis

From a temporal point of view, the complexity of GAMMA is dominated by the time spent querying the detector. The following table shows the mean elapsed time needed to compute one single query, for each attack and target.

	EMBER	**MalConv**
Padding	0.60 ± 0.24s	0.93 ± 0.33s
Section injection	0.60 ± 0.30s	0.86 ± 0.20s

Surprisingly, the sum of the time spent by the feature extraction phase and the prediction of EMBER is less than the time needed by the neural network to process all bytes.

4.2.1.5 Packing effect

Since these classifiers leverage only static features, it is reasonable to ask ourselves whether encoding the program content is already sufficient to evade detection, without applying all techniques we have introduced in Section 3.2. *Packing* is a technique used to reduce the size of an executable, by applying a compression, encryption or simply some encoding. As the effect of a packer completely changes the program representation on disk, it has been extensively used by malware authors in trying to hide and increase the difficulty of reverse-engineering.

In this context, we apply the famous packer *UPX* [93] to 1000 malware and 1000 goodware programs, and test the evasion rate for both MalConv and EMBER.

The effectiveness of applying UPX is shown in Figure 4.5. Each box-plot shows the distribution of the classifier confidence $f(\boldsymbol{x})$ on the malicious class. If $f(\boldsymbol{x}) \geq \theta$, being θ the decision threshold (solid red line), the sample is classified as malware. Packing increases the probability that a sample is classified as malware by both models. Both detectors attribute a malicious score when the sample is packed, as it can be clearly seen 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.

From these results, we believe that application of packing techniques is seen as a malicious trait by detectors. This might be caused by the abundance of packed malware inside the training set [8], opposed to the scarcity of packed goodware. As a result, models trained on such data might possess a bias that makes them wrongly assume that a sample is malicious only because it is packed. Also, given enough samples packed with a technique, the learning algorithm should be able to capture the signature left by the packer itself inside the packed program. For instance, the UPX packer creates two executable sections called

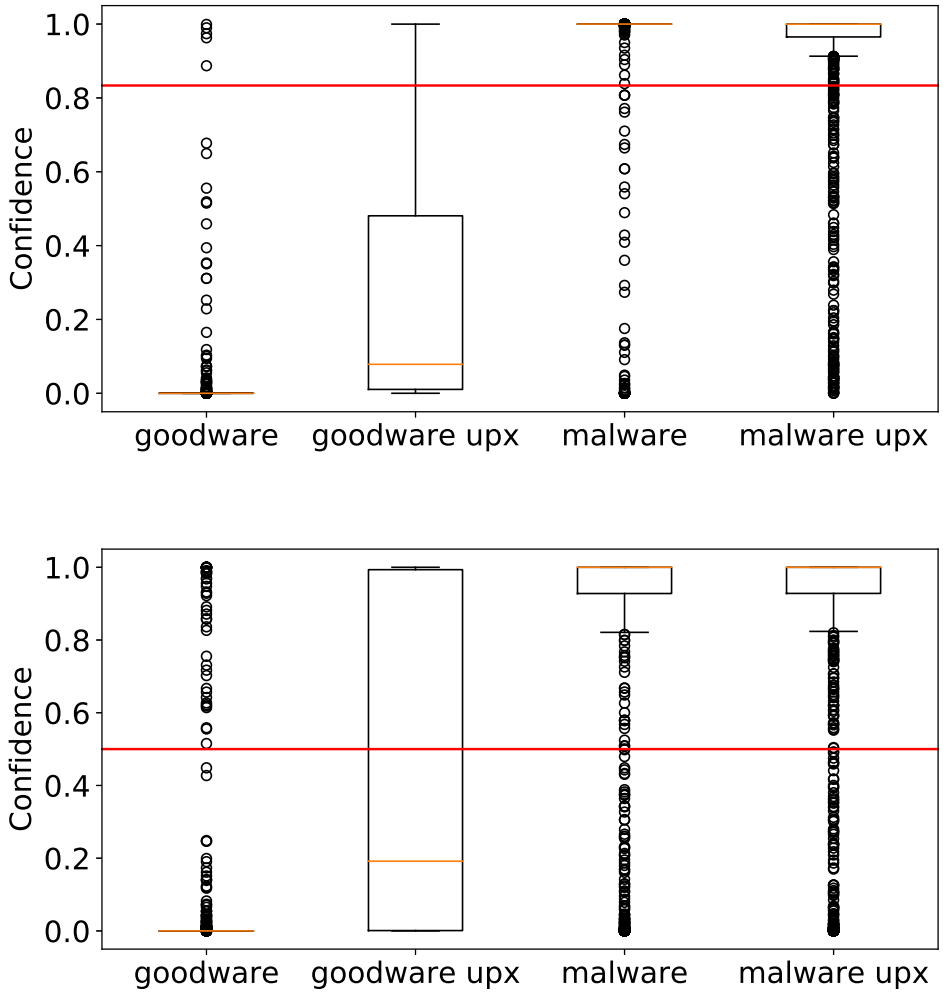


Figure 4.5: Effect of UPX packing on EMBER (top) and MalConv (bottom).

UPX0 and *UPX1*, that contain the extraction code and the original compressed program. We believe that evasion through packing techniques should more likely to be achieved by unseen packers, i.e. custom solutions developed by malware authors themselves.

4.2.1.6 Evaluation on Commercial Products

We are interested in understanding the effect of our methodology evaluated on commercial detectors. In this context, we are not interested in packing the input samples, as we believe that these methods should detect a threat even if the difference between the two versions is a small appended payload. The mutations 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 transformation can pose a threat to other antivirus programs.

We expected that most of commercial solutions should not be affected by such attacks, and we relied on the responses retrieved by VirusTotal [1], an online interface for many threat detectors. Such a service offers an API that can be used for querying the system, by uploading samples. We tested the performance of our attack by sending 200 malware samples, before and after attaching the adversarial payload to the sample, optimized against the EMBER classifier. As a baseline, we performed a test using the same data with a random payload of 50 KB attached. The following table shows how many malware are detected as such by the antivirus programs hosted on VirusTotal (70 in total).

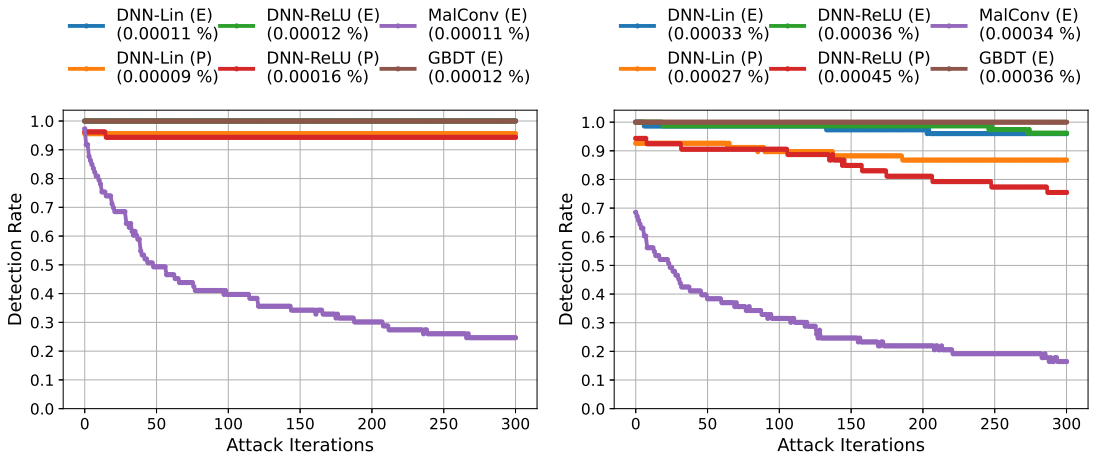
	Malware	**Random**	**Sect. Injection**
VT	46.56 ± 12.40	40.80 ± 12.40	34.50 ± 12.63

On average, both the random and the adversarial attack can decrease the number of detections, and the latter outperforms the former. It is clear that some of detectors, hosted on the remote service, manifest a weakness against small perturbations as well as the statistical algorithms we used for our analysis, even if the attack is not directly crafted against them.

4.2.2 Black-box query attacks

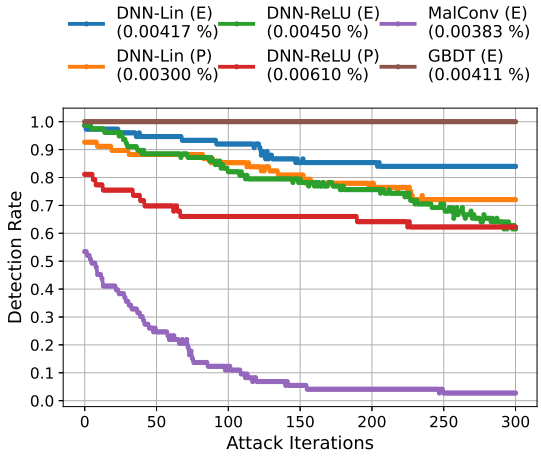
The query attacks behave worse than their white-box counterparts, as shown in Figure 4.6. We report the Detection Rate at each step of the black-box optimizer. For computing the Detection Rate, we used the threshold with 1% False Positive Rate. The number near the name of the classifier represents the size of the adversarial payload w.r.t. the mean file size of out malware test set. 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 3.4 (on page 34), 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).

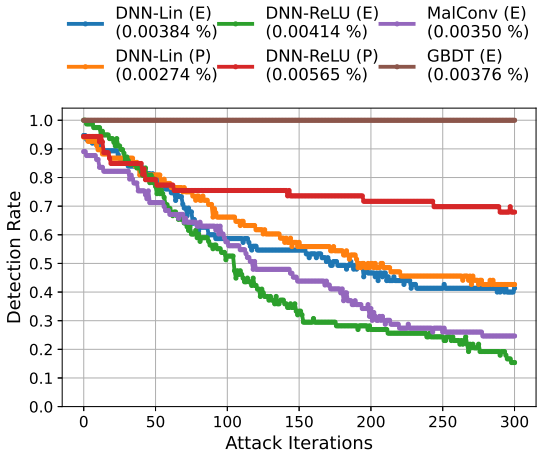


(a) Partial DOS

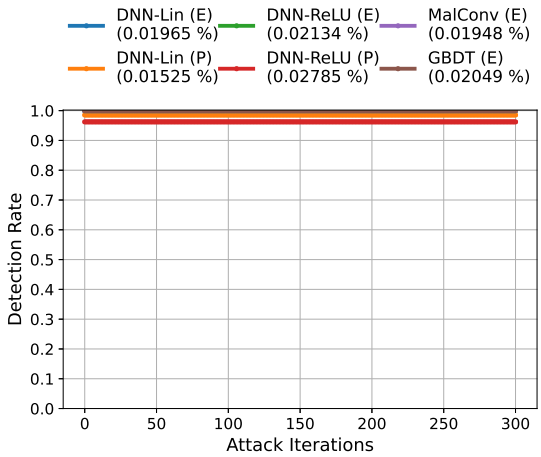
(b) Full DOS



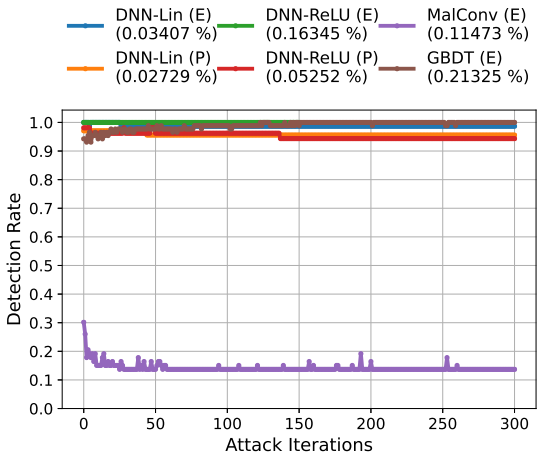
(c) Extend



(d) Shift



(e) Padding



(f) Gamma

Figure 4.6: The black-box query attacks against all models.

Since here we use a different threshold and a different setting than the experiments conducted in Section 4.2 (on page 51), the performance of GAMMA are different. It is effective against MalConv, but using such a low threshold for the GBDT, this attack is now not effective against the tree model. Previously, we have considered the standard detection, where the original threshold [8] is used (that is 0.82), and the addition of content taken from `.rdata`, for a maximum of 2.5 MB. Here, we have considered content harvested from `.data` sections for a maximum of 1 MB. We set the regularization parameter $\lambda = 10^{-5}$ to match the results shown in Section 4.2. This strategy is ineffective against the other networks since it leverage content appending, and, as discussed in Section 4.1 (on page 47), it is useless against the network proposed by Coull et al. [39].

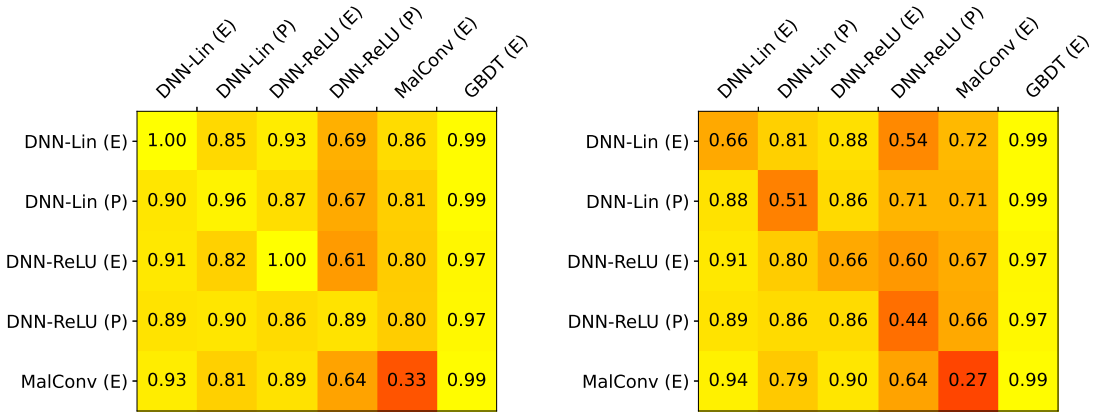
4.2.3 Black-box Transfer Attacks

Analyzing the behavior of classifiers against black-box transfer attacks is crucial, since attackers 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 examples, which we jokingly call *EXEmples*, crafted for the white-box attacks, and we test them against all other models.

In general, these transfer attacks are not effective, as clearly highlighted by Figure 4.7, but they still pose interesting results: the rows of each matrix show the model used for computing the adversarial EXEmples, while the columns show the model used for testing the transfer. The diagonal highlights the results of the white-box attack. For computing the Detection Rate, we used the threshold with 1% False Positive Rate. 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.

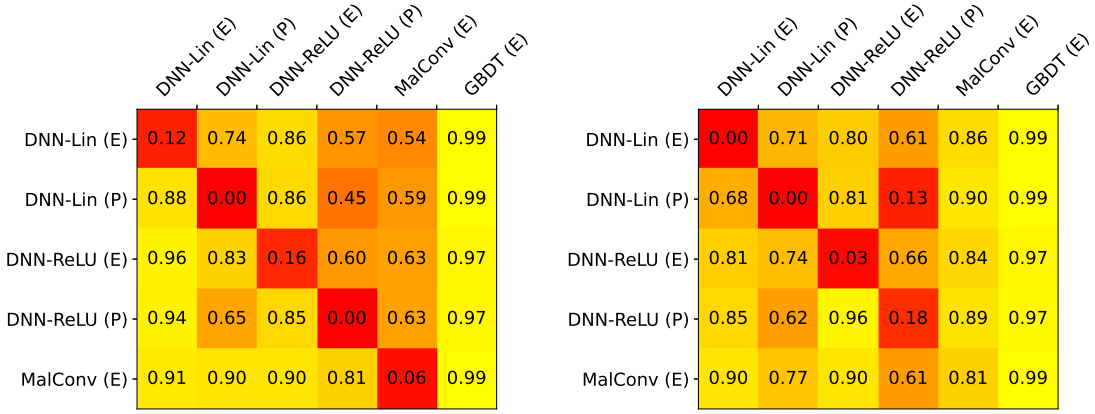
These attacks are not sufficient to subvert detection, but they highlight an interesting trends between models. This result might be explained by the non-linearity imposed by the ReLU activation functions, leading to many different local minima or maxima that are exploited by transfer samples. This effect 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 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 others.



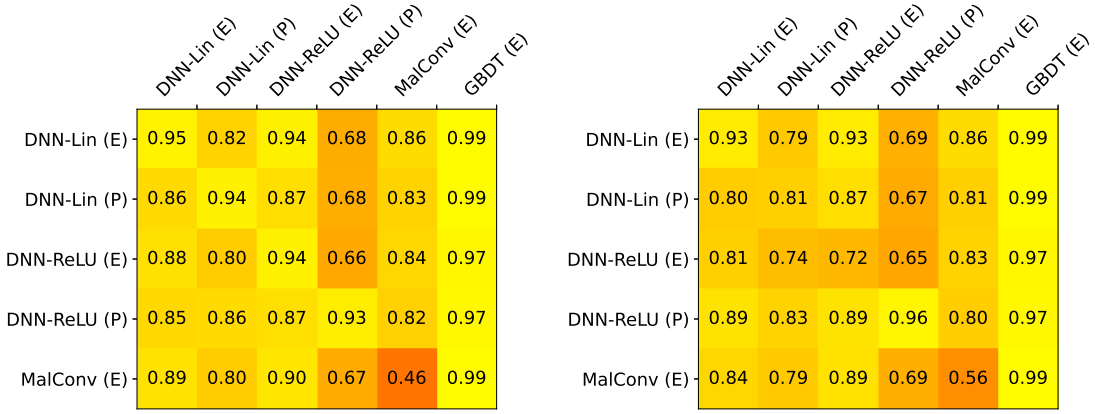
(a) Partial DOS

(b) Full DOS



(c) Extend

(d) Shift



(e) Padding

(f) FGSM

Figure 4.7: Comparisons of transfer attacks, expressed using the detection rate.

		C	γ	$avg(A)$	σ
Token-based	Naive Bayes	/	/	54.2%	1.0%
	Random Forest	/	/	87.3%	0.7%
	Linear SVM	19.30	/	80.5%	1.4%
	Gaussian SVM	278.25	0.013	93.1%	0.9%
SQLiGoT	Dir. Prop.	4.64	0.26	99.85%	0.07%
	Undir. Prop.	2.15	0.71	99.10%	0.2%
	Dir. Unprop.	2.15	0.26	99.74%	0.1%
	Undir. Unprop.	2.15	0.26	98.89%	0.2%

Table 4.2: WAF training phase results.

4.2.4 Mutational Fuzzing Results

We perform benchmark experiments to assess the detection rates of the WAFs introduced in Section 2.2.2. We trained them using a 5-fold cross-validation with 20,000 sane queries and 20,000 injections, and we used 15% of the queries for the validation set, and we used a portion of the test set for computing the benchmark in Table 4.3 (8000 benign and injection queries). This dataset is open-source and freely available online [94]. For both the token and graph based detectors, we coded our experiment using *scikit-learn* [95], which is a Python library containing already implemented machine learning algorithms. As regards the token-based classifiers, after the feature extraction phase the number of samples dropped to 768 benign and 7,963 injection queries. The tokenization method is basically an aggregation method: only a subset of all symbols are taken into account. The dataset is unbalanced, as the variety of sane queries is outnumbered by the variety of SQL injections. To address this issue, we set up *scikit-learn* accordingly, by using a loss function that takes into account the class imbalance [96]. Table 4.2 shows the results of the training phase where (i) C is the regularization parameter [97] that controls the stability of the solution, (ii) γ is the kernel parameter (only for the gaussian SVM) [98,99], and (iii) $avg(A)$ and σ are the average and standard deviation of the accuracy computed during the cross-validation phase over the validation set. The same compression happens for the graph-based classifiers: the algorithms that used directed graph reduces the dataset to 3216 benign and 12,659 injection queries, while the usage of undirected graph reduces the dataset to 3268 benign and 12,682 injection queries.

Table 4.3 shows the results of our experiment. The code used for producing these experiments is open-source [100] and available online. We evaluate the performance of each classifier by accounting three different metrics: (i) *accuracy*, (ii) *recall*, and (iii) *precision*. We denote the true positives as TP, true negatives as TN, false positives as FP and false negatives as FN. Accuracy is computed as $A = \frac{TP+TN}{TP+TN+FP+FN}$, recall is computed as

		A	R	P
ModSecurity CSR	Paranoia 1/2	86.10%	86.10%	100%
	Paranoia 3/4	91.85%	91.85%	100%
	Paranoia 5	96.46%	96.46%	100%
WAF-Brain	RNN	98.27%	96.73	99.8%
Token-based	Naive Bayes	50.16%	98.71%	50.08%
	Random forest	98.33%	98.33%	100%
	Linear SVM	98.75%	98.76%	100%
	Gaussian SVM	97.82%	97.82%	100%
SQLiGoT	Dir. Prop.	90.61%	97.30%	85.82%
	Undir. Prop.	96.38%	97.31%	95.54%
	Dir. Unprop.	90.52%	97.12%	85.80%
	Undir. Unprop.	96.25%	97.05%	95.53%

Table 4.3: Benchmark table.

$R = \frac{TP}{TP+FN}$ and precision is computed as $P = \frac{TP}{TP+FP}$.

Accuracy measures how many samples have been correctly classified, i.e., a sane query classified as sane or an injected query classified as malicious. Recall measures how good the classifier is at identifying samples from the relevant class, in this case the injection payloads. Scoring a high recall value means that the classifier labeled most of the real positives in the dataset as positives. Precision measures how many of the samples classified as relevant are actually relevant.

Since the Naive Bayes algorithm tries to discriminate between input classes by considering each variable independent one to another, it misses the real structure of the SQL syntax. Hence, it cannot properly capture the complexity of the problem. All other classifiers may be compared with different levels of paranoia offered by ModSecurity [101], showing their effectiveness as WAFs. WAF-Brain results are comparable to what the author claims on his GitHub repository.

The experiments were performed on a DigitalOcean [102] droplet VM with 6 CPUs and 16GB of RAM. For a baseline comparison we use an *unguided* mutational fuzzer. The unguided fuzzer randomly applies the mutation operators of Section 3.2 (page 27). Moreover, we execute 100 instances of the unguided fuzzer on each classifier. Then, we compare a single run of WAF-A-MoLE against the best payload generated by the 100 unguided instances over time. Both WAF-A-MoLE and unguided fuzzers are configured to start from the payload `admin' OR 1=1#`, initially detected with 100% confidence by each classifier.

4.2.4.1 Assessment results

Figure 4.8 and 4.9 show the evolution of the confidence score for each classifier. In each plot, we compare the best sample obtained by WAF-A-MoLE (solid line) and the best sample generated by all 100 processes of the *unguided* fuzzer (dashed line). The first group of plots (4.8) show the evolution of the confidence scores against the number of mutation rounds. The second group (Figure 4.9), shows the confidence score over the actual time of computation. In particular, we show the first 10 seconds of computation. Since some scores quickly degrade in the first milliseconds of computation, we report the x axis in log scale. Our experiments highlight a few facts that we discuss below.

4.2.4.2 Feature choice matters

As explained in Section 2.2.2 (on page 12), all considered classifiers are based on syntactic features. However, different feature set change the robustness of a classifier. For instance, WAF-Brain quickly lost confidence when the payload mutated, because WAF-Brain is trained from uninterpreted, fixed-length sequences of characters and our mutation operators can enlarge a payload beyond the adequacy of the length assumed by WAF-Brain. Also, Token-based classifiers do not perform well against mutations. The reason is that malicious and benign payloads overlap in the feature space.

All SQLiGoT versions showed to be robust against the unguided approach. These classifiers use the SVM algorithm as some of the token based classifiers, but their feature set imposes more structure inside the feature representation. Hence, random mutations have a negligible probability to evade them. Instead, since WAF-A-MoLE relies on a guided strategy, it can effectively craft adversarial examples, although more effort is needed.

4.2.4.3 Finding adversarial examples is non-trivial

SQLiGoT classifiers resist the unguided evaluation as it is unlikely that a mutation can move the sample away from a plateau region where the confidence of being a SQL injection is high. The main reasons are:

(i) SQLiGoT considered a large number of tokens (so reducing the collision problem that affects other classifiers, since the compression factor applied by the feature extractor is lower);

(ii) the structure of the feature vector is inherently redundant, i.e., each pair of adjacent variables describe the same token;

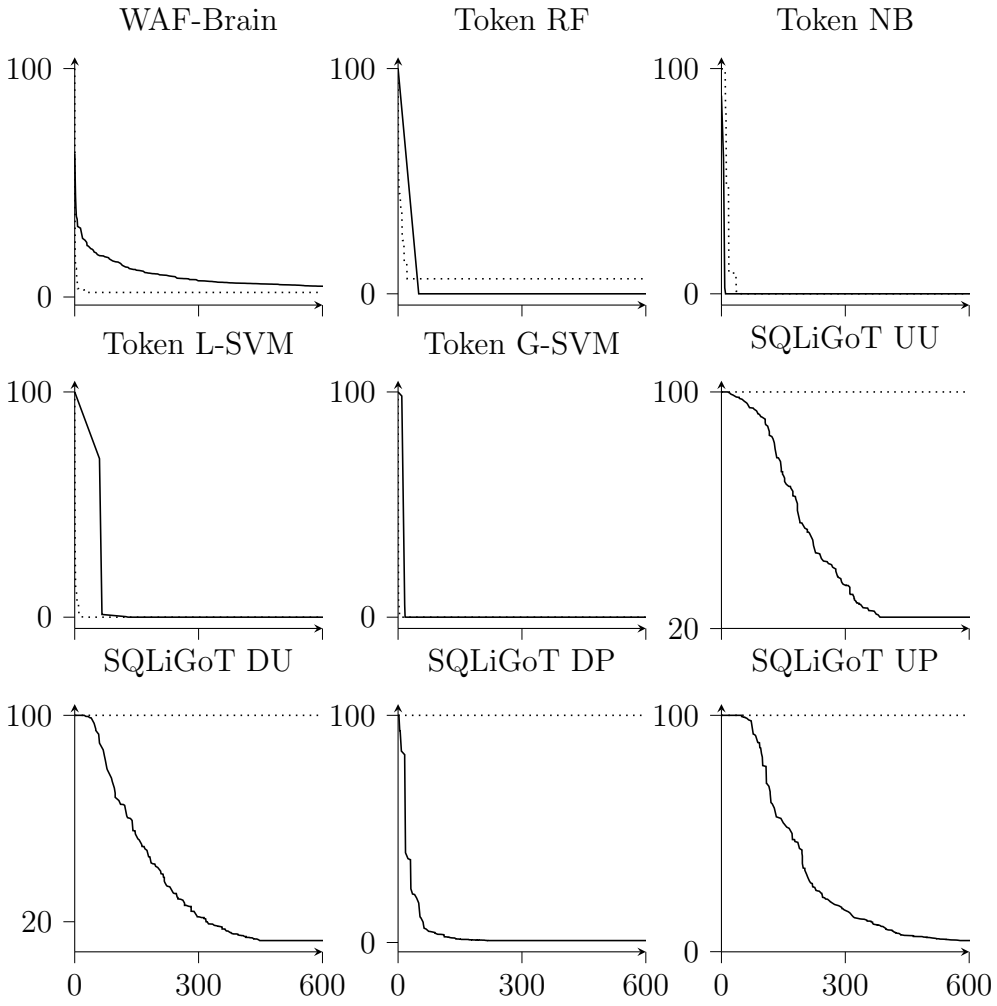


Figure 4.8: *Guided* (solid) vs. *unguided* (dotted) search strategies applied to initial payload `admin' OR 1=1#`, divided by iterations.

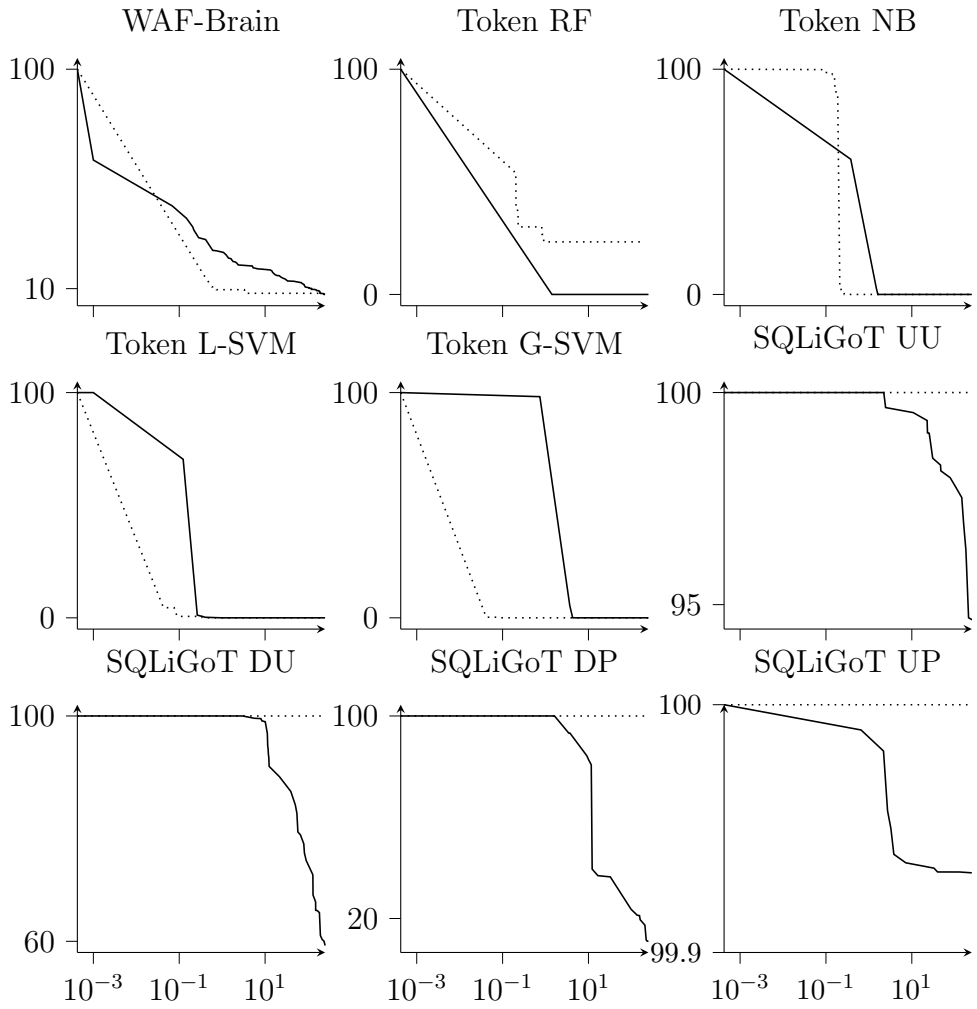


Figure 4.9: *Guided* (solid) vs. *unguided* (dotted) search strategies applied to initial payload `admin' OR 1=1#`, w.r.t. the time elapsed by the fuzzer.

(iii) the models are regularized, hence the decision function is smoother between input points and it manages to generalize over new samples.

4.2.4.4 WAF-A-MoLE effectively evades WAFs

Moving randomly in the input space is not an effective strategy. WAF-A-MoLE finds adversarial samples by leveraging on hints given by classifier outputs. The guided approach accomplishes what the unguided approach failed to, by moving points away from plateaus and putting them in regions of low confidence of being recognized as SQL injection.

Moreover, among the SQLiGoT classifiers, the *undirected unproportional* is the most resilient variant. Recalling the definition of the algorithm [10], the feature extractor assigns uniform weights to tokens in the same window instead of balancing the score w.r.t. the distance of the current token. Hence, the classifier gains some invariance over the sequence of extracted tokens, making it more robust to adversarial noise.

Chapter 5

Conclusions and Future Work

ATTACKING strategies proposed in this thesis focus on static detectors, and it is unlikely that the very same strategies can defeat dynamic classifiers too. This limitation is posed by the nature of *practical manipulation*, that exploit syntactical alterations of payloads. Extending these ideas to runtime events is not straightforward and will be the subject of further work.

Another limitation of our current practical manipulations is their byte-based nature. An interesting research direction would be to extend our manipulations to handle some structured information, that is, being able to alter sequence of bytes as "atomic groups", using some syntactic representation of the underlying file format, by leveraging C-like structure descriptions consisting of fields of different sizes. For instance, a classifier may use data taken from the *Import Address Table*, and byte-level modifications would not be enough in that case. Another example is changing one or more machine instructions with different, yet equivalent, instructions, constrained by the fact that the two byte sequences must be of the same size and no jumps should occur in the middle of such instructions. The latter, by itself, is an undecidable problem, in general, that could probably be addressed in practice by using some heuristics. These manipulations are akin to *binary rewriting techniques* [103], that allow altering the machine code of a binary, by adding, for instance, new functionalities. In this thesis, we avoided applying such mutations since we wanted to address only the structure of a program while keeping its code intact.

From a defender point of view, there are a few options for discarding or partially eliminating the adversarial noise we introduce inside samples, since our content-injection attacks comply with the specific constraints posed by the format. For instance, the injected content does not interfere with file alignment, specified in the file header, or the sample will be considered corrupted by the operating system. Also, as far as we know, all defensive techniques that have been proposed in the state-of-the-artare related to the domain

of images [104–107], but none of them are robust to adversarial examples [58, 108, 109]. Also, none of those defensive techniques have been applied in the security domains we have analyzed. Most defenses rely on the concept of *adversarial retraining*, that is including adversarial examples inside the training pipeline and repeating the process of fitting and computing attacks until convergence, or they apply *gradient obfuscation* techniques to produce noisy decision boundaries, where gradients vary their direction abruptly.

However, some partial countermeasures can be applied. Samples produced by *Partial DOS*, *Full DOS*, and *Extend* strategies (Section 3.2, page 27) can be easily sanitized by reverting the DOS header and DOS stub to their "default" versions.

The *Shift* manipulation can be detected by looking at the physical offsets of the sections inside samples, where the content has been moved inside the files. By inspecting such content, the random-like adversarial noise may raise suspicion, and those parts can be "stripped away" by overwriting them with, for instance, zeros.

All *Padding* techniques can be similarly detected by looking at where the last section ends. Every byte after that is considered as an overlay, and it can be manually analyzed.

The *Section injection* attack can be detected by analyzing the number of sections included inside a binary. Nevertheless, the attacker can also try to aggregate more benign content into a single section and see how the attack performance varies.

Speaking of SQL payloads, WAF-A-MoLE might not be able to fully explore the space by applying only random mutations. We did not try more sophisticated content injection techniques since we already scored evasion.

The main contribution of this thesis is RAMEn, a lightweight formalization that encapsulates all needs of attackers, with the practical manipulations applicable in the domain of choice and with all constraints expressed as a penalty term inside the optimization process.

We defined and applied new practical manipulations, crafted for the Windows malware and SQL injection detection domains. Such manipulations do not require a sandbox to validate the results, since they are semantic-invariant by design. We took into account state-of-the-art classifiers, presenting successful evasive adversarial examples against them, in both white-box and black-box settings.

In the Windows malware domain, we introduced three new attacks: two of them focus on exploiting the file format, by carving space inside programs to insert adversarial noise, and other one assembles such noise with portions of benign programs. In the white-box scenario, 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. In the black-box scenario, we showed how the regularization parameter added to the optimizer can modulate the size of the resulting adversarial attack, achieving the constraint of minimality that we have imposed on it.

We tested the performance of transfer attacks, showing how an attacker can take advantage of using only surrogate models they own. Then, we showed how the white-box attacks can be converted to black-box ones, by changing the optimizer used for altering the samples.

In the SQL domain, we showed how our mutational fuzzing approach can decrease the confidence of state of the art SQL injection detectors.

We showed how all attacks proposed in the state of the art can be encoded in our formalization without loss of generality, and implemented accordingly.

All Windows practical manipulations are available as a part of an open-source project named SecML Malware [89], while the SQL ones are provided with the code of WAF-A-MoLE [22, 100].

Bibliography

[1] VirusTotal. File statistics during last 7 days. https://www.virustotal.com/it/statistics/. (visited on 28 October 2020).

[2] OWASP. OWASP Top Ten. https://owasp.org/www-project-top-ten/. (visited on 28 October 2020).

[3] Joshua Saxe and Konstantin 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.

[4] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.

[5] William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin 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 Engineering and Applied Computing (WorldComp), 2016.

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

[7] Inigo Incer, Michael Theodorides, Sadia Afroz, and David 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.

[8] H. S. Anderson and P. Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*, April 2018.

[9] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[10] Debabrata Kar, Suvasini Panigrahi, and Srikanth Sundararajan. SQLiGoT: Detecting sql injection attacks using graph of tokens and svm. *Computers & Security*, 60:206–225, 2016.

[11] Anamika Joshi and V Geetha. SQL injection detection using machine learning. In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 1111–1115. IEEE, 2014.

[12] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[13] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[14] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583*, 2019.

[15] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. *arXiv preprint arXiv:1803.04173*, 2018.

[16] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Adversarial examples on discrete sequences for beating whole-binary malware detection. *arXiv preprint arXiv:1802.04528*, 2018.

[17] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo Rodosek. ARMED: How automatic malware modifications can evade static detection? In *2019 5th International Conference on Information Management (ICIM)*, pages 20–27. IEEE, 2019.

[18] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo Rodosek. AIMED: Evolving malware with genetic programming to evade detection. In *2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2019.

[19] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading machine learning malware detection. *Black Hat*, 2017.

[20] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 490–510. Springer, 2018.

[21] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983*, 2017.

[22] Luca Demetrio, Andrea Valenza, Gabriele Costa, and Giovanni Lagorio. WAF-A-MoLE: evading web application firewalls through adversarial machine learning. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1745–1752, 2020.

[23] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Functionality-preserving black-box optimization of adversarial windows malware, 2020.

[24] Luca Demetrio, Scott E. Coull, Battista Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial EXEmples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection, 2020.

[25] Raphael Labaca Castro, Corinna Schmitt, and Gabi 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.

[26] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. Automatic generation of adversarial examples for interpreting malware classifiers. *arXiv preprint arXiv:2003.03100*, 2020.

[27] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv preprint arXiv:1802.04528*, 2018.

[28] Octavian Suciu, Scott E Coull, and Jeffrey Johns. Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 8–14. IEEE, 2019.

[29] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. *arXiv preprint arXiv:1911.02142*, 2019.

[30] W Xu, Y Qi, and D Evans. Automatically evading classifiers: A case study on PDF malware classifiers. NDSS, 2016.

[31] Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com/, April 2017.

[32] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *arXiv preprint arXiv:1703.01365*, 2017.

[33] Avast Antivirus. AI and machine learning. https://www.avast.com/technology/ai-and-machine-learning. (visited on 28 October 2020).

[34] Kaspersky. Machine learning methods for malware detection. https://media.kaspersky.com/en/enterprise-security/Kaspersky-Lab-Whitepaper-Machine-Learning.pdf. (visited on 28 October 2020).

[35] Sophos. Intercept X Deep Learning. https://www.sophos.com/en-us/medialibrary/PDFs/factsheets/sophos-intercept-x-deep-learning-dsna.pdf. (visited on 28 October 2020).

[36] PaloAlto Networks. Artificial intelligence and machine learning in the security operations center. https://www.paloaltonetworks.com/resources/techbriefs/AI-and-ML-in-the-SOC. (visited on 28 October 2020).

[37] Microsoft. PE Format. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. (visited on 28 October 2020).

[38] Scott E. Coull Jeffrey Johns. Representation learning for malware classification. https://www.camlis.org/2017/jeffreyjohns. (visited on 28 October 2020).

[39] Scott E Coull and Christopher 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.

[40] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[41] BBVA. WAF-brain, the clever and efficient firewall for the web. https://github.com/BBVA/waf-brain. (visited on 28 October 2020).

[42] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[43] Ryohei Komiya, Incheon Paik, and Masayuki Hisada. Classification of malicious web code by machine learning. In *2011 3rd International Conference on Awareness Science and Technology (iCAST)*, pages 406–411. IEEE, 2011.

[44] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 16–25. ACM, 2006.

[45] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. Jun 2012.

[46] Battista Biggio, Luca Didaci, Giorgio Fumera, and Fabio Roli. Poisoning attacks to compromise face templates. page 1–7. IEEE, Jun 2013.

[47] Yuntao Liu, Yang Xie, and Ankur Srivastava. Neural trojans. *arXiv:1710.00942 [cs]*, Oct 2017. arXiv: 1710.00942.

[48] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv:1712.05526 [cs]*, Dec 2017.

[49] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. Exploring backdoor poisoning attacks against malware classifiers. Mar 2020.

[50] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. page 601–618, 2016.

[51] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1322–1333, 2015.

[52] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[53] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[54] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[55] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 372–387. IEEE, 2016.

[56] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. A general framework for adversarial examples with objectives. *ACM Transactions on Privacy and Security*, 22(3):1–30, Jul 2019. arXiv: 1801.00349.

[57] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 3–14. ACM, 2017.

[58] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[59] Guoming Zhang, Chen Yan, Xiaoyu Ji, Taimin Zhang, Tianchen Zhang, and Wenyuan Xu. Dolphinatack: Inaudible voice commands. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*, page 103–117, 2017. arXiv: 1708.09537.

[60] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner, and Wenchao Zhou. Hidden voice commands. page 513–530, 2016.

[61] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. *arXiv:1801.01944 [cs]*, Jan 2018. arXiv: 1801.01944.

[62] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[63] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. page 18.

[64] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. On the intriguing connections of regularization, input gradients and transferability of evasion and poisoning attacks. *arXiv preprint arXiv:1809.02861*, 2018.

[65] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv:1605.07277 [cs]*, May 2016.

[66] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. *arXiv preprint arXiv:1804.08598*, 2018.

[67] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. ZOO: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security - AISec '17*, 2017.

[68] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[69] Mahmood Sharif, Keane Lucas, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. Optimization-guided binary diversification to mislead neural networks for malware detection. *arXiv preprint arXiv:1912.09064*, 2019.

[70] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[71] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer, 2017.

[72] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. DREBIN: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[73] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android HIV: A study of repackaging malware for evading machine-learning detection. *arXiv preprint arXiv:1808.04218*, 2018.

[74] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017.

[75] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Trans. Dependable and Secure Computing*, In press.

[76] Pavel Laskov et al. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE symposium on security and privacy*, pages 197–211. IEEE, 2014.

[77] Charles Smutz and Angelos Stavrou. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th annual computer security applications conference*, pages 239–248. ACM, 2012.

[78] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 15–26. ACM, 2017.

[79] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 3381–3387. IEEE, 2008.

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

[81] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems*, pages 971–980, 2017.

[82] Leonard Kaufmann and Peter Rousseeuw. Clustering by means of medoids. *Data Analysis based on the L1-Norm and Related Methods*, pages 405–416, 01 1987.

[83] Cristian I Pinzon, Juan F De Paz, Alvaro Herrero, Emilio Corchado, Javier Bajo, and Juan M Corchado. idmas-sql: intrusion detection based on mas to detect and block sql injection through data mining. *Information Sciences*, 231:15–31, 2013.

[84] Abdelhamid Makiou, Youcef Begriche, and Ahmed Serhrouchni. Improving web application firewalls to detect advanced sql injection attacks. In *2014 10th International Conference on Information Assurance and Security*, pages 35–40. IEEE, 2014.

[85] Melvin Earl Maron. Automatic indexing: an experimental inquiry. *Journal of the ACM (JACM)*, 8(3):404–417, 1961.

[86] Parul Garg. Fuzzing – Mutation vs. Generation. https://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/. [Online; accessed 29-June-2019].

[87] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Mutation-based fuzzing. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-05-21 19:57:59+02:00.

[88] Jeremy D'Hoinne, Adam Hils, and Claudio Neiva. Magic quadrant for web application firewalls. Technical report, Gartner, Inc., August 2017.

[89] Luca Demetrio. SecML Malware plugin. https://github.com/zangobot/secml_malware. (visited on 28 October 2020).

[90] Marco Melis, Ambra Demontis, Maura Pintor, Angelo Sotgiu, and Battista Biggio. secml: A python library for secure and explainable machine learning, 2019.

[91] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[92] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[93] The UPX Team. UPX, the Ultimate Packer for eXecutables. https://upx.github.io. (visited on 28 October 2020).

[94] Luca Demetrio, Andrea Valenza, Gabriele Costa, and Giovanni Lagorio. WAF-A-MoLE Dataset. https://github.com/zangobot/wafamole_dataset.

[95] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[96] Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M Buhmann. The balanced accuracy and its posterior distribution. In *2010 20th International Conference on Pattern Recognition*, pages 3121–3124. IEEE, 2010.

[97] Andrey Nikolayevich Tikhonov. On the stability of inverse problems. In *Dokl. Akad. Nauk SSSR*, volume 39, pages 195–198, 1943.

[98] Mark A Aizerman. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and remote control*, 25:821–837, 1964.

[99] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008.

[100] Andrea Valenza, Luca Demetrio, Gabriele Costa, and Giovanni Lagorio. WAF-A-MoLE. https://github.com/AvalZ/waf-a-mole.

[101] TrustWave SpiderLabs. ModSecurity, Open Source Web Application Firewall. https://www.modsecurity.org/. (visited on 28 October 2020).

[102] DigitalOcean LLC. DigitalOcean. https://www.digitalocean.com/. (visited on 28 October 2020).

[103] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3):1–37, 2019.

[104] Dongxian Wu, Shu-Tao Xia, and Yisen Wang. Adversarial weight perturbation helps robust generalization. *Advances in Neural Information Processing Systems*, 33, 2020.

[105] Yair Carmon, Aditi Raghunathan, Ludwig Schmidt, John C Duchi, and Percy S Liang. Unlabeled data improves adversarial robustness. In *Advances in Neural Information Processing Systems*, pages 11192–11203, 2019.

[106] Vikash Sehwag, Shiqi Wang, Prateek Mittal, and Suman Jana. Hydra: Pruning adversarially robust neural networks. *Advances in Neural Information Processing Systems*, 33, 2020.

[107] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597. IEEE, 2016.

[108] Francesco Croce, Maksym Andriushchenko, Vikash Sehwag, Nicolas Flammarion, Mung Chiang, Prateek Mittal, and Matthias Hein. Robustbench: a standardized adversarial robustness benchmark. *arXiv preprint arXiv:2010.09670*, 2020.

[109] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International Conference on Machine Learning*, pages 274–283, 2018.

Ringraziamenti

Ci sono moltissime persone che voglio ringraziare per il compimento di questa impresa lunga tre anni e vissuta dal primo all'ultimo giorno. Sono stati tre anni di crescita mentale, emotiva, sociale, scientifica, artistica e molto molto altro. Chi mi conosce e chi mi ha seguito fino ad ora ha vissuto in prima persona tutti questi cambiamenti: momenti di gioia e dolore, momenti in cui ho pensato di mollare tutto e ricominciare da capo, momenti in cui l'adrenalina si conteneva a stento dentro questa pelle e tanto altro ancora.

Inizio con il ringraziare Giovanni Lagorio, il mio relatore, senza il quale non sarei riuscito a concludere questo percorso. E non è ancora uscito Metroid Prime 4, ma ti terrò aggiornato nel futuro.

Ringrazio Battista Biggio, per il tutto supporto e le dritte che mi ha fornito, per aver sopportato le mie mail, i miei messaggi Skype e altro ancora (dai che tra poco arrivo a Cagliari). E si, concordo sempre di più che questo lavoro "non sia per i deboli di cuore"!

Ringrazio Alessandro Armando per tutte le riunioni e le domande a brucia-pelo che mi hanno sempre spronato a riflettere e ritornare sui miei passi.

Ringrazio Fabio Roli per tutta la gentilezza che mi ha mostrato nell'avvicinarmi al mondo dell'Adversarial, ma anche per numerosi i consigli e correzioni sui lavori fatti insieme.

Ringrazio mia mamma, Daniela, per le nostre piccole tradizioni come le colazioni genovesi e le passeggiate, per le lasagne, per l'ispirazione continua che mi fornisce. Sei un gigante e non te lo scordare mai. E si, ti insegnerò altri passi di danza, basta che tu me lo chieda.

Ringrazio mio papà Guerino, per essere il primo artista che ho incontrato nella mia vita, per tutti i momenti di condivisione, per tutti i momenti di bricolage in casa e perché sei una colonna di titanio. Ci sono cose che ti piegano, ma nulla ti spezza.

Ringrazio mio fratello Andrea, infinito universo di storie, racconti e personaggi che non si esauriscono mai, per le giocate fino a tardi, per gli spunti e per il sostegno.

Ringrazio Alessio, il mio coinquilino perché "cosa mai poteva andare storto?" , per tutte le cose nuove che mi ha fatto conoscere, dai canali YouTube ai libri ai giochi ai podcast, per

avermi sopportato in quarantena, per le nostre disavventure nell'era COVID e per tutti i piani di espansione casalinga.

Ringrazio Andrea, il sommo *Avalz*, per tutti i consigli, le giocate, le pizze, le chiamate su Discord, le missioni insieme, gli incitamenti, i meme ed il tuo essere unico.

Ringrazio Enrico, il mitico *enriquez*, per essere il saggio artista del gruppo, poche parole ma sempre buone. E sempre giuste.

Ringrazio Simone, il grande *packmad*, per essere un cuore d'oro sotto una corazza savonese, per tutti i consigli che mi hai dato e per tutti i momenti in cui mi sei stato vicino.

Ringrazio tutto il team ZenHack, con la quale ho giocato alle CTF di questi tre anni (purtroppo poche poche sfide di Adversarial, ma quelle poche...).

Ringrazio Giulia, per le carbonare improvvisate, per le stanze da rimettere a posto, per la carta da parati da sciogliere a colpi di insulti, per quella maratona di "American Horror Story" non ancora finita, per i gatti.

Ringrazio Simone, per solida amicizia fatta di poche parole ma molto nerdismo, molti dolci, molto sushi e molti giochi in VR. Ogni volta che ho bisogno di parlare di tecnologia, tu sei il primo a cui voglio rivolgere la parola.

Ringrazio Arianna, per tutte le cene e i pranzi telematici, per tutte le foto, video e canzoni che ci siamo scambiati, rendendo la distanza soltanto un limite fisico. Ma le amicizie vere non hanno limiti fisici.

Ringrazio Riccardo, perché abbiamo creato una sorta di jukebox vivente dentro la nostra chat, fatta anche di meme, di scienza e di spiegoni.

Ringrazio Stefano per la sua infinita dolcezza e conoscenza. Hai presente quando ti serve sapere qualcosa che mai avresti pensato? Ecco, lui lo sa.

Ringrazio Patrizio! Con il punto esclamativo! Perché è grosso! E perché è ancora più grosso! Vabbè avete capito: è una delle persone più spesse che conosca, sia fisicamente che mentalmente.

Ringrazio Alex, perché nonostante le poche parole, sento esistere un legame forte, fatto di eventi e di pochi confronti, ma molto profondi.

Ringrazio Simone e Giorgia perché parlare con loro genera in me una calma estrema, per "le cronache di Imperia" e per essere sempre presenti in ogni occasione, con questa loro aura rassicurante.

Ringrazio Fede, il mio video-maker di fiducia, per le idee da set, per i montaggi, per le future collaborazioni. Aspetto con ansia il tuo primo podcast o la tua prima produzione, sappilo.

Ringrazio Luca, perché in fondo in fondo siamo due piccoli "BORIS" che ascoltano musica hard-bass.

Ringrazio Max, perché nonostante la vita sia frenetica, il modo in cui la vivi e la descrivi la rende meno improvvisa e un po' più lenta. E il che non è un male, anzi!

Ringrazio Save, per il trash di qualità e perché ti devo ancora ridare il tuo cappello. Con un anno di ritardo. Ce la faremo!

Ringrazio Elenina, perché sei stata la persona che mi ha spinto in un momento di crisi nera. Grazie a te, ho preso una decisione molto importante. Grazie per tutto il tea, per le foto di Pelù e per essere un'amica fidata.

Ringrazio Gaurvi, because she's been there for me, when I needed most. Because we have our place for chatting, in front of a hot chocolate, full of cream and gossip. And tea. Lot of tea.

Ringrazio Kia, perché non solo è diventata la mamma più forte del mondo, ma perché è una delle persone più rassicuranti che conosca. E perché è anche una delle persone più folli che conosca. Ma proprio per questo ti voglio bene.

Ringrazio Maura, perché non solo saremo futuri colleghi, ma perché valanghe di nerdismo ci attendono! E di reti neurali rotte, o bizzarre, o tizi che suonano i bonghi mentre sei in riunione. Grazie per l'accuratissima mappa di Cagliari, grazie per le invenzioni last-minute e per tutto l'aiuto che mi stai fornendo.

Ringrazio Romina, la mia insegnante di tap. Dolcezza, professionalità, potenza, leggerezza... è impossibile descriverti in poco, ed è impossibile comunicare quanto tu mi abbia insegnato, quanto tu mi abbia fatto sentire felice e quanto ti sia grato. Grazie a te, sento di aver fatto un salto di qualità notevole e vedo il prossimo ulteriore miglioramento all'orizzonte.

Ringrazio Giulia, perché ci sei sempre stata, perché siamo andati a prenderci quei meravigliosi sesto e tredicesimo posto ai mondiali, perché ci capiamo e siamo fatti della stessa trama.

Ringrazio Giuggi, perché siamo due Potteriani convinti, perché sei creativa e pazza, perché mi diverto sempre un sacco quando usciamo.

Ringrazio Ali, perché siamo entrambi molto logici ma molto emotivi, perché siamo malati di serie tv e perché sei dolcissima.

Ringrazio Enzo, per tutte le risate che mi hai strappato, per i bicchieri di vino nella casa nuova, per le camminate e per avermi prestato quegli occhiali che sono volati con me a raggiungere il sesto posto ai mondiali. E li ho indossati tutto il tempo.

Ringrazio Simone, ringrazio Maier. Ringrazio una persona unica, con uno stile unico, con

questa tremenda forza artistica che non ha confini. Arte che si rinnova in ogni scatto, ogni movimento, ogni intenzione. Hai sbloccato qualcosa dentro di me, questa voglia di esplorare, voglia di fare ricerca nel movimento. E di questo ti sarò sempre grato.

Ringrazio Maela, per essere lo sguardo più presente e profondo che conosca, perché i tuoi movimenti sono imbevuti di calma ragionata, perché ballare con te mi permette di respirare e non abbandonarmi alla fretta.

Ringrazio Gio, perché sento molte direzioni diverse nella sua espressione artistica, per i suoi silenzi rumorosi e pieni di dettagli, per il tuo mondo che porti dietro agli occhi accesi.

Ringrazio Ermanno, perché anche se ci conosciamo da poco riesco a percepire un'affinità positiva. Siamo entrambi sonori, in un certo senso. E non vedo l'ora di continuare a collaborare con te, con foto, video e momenti leggeri.

Ringrazio Ciao, because he taught me so many secret drawing arts, because of the amazing "Ghost with a Gun", and for being one of the kindest and most talented artists I've ever met.

Ringrazio Lana, because she's literally the best pole-dancer I've ever known. Your talent, your art, your strength: you inspire me a lot to create. When I see one of your choreography, I feel a urge for creating something. I wish you the best of the best, my friend.

Ringrazio Noah, because I feel how much she believes in my art and my movements. Because she's one of the most humble person I know. Because she keeps creating and creating and creating. I can't wait to dance with you again. Thank you for "Dancers in lockdown": I keep feeling grateful for that.

Ringrazio Galen, because that day, with "River", you planted a seed of awareness, of who can I be as a dancer. Taking part to your intensive course was one of the most exciting things I've ever done in my life, and the thought of those moments keep pushing me with strength. Thank you for your art and your energy.

Concludo con un ultimo grazie a tutti quelli che non ho menzionato qui, ma che hanno contribuito al raggiungimento di questo traguardo.