Computers & Security 116 (2022) 102643

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

TC 11 Briefing Papers

Optimization of code caves in malware binaries to evade machine learning detectors

Javier Yuste[a], Eduardo G. Pardo[a,*], Juan Tapiador[b]

[a] Universidad Rey Juan Carlos, C/Tulipán s/n, Móstoles, 28933, Spain
[b] Universidad Carlos III de Madrid, Avda. de la Universidad, 30, Leganés, 28911, Spain

ARTICLE INFO

Article history:
Received 30 June 2021
Revised 22 December 2021
Accepted 5 February 2022
Available online 8 February 2022

Keywords:
Malware
Evasion
Machine learning
Adversarial example
Genetic algorithm

ABSTRACT

Machine Learning (ML) techniques, especially Artificial Neural Networks, have been widely adopted as a tool for malware detection due to their high accuracy when classifying programs as benign or malicious. However, these techniques are vulnerable to Adversarial Examples (AEs), i.e., carefully crafted samples designed by an attacker to be misclassified by the target model. In this work, we propose a general method to produce AEs from existing malware, which is useful to increase the robustness of ML-based models. Our method dynamically introduces unused blocks (caves) in malware binaries, preserving their original functionality. Then, by using optimization techniques based on Genetic Algorithms, we determine the most adequate content to place in such code caves to achieve misclassification. We evaluate our model in a black-box setting with a well-known state-of-the-art architecture (*MalConv*), resulting in a successful evasion rate of 97.99 % from the 2k tested malware samples. Additionally, we successfully test the transferability of our proposal to commercial AV engines available at VirusTotal, showing a reduction in the detection rate for the crafted AEs. Finally, the obtained AEs are used to retrain the ML-based malware detector previously evaluated, showing an improve on its robustness.

1. Introduction

The number of malicious software (malware) samples actively being used has grown exponentially over the last two decades (Aleshkin and Lesko, 2019). Due to the high return of investment of this kind of software, cybercrime has become a profitable business (Anderson et al., 2019; Connolly and Wall, 2019; Huang et al., 2018). In this new scenario, traditional detection techniques, mainly based on signatures (Gandotra et al., 2014) and observed behavior (Bazrafshan et al., 2013; Brumley et al., 2008), have been shown unable to deal with the current volume and complexity of the threats. Signature-based techniques search for previously known malicious patterns. On the other hand, behavioral techniques try to detect malware based on its interactions with the system at runtime through the use of heuristic rules. Given the overwhelming number of never-seen-before malicious samples to be analyzed, recent progresses in Artificial Intelligence (AI) have been incorporated to tackle the problem of malware detection (Sahay et al., 2020; Xue et al., 2019).

Although the use of AI for malware detection is not new (El-Bakry, 2010; Firdausi et al., 2010; Shah et al., 2013), its recent advances, mainly based on the use of Machine Learning (ML) techniques, have contributed to increase the performance of the aforementioned task. Generally speaking, ML techniques are usually based on the construction of models based on the analysis of data samples. These models represent an abstraction of the application domain with the aim of classifying new instances. Therefore, they work under the assumption that new data from the same context, not yet evaluated, will follow the same statistical distribution represented in the model.

Artificial Neural Networks (ANN) have become a very successful ML technique in many different tasks. Specifically, Deep Learning (DL) architectures (i.e., a particular type of ANN) have been applied in contexts such as image recognition (Simonyan and Zisserman, 2014), natural language processing (Young et al., 2018), voice speech recognition (Das, 2019), or malware detection (Sahay et al., 2020), among others.

Despite its effectiveness when applied to malware detection, DL cannot solve all problems in this domain nor it is exempt from shortcomings (Gibert et al., 2020b). One of the best-known weaknesses is their vulnerability against a class of evasion techniques commonly referred to as Adversarial Examples

* Corresponding author.
E-mail addresses: javier.yuste@urjc.es (J. Yuste), eduardo.pardo@urjc.es (E.G. Pardo), jestevez@inf.uc3m.es (J. Tapiador).

https://doi.org/10.1016/j.cose.2022.102643

(AEs) (Papernot et al., 2016). AEs are carefully crafted inputs that are misclassified when provided to DL models. Due to the hostile nature of the environments in which malware detection solutions are deployed, AE attacks and defenses have recently been the subject of extensive research (two reviews can be found in Chakraborty et al. (2018); Yuan et al. (2019)). Within the malware detection field, an AE has to be designed in such a way that it evades the detection capabilities of the DL network while preserving its malicious functionality.

The main contribution of this paper is the proposal of a new method to derive AEs from existing malware binaries that were previously detected by a DL model. Furthermore, our method produces fully functional AEs in a reasonable amount of time. To evaluate the performance of our proposal, we tested the samples produced by our method against a state-of-the-art malware detector based on DL (*MalConv*) that was recently proposed (Raff et al., 2018). The obtained results are compared with seven previously proposed approaches (Demetrio et al., 2019, 2020, 2021; Kolosnjaji et al., 2018; Sharif et al., 2019), and also with commercial anti-virus detectors. Finally, the obtained AEs have been used to retrain the aforementioned ML-based malware detector, showing an improve on its robustness. Our technique can be used both to evaluate the robustness of current DL models used for malware detection, and as a tool to generate new samples that can be used for adversarial retraining, which might result in stronger DL models. In order to facilitate the reproducibility of our results and to foster further research in this area, we make public the code and data used in this work[1].

The rest of the paper is organized as follows. In Section 2 we discuss the state of the art in the generation of AEs and identify the key research gaps. In Section 3 we describe our method to generate AEs, which we experimentally validate in Section 4. Finally, Section 5 concludes the paper by summarizing our key contributions.

2. Related work

We next review related works in the areas of malware detection based on machine learning techniques and the generation of Adversarial Examples. Then, we identify the most outstanding research gaps.

2.1. Machine learning based malware detection

ML is increasingly playing an important role in the field of malware detection, especially in combination with traditional techniques. In this area, there exists a heterogeneous corpus of input features to DL models: Application Programming Interface (API) imports (Saxe and Berlin, 2015), API calls collected at runtime (Kolosnjaji et al., 2016), malware images (Liu et al., 2020), behavior (Smith et al., 2020), headers of files (Radwan, 2019; Raff et al., 2017), permissions (Hojjatinia et al., 2019) or even whole binaries (Raff et al., 2018). Regardless of the actual features provided as input to the network, all these models produce as output either a binary result (malware/benign), or a multiclass result distinguishing the type or family of the malware sample.

ML approaches for malware detection can also be classified according to the method used to extract the features from the samples that will be used as input data. Static analysis approaches collect features from the sample's code and other software artefacts, without running the program (Hojjatinia et al., 2019; Lee et al., 2019). Dynamic analysis approaches rely on features obtained by running the sample (Kolosnjaji et al., 2016), often in an instrumented sandbox. Some approaches use a combination of static and dynamic approaches (Wang et al., 2017). Feature extraction techniques (either static or dynamic) present the common difficulty that they need a system devoted to extract the features (Aghakhani et al., 2020).

A recent proposal introduces an alternative approach that avoids the need of the feature-extraction step (Raff et al., 2018). The authors propose a DL model that receives the whole binary sample (i.e., its raw bytes) as an input. This work inspired similar featureless approaches (Krčál et al., 2018; Le et al., 2018; Millar et al., 2020) . Additionally, it has been used as a comparison framework (Anderson and Roth, 2018; Coull and Gardner, 2018; Roth et al., 2019) and also as a target for the design of AE attacks (Demetrio et al., 2019, 2020, 2021; Kolosnjaji et al., 2018; Kreuk et al., 2018; Sharif et al., 2019).

2.2. Adversarial evasion attacks

Attacks on machine learning have been known for more than a decade (Biggio et al., 2013; Biggio and Fabio, 2018; Szegedy et al., 2014). AEs attacks have been classified in different categories depending on the knowledge that the attacker has about the model tested (Anderson et al., 2017): i) the attacker has fully access to the model; ii) the attacker does not have access to the model, but it is able to try it and receive the scores provided by the model as a response; and iii) the attacker can only probe the detector and receive labels (e.g., malicious or benign) in response. The first category can be considered as a white-box approach and eases the development of efficient AEs. On the other hand, the second and third categories can be classified as black-box approaches. These approaches are harder for the attackers, but they are also the most realistic scenarios. Recently, some theoretical efforts have been made towards the discovery of the desired properties in the design of AE attacks (Amsaleg et al., 2021; Pierazzi et al., 2019). The use of AEs has also been suggested in previous works in the literature as a method to increase the robustness of DL models (Chen et al., 2017; Hashemi and Mozaffari, 2019).

In this paper, we introduce a method for designing AEs from existing malicious Portable Executable (PE) binaries by introducing perturbations in the binary. In contrast to other domains such as image recognition, the generation of AEs in malware detection presents harder constraints. Specifically, the modification of a functional binary (i.e., compiled code) is prone to corruption, which might result in a non-functional binary. Valid methods in this domain must be able to generate AEs that evade the target DL network while preserving their semantics and execution integrity. We next discuss the most relevant techniques proposed in this area.

A common strategy for the introduction of perturbations in malware binaries consists of appending data at the end of the sample. This is a safe way to introduce perturbations since the original functionality remains untouched. This is due to the fact that the original code and data sections are not modified, and the references contained within the code do not need to be updated. The appended data is still readable by the DL model (i.e., the malware detector) and, furthermore, can be freely edited with the aim of minimizing the probability that the binary is classified as malware. Using this technique, Kolosnjaji et al. (2018) propose a white-box approach based on appending carefully crafted bytes at the end of files, which are optimized via a direct gradient-based attack. Their proposal is tested against *MalConv* in Kolosnjaji et al. (2018). Similarly, Kreuk et al. (2018) propose to append bytes at the end of PE binaries to achieve misclassification, using a surrogate loss function to optimize the introduced bytes in a white-box approach. When indicating that both proposals are tested in a white-box approach, we mean that the internal information of the DL model (i.e., number of layers, number of neurons, weights, etc.) is known to the

[1] https://github.com/JavierYuste/Optimization-of-code-caves-in-malware-binaries-to-evade-Machine-Learning-detectors.

attacker, who uses this knowledge to optimize the introduced perturbations.

A different approach in Sharif et al. (2019) proposes a mechanism to replace recognized instructions with semantically equivalent code. Note that, when working with PE binaries, separating the data from the code is a nontrivial task. This approach is limited by the number of available instructions to be replaced, and by the number of available modifications for each instruction. Additionally, the code to be replaced cannot be self-modifiable (i.e., instructions should not be modified at runtime). To address these shortcomings, the approach also uses a displacement technique, rearranging code sequences and inserting new ones, which allows them to further modify the samples at the cost of increasing their size. The proposal is also tested against *MalConv* and other DL models in Sharif et al. (2019). The authors use the structural properties of the network in a white-box approach, but also test their method in a black-box setting where only the output score of the model is known.

In Demetrio et al. (2019), the authors propose a method, named Partial DOS, which modifies some bytes in the DOS header of PE files, with the objective of generating an AE. This approach is later revisited in Demetrio et al. (2020), where the authors extend Partial DOS by modifying additional bytes of the DOS header. They called this extension Full DOS. This is, to the best of our knowledge, the first time that a metaheuristic (i.e., a Genetic Algorithm) was used to optimize the content of the modified bytes. In Demetrio et al. (2021), the same authors explore an alternative method, named GAMMA, to craft AEs from existing malware. The technique relies on the injection of benign content in different parts of a PE file. The authors report the differences in the detection rate of common detectors from VirusTotal (Chronicle, 2004) when comparing the original malware samples and the generated AEs.

Other techniques to generate AEs in this domain take advantage of memory alignment (Kreuk et al., 2018; Szor, 2005). Particularly, sections in PE binaries often contain padding bytes (i.e., not used bytes) that are known as *code caves* in the literature (Szor, 2005). These spaces are suitable to be modified and they have been frequently used to trojanize legitimate applications, since the perturbations introduced there may be hardly recognizable. While the location of these spaces might be relatively easy to identify, the available space is very limited. The use of *code caves* is exploited by Demetrio et al. (2020) but they only use the space between the headers and the sections in the PE binary. This proposal is combined with the modification of unused bytes in the headers of the file.

2.3. Key research gaps

ML techniques are used as a key component in malware detectors in combination with traditional techniques. One key strategy to make such ML techniques more robust is to investigate the automatic generation of AEs in black-box settings. Previous works have identified code caves as a powerful vehicle to generate AEs in these settings, and some authors have explored the use of optimization techniques to automatically determine the content of such code caves.

Our work builds on and improves previously proposed techniques in this area by presenting a method that can determine, for each code cave: the smallest size needed, the right location, and the adequate content. In doing so, we introduce a more precise formulation of the problem that will result in AEs of higher quality than previous techniques. Furthermore, we couch the three previous tasks as an optimization problem and show that the use of efficient optimization techniques can solve them in a reasonable time.

3. Our proposal

In this paper, we explore the design of AE attacks against DL models which receive raw bytes as input in a black-box setting. Our goal is to propose a new method to design AEs from existing malware binaries in the PE format commonly used in Windows operating systems (Singh, 2009). Since modifying bytes in a compiled PE binary is prone to corrupt its functionality, our method is based on introducing additional bytes that can be freely manipulated without altering the original behavior of the PE file. We cannot know in advance which bytes have a larger influence on the output, nor which content is more suitable to produce missclasification, since we are not assuming any knowledge about the target DL model to be evaded. Therefore, we need an exploratory method that can introduce bytes at different locations within the PE file and modify them in a systematic way. Based on these assumptions, our proposal consists of two main phases. First, we build a modified sample (see Section 3.1) in order to introduce unused blocks in the PE file, guaranteeing that the original functionality of the sample remains untouched. Second, we optimize the content of the previously introduced blocks (see Section 3.2) to minimize the probability of detection by the considered model.

The aforementioned method is summarized in Fig. 1. First, we introduce a predefined unused space before each section in the PE file (step 1). Then, we test if the sample is detected by the target DL model (step 2). If not, the algorithm stops (step 7). Otherwise, we proceed to optimize the content of the introduced spaces (step 3). If the modified sample is not detected by the target DL after the optimization phase (step 4), and the size of the unused space is not larger than a predefined threshold which will be described in the following sections (step 5), we increase the size of the unused space (step 6) and we start a new iteration of the algorithm. This procedure is repeated until an AE is crafted (i.e., it evades the DL model) or the total size of the introduced space is larger than the predefined threshold.

We note that this approach to build AEs based on code caves refines previous work by handling together three related but separate tasks: (1) determining dynamically the smallest size needed to introduce one or more code caves for evading a ML detector (steps 1 and 6 in Fig. 1); (2) studying the right location for the code caves (step 1 in Fig. 1); and (3) introducing the adequate content of the code caves by using an advanced optimization technique (step 3 in Fig. 1). In the latter, we also illustrate the importance of performing a fine-tuning of the optimization technique in comparison with other approaches.

3.1. Modification of samples

This procedure starts with a sample consisting in a functional malicious PE file. Each PE file can be divided in two parts: headers and sections. Headers contain the information necessary for the operating system to load the PE file in the memory of the computer. On the other hand, sections contain the code and data necessary for the correct execution of the PE file.

The operating system loader is responsible for creating a new process, reserving some address space in memory for this process, and mapping every section from the PE file in that space (Yosifovich et al., 2017). In a general overview, the operating system reads the headers from the PE file which contain the *sections table* that specifies, for each section, the location of the section on the disk and, additionally, the space that will be needed in memory at execution time.

It is important to notice that the amount of space used in memory must be multiple of the page size of the operating system. Therefore, some unused space might be available in the last page in memory. Similarly, the sections in the PE file are stored on the

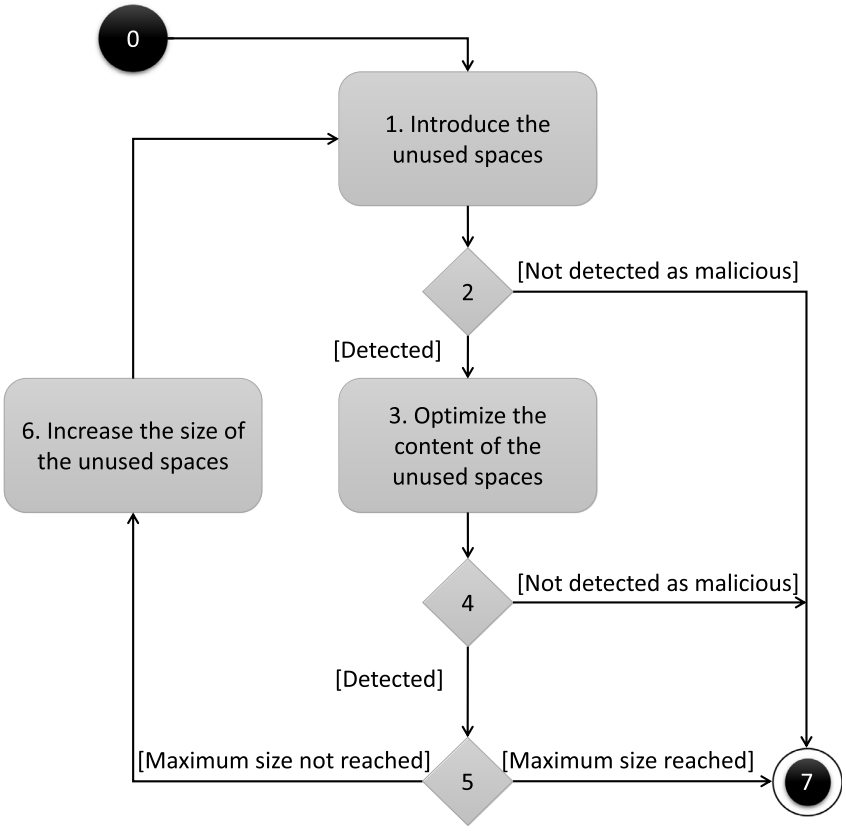


Fig. 1. Activity diagram of the proposed method.

disk in positions aligned with the page size, which is indicated in the header of the PE file. Therefore, as it was the case of the space in memory, there might be some available space at the end of each section stored on disk. This space has been historically exploited to store malicious code within legitimate applications (Szor, 2005).

Since the headers specify the addresses and sizes of each section, the mapping process from disk to memory can be controlled such that not everything is copied from disk. This fact opens an interesting possibility for our goal: we may introduce unused spaces in between sections that will not be loaded in memory. Although this technique is not new (Demetrio et al., 2020; Kaspersky, 2005), to the best of our knowledge, it has not yet been fully exploited to generate AEs for malware detection. The introduction of unused space has been only explored between the headers and the first section. In particular, we extend the current proposals by creating *code caves* in between sections. Furthermore, we propose a dynamic widening of the existing available space, depending on the binary explored, with the aim of constructing a successful AE.

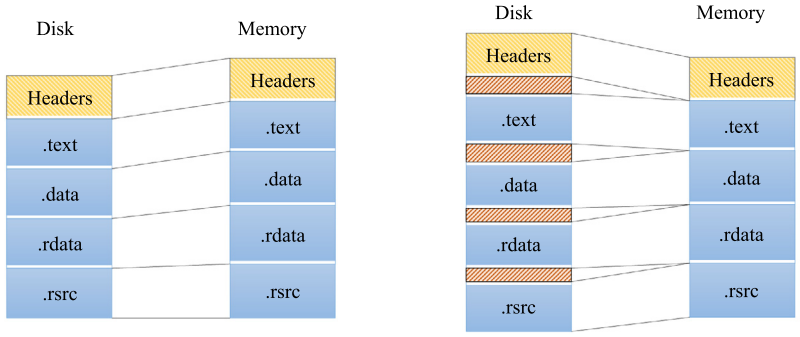
Fig. 2a shows an example of an original PE file structure on disk and the associated memory mapping. As shown in the figure, sections of the PE file are loaded in memory in the same order as they are stored on disk. Note also that some sections might occupy a larger space in memory than on disk. Our proposed method for introducing modifications in a sample is based on exploiting these observations. The modifications consist of unused space on the disk (in between sections) that can be modified in the next step without corrupting the sample. Fig. 2b shows an example of the modification of the PE file depicted in Fig. 2a. In this case, we illustrate the introduction of 4 slots of unused space (in rectangles with orange dashed lines) of different sizes, one before each section. Note that the introduced slots are not mapped into memory. Therefore, the bytes introduced in those spaces do not affect the running time nor the behavior of the original program. Since the introduced bytes are never executed, they are meaningless for running purposes and do not need to represent valid instructions.

The method proposed to introduce unused spaces can be summarized as follows:

1. Parse the header of the PE file.
2. Extract the starting address of each section on disk.
3. Choose a target section t. Let A_t be its starting address on disk.
4. Determine the size S_t of the code cave to introduce before section t.
5. For every section i in the PE binary starting at a position A_i such that $A_i \geq A_t$, modify the starting address in the header of the PE file (i.e., modify the corresponding entry in the *sections table*) such that the new starting address is $A'_i = A_i + S_t$.
6. Shift the content at disk of each section whose header has been modified in step 5 to its new starting address.
7. Repeat steps 2–6 as many times as needed (i.e., one for each target section).

Note that the algorithm considers that the spaces are introduced in order, starting from the beginning of the file. Additionally, since it modifies the pointers in the headers on disk, these spaces are never loaded in memory, as discussed before. Finally, the unused spaces recently introduced will be modified in a later step (see Section 3.2) in order to evade the detection by the ML model.

The procedure needs to determine the amount of initial space introduced (let S be the sum of the different S_t). However, this quantity along with the optimization phase might not be able to evade the DL model. In that case, we need to progressively increase the size of the total unused space introduced until the modified sample evades the DL model or it reaches a maximum size. The initial value of S, the increase steps, and the maximum size al-



(a) Original sample before modification.

(b) Modified sample with introduced unused spaces.

Fig. 2. Representation of the memory mapping of the original sample and a modified version with unused spaces introduced by the attacker.

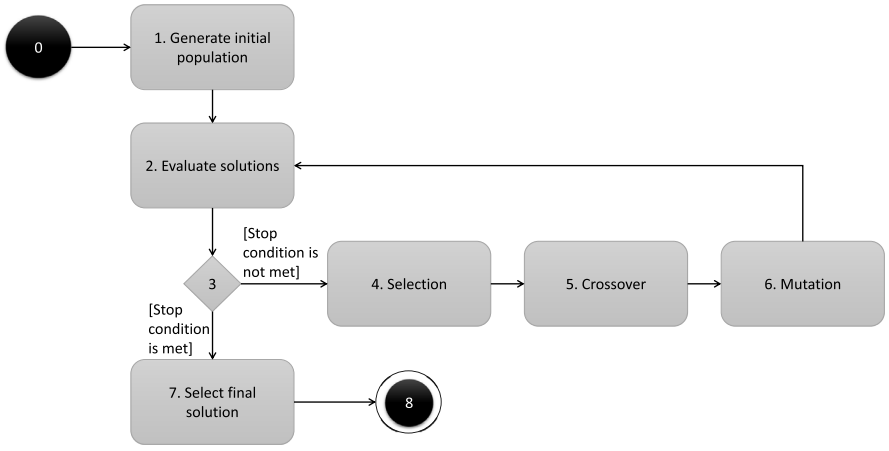


Fig. 3. Activity diagram of a Genetic Algorithm.

lowed before stopping can be considered search parameters and will be discussed in Section 4.2.

3.2. Optimization

We next describe our approach to fill the inserted unused spaces with the aim of improving the evasion rate. In particular, the objective function is to minimize the probability of a modified PE binary being classified as malware by the DL model. To tackle this optimization problem, we propose a procedure based on the use of metaheuristics (Gendreau and Potvin, 2010), to determine the right content of the unused spaces.

Metaheuristics are high-level procedures which help subordinate heuristics to escape from local optima, finding efficient solutions to hard optimization problems in a very short time. Evolutionary Algorithms are a notable class of metaheuristics inspired by natural processes. Specifically, we propose the use of Genetic Algorithms (GAs). GAs explore the search space by using several bioinspired operators (selection, crossover, and mutation) to optimize an initial population of solutions. The use of GAs instead of other successful metaheuristics is based on the characteristics of the tackled problem. In this case, we are facing a black-box optimization problem (i.e., there is not access to the implementation details of the model used to evaluate a solution). Therefore, no gradient information can be used to optimize the inputs, which makes the family of metaheuristics based on local search procedures not so suitable in this setting.

Fig. 3 provides a flow chart of the main steps of a GA. The method is based on the generation of a group of feasible solutions for the problem (known as the initial population) as the starting point (step 1). Then, these solutions are evaluated (step 2) and, while the stop condition is not met (step 3), the GA selects a subgroup of solutions (selection in step 4) based on quality and diversity criteria. Next, in step 5, it performs a crossover operation (construction of new solutions by combining the characteristics of two or more previously existing ones) trying to capture the best characteristics of each solution, to obtain a better combined solution. Finally, in step 6, the GA applies a mutation operation which randomly modifies parts of the obtained solution. Once this step has been made and the obtained solutions have been reevaluated, the process is repeated as many times as necessary. Each of these iterations is known as a generation.

The inputs to the optimization phase described in this section are the ones provided by the output of Section 3.1. In particular, they are PE files modified in such a way that there are unused spaces that have been introduced before one or more sections. However, as it is customary in optimization, real instances (problem examples) cannot be provided straight away to the optimization algorithms, but they usually need to be represented in such a way that the algorithm is able to handle them. In this case, we are only interested in the introduced unused spaces before the sections. We refer to those spaces as target blocks. All target blocks in a PE file are then concatenated into a single array of bytes which represent a solution for the GA.

In Fig. 4 we depict the transformation of the real instance (i.e., the modified PE binary) into an empty solution represented by an array of bytes (see steps 1 and 2 in Fig. 4). Following the GA terminology, this array represents a chromosome (a solution for the

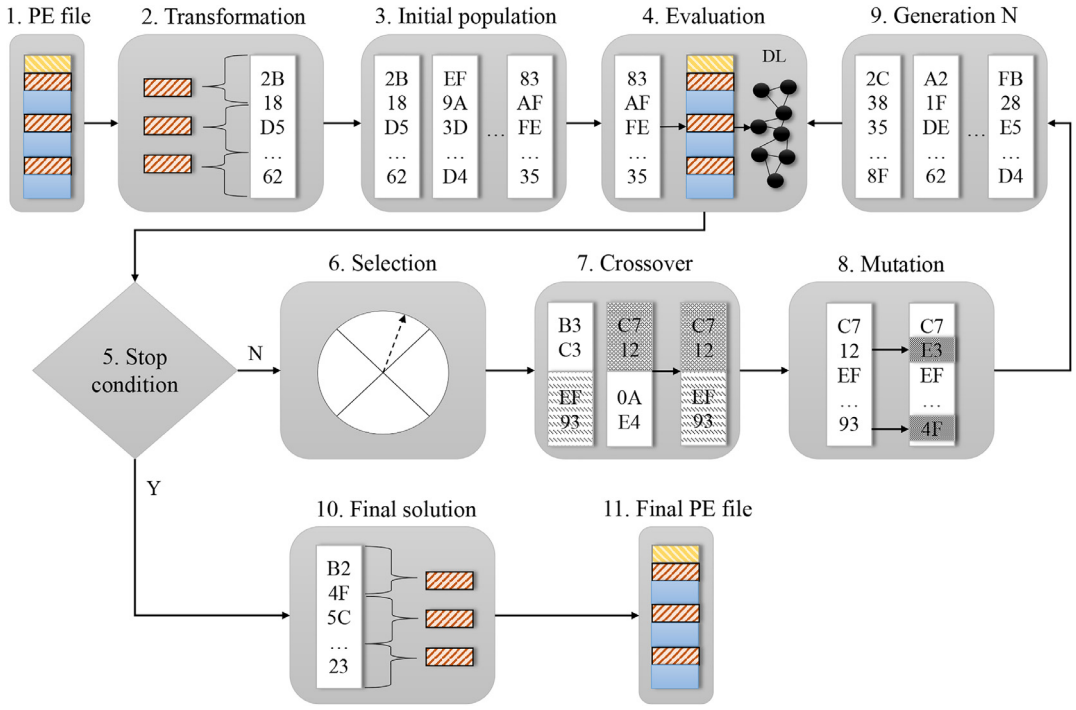


Fig. 4. Adaptation of the general GA scheme to the problem of evading PE malware.

optimization process). Additionally, each byte in the array represents a gene of the solution. Note that each gene is one byte that can take up to 256 different values.

The optimization process starts by building an initial population of solutions (step 3). Each solution of the population is built by introducing a random byte in each position of an empty array with a length equal to the total unused space in bytes. The actual size of this population is an experimental parameter and it might vary among different algorithmic designs. Moreover, it is important to remark that the representation of a solution in this context is only valid for the tackled optimization problem. When a solution needs to be evaluated, it is necessary to write the bytes which are currently in the solution, back into their corresponding unused blocks of the PE file. Then, the new PE file is provided as an input to the DL model to evaluate the solution (step 4). The quality of the solution is measured by the output of the DL model. Note that this is a value ranging from 0 to 1, where values close to 1 indicate malicious, while values close to 0 indicate benign. Therefore, the lower the output, the better the solution. As it was described in the introduction of the GAs, once the population is conformed, it evolves into a new generation by applying the selection (step 6), crossover (step 7), and mutation (step 8) operators, until the stopping criterion (evaluated in step 5) is met. This stopping criterion is usually related to the execution time, to the number of generations, or to the quality of the obtained solutions in the population. We detail the stopping criterion used in Section 4.

Next, we describe the implementation of each of the key operators within the GA:

- Selection: this operator performs a selection of a subgroup of solutions within the current population to be crossed in the following step. We propose two different selection operators: i) elitism and ii) tournaments (Miller et al., 1995). In the first case, the solutions are selected by their quality conserving the best solutions of the population. In the latter, each tournament tries to find a balance between randomization and intensification. Particularly, it consists of selecting ten solutions at random and then picking the best solution within this group to be included in the diverse group. Note that each chosen solution is extracted from the population, so a single solution cannot be selected twice. We explore the performance of the combination of these two criteria in Section 4.2.
- Crossover: each solution from the elitist group is crossed over with each solution from the diverse group, producing two offspring solutions per crossover. We propose here the use of two different crossover operators inspired by the uniform crossover: i) based on random voting, and ii) based on weighted voting. Particularly, in both of them we divide each chromosome in different chunks of the same size. The first chunk of each offspring is the first chunk of one of the parents, the second chunk of each offspring is the second chunk from one of the two parents, and so on. The difference between the random and weighted voting is that the probability of selection in the first case is equal for both chunks, while the probability in the second strategy is pondered depending on the quality of the solution (i.e., chunks of better solutions have a higher chance of being selected). In particular, the probability (P) of selecting a chunk i from the first parent (p_1) in the weighted voting strategy is $P_i = 1 - (q_1/(q_1 + q_2))$, where q_1 is the quality of the first parent and q_2 is the quality of the second parent. Finally, the size of the chunks considered for crossover is a search parameter that must be empirically adjusted.
- Mutation: the offspring solutions obtained after the crossover are then mutated with a particular probability p_1. Then, if a solution is selected to be mutated, a percentage (p_2) of its genes are mutated (modified by a random value). Notice that p_1 and p_2 are search parameters that must be experimentally adjusted and that will be studied in the experimental section.

To maintain a constant number of solutions across generations, the next generation is built with the best solution from the previous generation and the best solutions obtained from the offspring. If the algorithm does not stop, this new generation becomes the initial population for the next iteration and steps 5–8 from Fig. 4 are repeated.

It is important to remark that aggressive selection mechanisms (such as the use of an elitist group, or the selection of the best solution for the next iteration) may speed up convergence, impeding sufficient exploration of the problem space and falling into local optima regions (Oliveto et al., 2018). Therefore, to decrease the convergence rate, it is important to empirically adapt the parameters of the mutation operator.

4. Experimental results

In this section, we present the experiments devoted to test the algorithmic proposal introduced in Section 3. First, we describe the dataset collected and used in our experiments (Section 4.1). Then, we perform a set of preliminary experiments to adjust the configurable parts of our algorithm (Section 4.2). In Section 4.3, we present the results of the final experimentation performed, where we show the ability of our proposal to evade the targeted DL model (Raff et al., 2018). In Section 4.4, we evaluate the utility of using the crafted AEs in the task of retraining a ML-based malware detector. We extend the evaluation of the proposed approach by comparing it with other methods over a dataset of modern ransomware samples in Section 4.5. Finally, we describe some limitations of our work in Section 4.6.

The targeted model (*MalConv*) consists of a convolution network architecture which returns a value in [0,1]. Its objective is to classify PE files as benign or malicious based on their raw content. Note that we consider that values larger than 0.5 indicate that a sample must be classified as malware, as it is assumed in other proposals (Kolosnjaji et al., 2018; Kreuk et al., 2018). It is also important to remark that we used a pretrained implementation that was made publicly available at Anderson and Roth (2018). We refer the reader to Raff et al. (2018) for further details about the DL model. The results obtained with the DL model have been favorably compared with four previous approaches in the state of the art. Additionally, we have retrained the network with the generated AEs and verified the existence of an increase of its robustness. Finally, we evaluate the transferability of our proposal to commercial anti-viruses.

All experiments were run on an Intel®Core™i5-8250U CPU @ 1.60GHz, with 16 Gb RAM. The Operating System used was Ubuntu 20.04.1 LTS, 64-bit. All implemented methods were coded in Python 3.7. We used the open source tools Radare2[2] and Pefile[3] to manipulate the PE binaries.

4.1. Dataset

To test our proposal, we collected a dataset of malicious samples from *VirusShare* (Roberts, 2020). We first downloaded 42,658 samples and then filtered out those files that were not in 32-bit PE format (.js,.html, 64-bit binaries, etc.). This resulted in 3540 PEs. In order to avoid duplicated or highly similar samples that could bias our results, we compared the similarity between each pair of samples using *ssdeep* (Kornblum, 2006) and filtered out samples that were almost identical, resulting in 3131 samples. Finally, we selected the samples that were detected as malicious by *MalConv* (i.e., those with a prediction score larger than 0.5). The resulting dataset contained 2036 different 32-bit PE malicious samples. The average and median scores (in the range [0,1]) for this dataset reported by *MalConv* were 0.9804 and 0.9999, respectively, with a standard deviation of 0.0687. To determine the validity of the high detection rate obtained by *MalConv*, we checked if any of the collected samples were previously used for training the model (EMBER dataset in Anderson and Roth (2018)). We found that only 2 out of the 2036 samples were part of the training set. Therefore, this indicates a very large confidence in classifying the samples as malware.

To validate the dataset for our proposal, we scanned it using the 79 AV engines available in VirusTotal (Chronicle, 2004). All samples but 3 were detected by at least one AV, and only 17 samples were detected by less than 4 vendors. On average, the collected samples were detected by 54.81 different AVs. Therefore, we assume that the samples collected can be considered as malware. To check the diversity of malware types in the dataset, we analyzed the labels returned by the anti-virus engines and found that 48.76 % of the samples are considered trojans, 20.00 % are considered worms, 14.65 % are labeled as viruses, 2.96 % are labeled as adware, 2.83 % are considered downloaders, and 0.35 % are labeled as ransomware.

Finally, since our method alters the binary size, we analyzed the distribution of file sizes of the original samples. The size of the binaries ranges from 4096 bytes to 21 megabytes. The number of sections varies from 1 to 18, with an average of 4.58 sections per PE file. Up to 49.69 % of the collected samples have one or more virtual sections, and at least 12.21 % seem to be packed.

4.2. Preliminary experiments

Since the method introduced in Section 3 depends on several parameters that might influence the performance of the algorithm, we have conducted a set of preliminary experiments to study the behavior of the most relevant ones. For these experiments, we use a reduced subset of representative instances composed by 100 samples (4.91 % of the overall dataset) that were selected following the same distribution (in terms of size) than the original dataset. All experiments in this section have been performed over this reduced dataset.

As far as the GA is concerned, we present here the experiments performed to select the best crossover and mutation strategies (see Section 4.2.1 and 4.2.2 respectively). For the sake of simplicity, we do not report other preliminary experiments to determine the size of the population (set to 50 individuals) or the number of solutions selected for the crossover (set to 10 individuals). For the selection operators used in the GA, we explored a selection based on quality, also known as elitism (only the best solutions are chosen) and a selection based on tournaments (each tournament consists of selecting ten solutions at random and the best among them is selected for the next iteration). Finally, we also explored the combination of these two strategies (choosing half of the solutions by elitism and the other half by tournaments). We observed that the use of both selection operators in combination outperformed the use of each of them in isolation. This fact is commonly observed in the performance of Genetic Algorithms since it helps to find a balance between intensification (the best solutions are chosen) and diversification. Additionally, it is important to notice that we used a mixed stopping criterion for the algorithm, based on the quality of the solution found. Since we are trying to build an AE starting from a malicious PE file, the GA halts at any iteration as soon as the modified sample is classified as non-malware. Alternatively, the method stops if the improvement found in the last 10 generations is smaller than 1 %, but letting the algorithm to run for at least 50 generations.

As for the amount of introduced space, we present here the experiment performed to determine the influence of its location (see Section 4.2.3). Again, for the sake of simplicity, we avoid reporting other preliminary experiments such as determining the initial space introduced S (set to 1 % of the total size of the PE file); the increase of unused space introduced at each iteration (set to 3 % of the total size of the PE file); or the maximum allowed size of the introduced space (set to 100 % of the total size of the PE file).

[2] https://rada.re/n/.
[3] https://github.com/erocarrera/pefile.

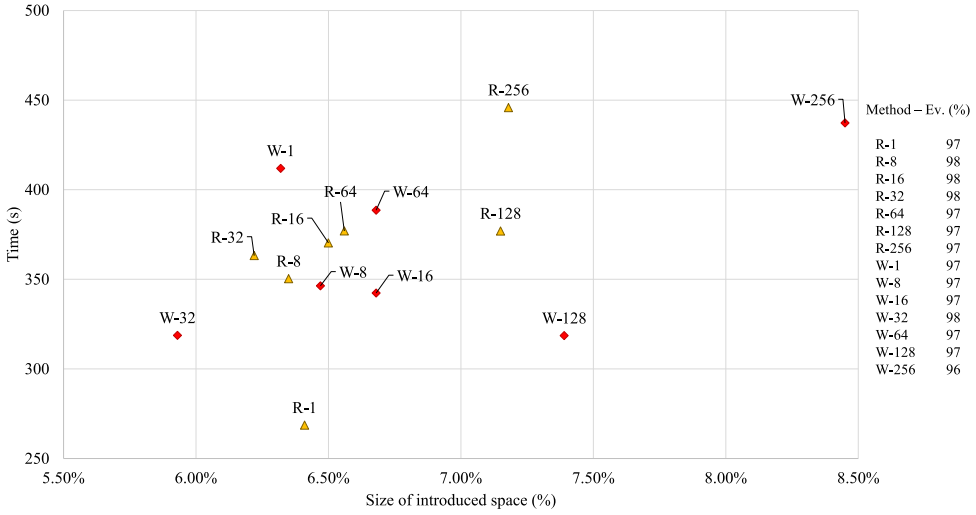


Fig. 5. Representation of the average size increment and the average time needed to build an AE.

Finally, in Section 4.2.4, we report a comparison between the performance of the GA and the performance of a random content generator when optimizing the content of the introduced code caves.

4.2.1. Crossover

First, we compared the 2 different crossover methods (random voting and weighted voting, denoted with an R and a W respectively) proposed in Section 3.2, with 7 different configurations of the size of the chunks in which solutions were split to be crossed over. Particularly, we decided to test both crossover methods by using chunks of 1, 8, 16, 32, 64, 128, and 256 bytes. The resulting 14 methods have been named as: R-1, R-8, R-16, R-32, R-64, R-128, R-256, W-1, W-8, W-16, W-32, W-64, W-128, and W-256. We performed two different experiments to evaluate these methods. Note that, in addition to the general parameters already set, we fixed the probability of mutation of a solution to a 75 %, and the percentage of genes to be mutated to the 0.2 % of the total number of genes of the solution. These choices were made as a trade-off between diversity and time of convergence of the method.

1. In the first experiment, for each sample we tried to find the smallest modified sample that was not detected by the target DL model. For this experiment we reported the average size of the total introduced spaces and the time needed to reach the evasion value. Note that in some cases the method was not able to evade the DL model. In those cases, we reported the total time used until the method stopped following the criteria introduced in Section 3, and the maximum introduced size tried. In Fig. 5 we represent the results obtained by each method tested. Particularly, we depict the representation of the average size increment for all the modified instances and the averaged time needed to build the AEs. At a first glance, the shorter the time and the smaller the size, the better. However, there might be configurations of the algorithms which stands out for one of the metrics despite the fact that they are not so successful for the other. Additionally, in the legend of this figure we also present the percentage of evaded samples by each crossover method. In this case, we highlight two methods: “W-32” configured with the weighted voting and chunks of 32 bytes and “R-1” configured with a random voting and chunks of 1 byte. The former was the one which achieved the smaller size increment, while the latter was the fastest method to process all the samples. However, the method “W-32” achieved an evasion rate of 98.0 % (which is the largest among the tested ones), while “R-1” achieved an evasion rate of 97.0 %. These two configurations have been selected for a further evaluation.
2. Given the previous results, the second experiment aims at observing the convergence of the methods over time. We fixed the incremented size of the introduced spaces to the 5 % of the original samples size and we set the time limit for the execution to 480 seconds. We reported the averaged best prediction confidence of the model every 30 seconds, for each of the compared variants of the methods studied. Particularly, in Fig. 6 we present the results for the 2 best variants previously selected (“W-32” and “R-1”). As we can observe, the behavior of both methods is very similar in terms of convergence. Considering that the evasion of the model is achieved when less than a 50 % of prediction confidence is reached, we find that both methods were able to reach this value in less than 90 seconds on average. Also, we observe that the improvement in the evasion rate after 300 seconds can be considered negligible ($<$1 %).

Despite the similarity of the two best configurations, we have configured the crossover method for the rest of our experiments with the weighted voting strategy with chunks of 32 bytes, since it achieves a slightly better evasion rate.

4.2.2. Mutation

As discussed in Section 3.2, the mutation operator has two parameters (p_1 and p_2) that need to be adjusted experimentally. Here, we test different configurations of such parameters. In particular, p_1 indicates the probability of a solution in the population to be mutated, while p_2 indicates the percentage of genes in each solution to be mutated. In Fig. 7 we report the results obtained for the combination of different values of p_1 with different values of p_2. Specifically, we considered $p_1 = \{10, 20, 40, 60\}$ and $p_2 = \{0.2, 0.4, 0.6, 0.8, 1.0\}$. In that figure, we report the average size increment in percentage of the AE and the time needed to reach it. Again, at a first glance, the smaller the size and the shorter the time, the better. As we can observe in the figure, the combination 10-1, which mutates the 10 % of the solutions with a mutation rate of the 1 % of the genes, together with the combination 40-1, which mutates the 40 % of the solutions with a mutation rate of the 1 % of the genes, are the two best combinations in the experiment. However, the averaged time needed by the combination 40-1 is considerably larger than the time needed by the com-

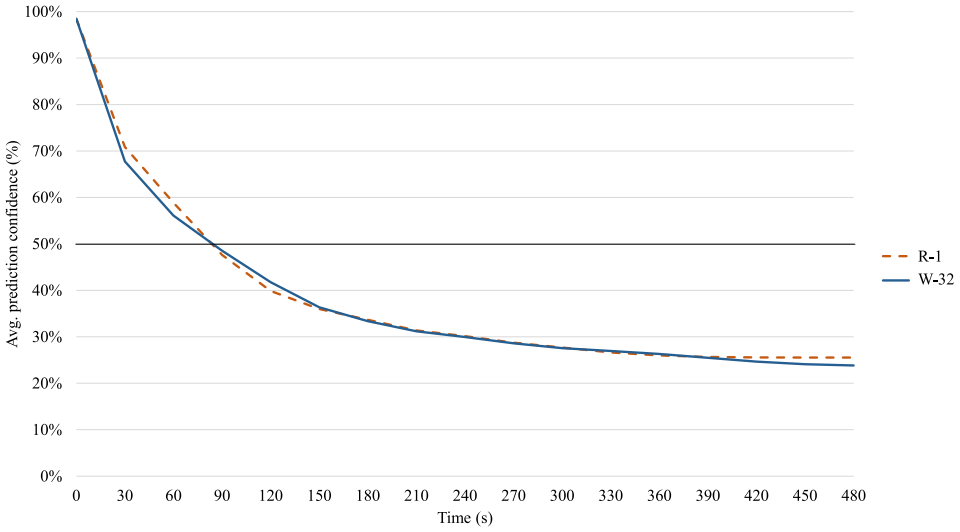


Fig. 6. Evolution of the prediction score of the DL model over the time.

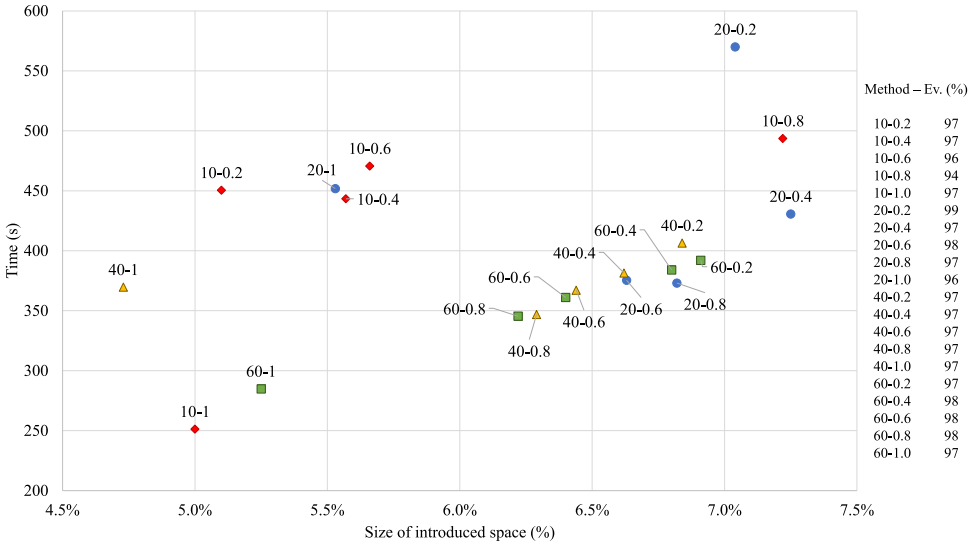


Fig. 7. Representation of the average size increment and the average time needed to build an AE.

bination 10-1 (119 seconds between both methods), while the size increment is not very large (0.3 % between both methods). Therefore, we have selected the combination $p_1 = 10$ and $p_2 = 1$ for the rest of the experiments in the paper.

4.2.3. Influence of the location of introduced spaces

Once all parameters of our GA have been adjusted, we test the influence of the location where the unused spaces are introduced. Recall that our technique exploits the possibility of inserting a block of unused space before each section in the PE file. All previous experiments have been performed by dividing the unused space in equal chunks before each available section. In this experiment, we study the influence of the position of the introduced unused space within the PE file. In particular, we compare the previous approach with the results obtained by introducing the same unused space in a single block before each section.

The samples in the preliminary dataset have been classified depending on the number of sections in the PE file. Then, we have selected the subsets of samples with three, four, and five sections, resulting in 10 samples with three sections, 19 samples with four sections, and 16 samples with five sections.

In Table 1 we report the results of the aforementioned experiment grouped by subsets of instances (3 sections, 4 sections, or 5 sections). For each group, we report two columns with the total number of evaded samples (Evaded Samples) achieved and the average CPU Time in seconds (Time(s)). Each row of Table 1 represents the results of the experiment for a location of the unused space tested. For instance, the first row corresponds to the situation where the unused space has been introduced before Section 1, and so on. In general, we can observe that in most of the cases the position where the unused space was introduced did not affect considerably the final result in terms of the number of AEs crafted (those able to evade the DL model). More specifically, some positions seemed to be more suitable for a particular subset, but that was not consistent in others. Additionally, in the original design of *MalConv* (Raff et al., 2018), the authors chose a convolution network architecture with a global max-pooling for the convolutional activations. Therefore, the model is supposed to find features re-

Table 1
Number of evaded samples and average time of execution for each of the selected subsets in the *Sections importance* preliminary experiment.

	3 sections (10 samples)		4 sections (19 samples)		5 sections (16 samples)	
Section	Evaded samples	Time (s)	Evaded samples	Time (s)	Evaded samples	Time (s)
1	4	183.69	8	225.75	12	171.91
2	5	217.46	8	244.45	11	206.15
3	5	183.82	10	253.11	11	208.23
4	-	-	9	196.80	11	218.02
5	-	-	-	-	11	214.17
All	5	177.57	8	199.73	11	224.12

Table 2
Comparison of the evasion obtained when using a GA or using a random content generator.

Method	Time (s)	Evaded	Total	Ev. rate (%)	Size (%)
Random content generator	1577.45	62	100	62.00 %	49.73 %
GA	685.25	99	100	99.00 %	6.52 %

gardless of their location. Although the idea of testing different locations within the PE file is not effective for this model, it might be a useful approach to test other networks. Also, we do not observe a noticeable reduction of time depending on the strategy used. Therefore, as we could not find significant differences with respect to the position of the unused space, and the number of sections differs among the samples, we keep the strategy of splitting the introduced space equally before every section in the PE file, to design the most general method.

4.2.4. Comparison of the performance of the GA vs a random content generator

To validate the contribution of the GA in the optimization of the unused spaces introduced in the samples, we have compared its performance with respect to a random generator for the content. To do so, we replace the GA with a random bytes generator maintaining the same stopping criteria (i.e., number of iterations without improvement). Table 2 reports the obtained results. As it can be observed, the method using a random generator to determine the cave content achieves a 62 % of evasion rate in 1577.45 s, while the GA is able to achieve a 99 % of evasion in half the time. Additionally, the size of the perturbations introduced was much larger in the case of the random generator (49.73 %) than in the case of the GA (6.52 %).

4.3. Evaluation

To evaluate our proposal, we test the performance of our approach against *MalConv* and compare the results with those of the seven previous methods identified in Section 2 (Demetrio et al., 2019, 2020, 2021; Kolosnjaji et al., 2018; Sharif et al., 2019).

For the sake of simplicity, the first comparison has been performed over the preliminary dataset (100 samples). However, one of the methods tested (Sharif et al., 2019) presents the restriction that it is only applicable to nonpacked samples. Therefore, to have a fair comparison, we selected those instances suitable for all methods, resulting in 37 samples out of the 100 samples from the preliminary dataset. We refer to this subset as the reduced preliminary dataset.

In Table 3 we report the performance of each method over the aforementioned 37 instances. It is important to note that the results presented for the methods denoted as “Padding” (Kolosnjaji et al., 2018), “Extend” (Demetrio et al., 2020), “Shift” (Demetrio et al., 2020), “Partial DOS” (Demetrio et al., 2019), “Full DOS” (Demetrio et al., 2020), and “Gamma” (Demetrio et al., 2021) have been obtained with the original implementation made available in Demetrio and Biggio (2021). Unfortunately, the code for the method denoted as “Sharif” was not available and the results have been obtained with our own implementation of the ideas proposed in Sharif et al. (2019). For each of the considered methods included in Table 3, we report four columns: the average time per sample (in seconds) needed by each algorithm (Time (s)); the number of not corrupted AEs which successfully evade *MalConv* (Evaded); the total number of samples tested (Total); and the evasion rate (Ev. Rate (%)). As we can observe from the obtained results, our proposal was very successful in the task of constructing an AE from the samples used. Particularly, it was able to produce an AE for all the samples tested, achieving an evasion rate of 100 %. However, this behavior needs to be corroborated over the whole dataset since the samples in this reduced dataset were used to adjust the parameters of our proposal. On the contrary, the “Padding” method performed the worst, since it was unable to craft a single AE. It was followed by “Sharif” with an evasion rate of 18.92 %, “Partial DOS” with an evasion rate of 24.32 %, “Shift” with an evasion rate of 29.73 %, “Full DOS” with an evasion rate of 37.84 %, “Extend” with an evasion rate of 75.68 %, and “Gamma” with an evasion rate of 75.68 %. Moreover, it is remarkable that the method denoted as “Sharif” was considerably slower than the rest of the approaches. Notice that the authors of this method originally reported an evasion rate of 33.00 % in Sharif et al. (2019). The differences found in the performance might be partially explained by the threshold used in their original experiments. Let us remember that we consider 0.5 as a threshold for *MalConv* (i.e., a sample is classified as malicious if the prediction score is greater than 0.5), as it was established in other previous proposals (Kolosnjaji et al., 2018; Kreuk et al., 2018). In contrast, Demetrio et al. (2020) and Sharif et al. (2019) used greater thresholds in their original experiments. Note that this eases the generation of AEs since it contributes to increase the evasion rate, but it might reduce the robustness of the proposal.

In the next experiment, we extend our comparison to the whole dataset, formed by 2036 samples. Since the “Sharif” method is not applicable to packed samples and its performance is not very competitive (either in time or evasion rate), we have removed it from this experiment. In Table 4 we report the results of the comparison of our approach with the other six methods of the state of the art (“Extend”, “Shift”, “Padding”, “Partial DOS”, “Full DOS”, and “Gamma”). Again, for each method, we report the same four columns aforementioned. This time, the evasion rate of the methods decreased slightly with respect to the previous experiment, except for “Gamma” and “Full DOS”. Additionally, “Padding” was able to construct an AE in the current dataset.

Table 3
Comparison of the evasion obtained by the different methods, over the reduced preliminary dataset.

Method	Time (s)	Evaded	Total	Ev. rate (%)
Ours	493.77	37	37	100.00 %
Gamma (Demetrio et al., 2021)	89.63	29	37	75.68 %
Extend (Demetrio et al. 2020)	4.77	29	37	75.68 %
Full DOS (Demetrio et al., 2019)	9.47	14	37	37.84 %
Shift (Demetrio et al., 2020)	8.78	11	37	29.73 %
Partial DOS (Demetrio et al., 2019)	10.42	9	37	24.32%
Sharif (Sharif et al., 2019)	337866.63	7	37	18.92 %
Padding (Kolosnjaji et al., 2018)	9.46	0	37	0.00 %

Table 4
Comparison of the evasion obtained by the different methods, over the whole dataset.

Method	Time (s)	Evaded	Total	Ev. rate (%)
Ours	1017.68	1995	2036	97.99 %
Gamma (Demetrio et al., 2021)	96.81	1756	2036	86.25 %
Extend (Demetrio et al., 2020)	8.53	1211	2036	59.48 %
Full DOS (Demetrio et al., 2019)	10.38	804	2036	39.49 %
Shift (Demetrio et al., 2020)	14.04	401	2036	19.70 %
Partial DOS (Demetrio et al., 2019)	10.23	338	2036	16.60 %
Padding (Kolosnjaji et al., 2018)	11.88	1	2036	0.05 %

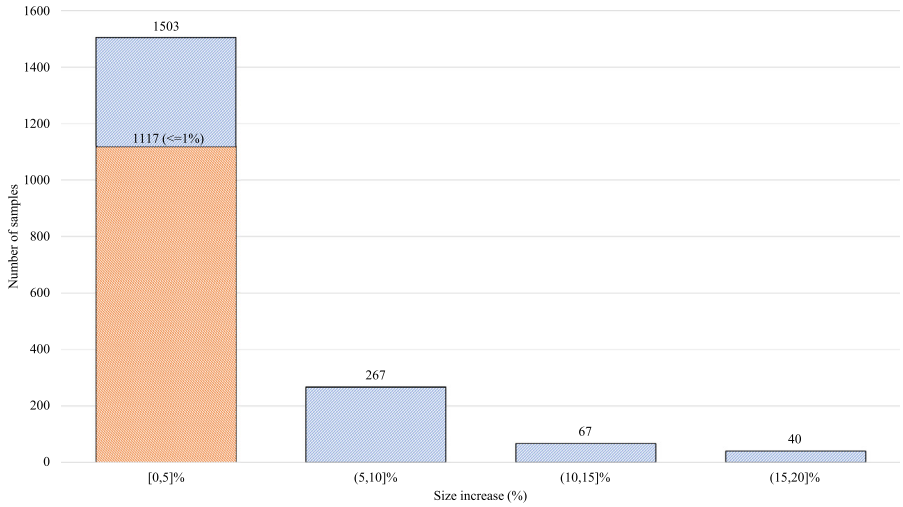


Fig. 8. Histogram of the number of samples with an introduced size in the same interval.

Note that our proposal outperforms the other six in terms of evasion rate in this experiment, even though it is more time consuming. However, we run the other six algorithms as long as they keep improving the solution, and we reported the time instant in which they were unable to make any further improvement. Additionally, we observed the evasion rate of our proposal at the same time span than the rest of the methods (resulting in an evasion rate of 27.75 % in 9 seconds, 29.57 % in 12 seconds, and 31.04 % in 14 seconds). In this sense, the "Extend" and "Gamma" methods present a reasonable evasion rate in very short computing time. However, we believe that the extra time needed by our proposal to achieve higher evasion rates than the rest of the methods is worth spending from an attacker perspective, since none of these approaches (either ours or the ones in the state of the art) are designed to be run on a real-time scenario. Furthermore, our method is easily parallelizable, since the exploration for different sizes of unused space can be performed at the same time.

Our next experiment is devoted to analyze the increase in the size needed by our algorithmic proposal to produce an AE. In Fig. 8, we report an histogram which classifies the modified samples able to produce an AE, when the introduced unused space is smaller than 20 % of its original PE file (1,879 out of 2,036). The histogram reported includes four bars, where each bar represents an interval of the size (in %) of the unused space introduced. Particularly, we report those AEs which needed the introduction of an unused space with a size smaller than the 20 % of its original PE file, in steps of 5 %. Notice that in 1503 samples, the algorithm needed less than 5 % of unused space to produce an AE. Furthermore, 1117 samples out of 2036 needed less than 1 %. On the other hand, only 118 samples needed an increase larger than 20 % of the original PE file, while the method was unable to produce an AE in 41 samples.

Finally, our last experiment is devoted to test the influence on the detection rate of the proposed technique against commercial anti-viruses. In particular, we have tested the original 2036 samples (and their corresponding modified versions for each of the compared methods) against the anti-viruses available at the platform VirusTotal (Chronicle, 2004) on September, 2021, and reported the average number of detections. On average, the original samples were detected by 57.12 anti-viruses (out of 79). On the other hand, the modified samples obtained by any of the compared methods have reduced the number of average detections.

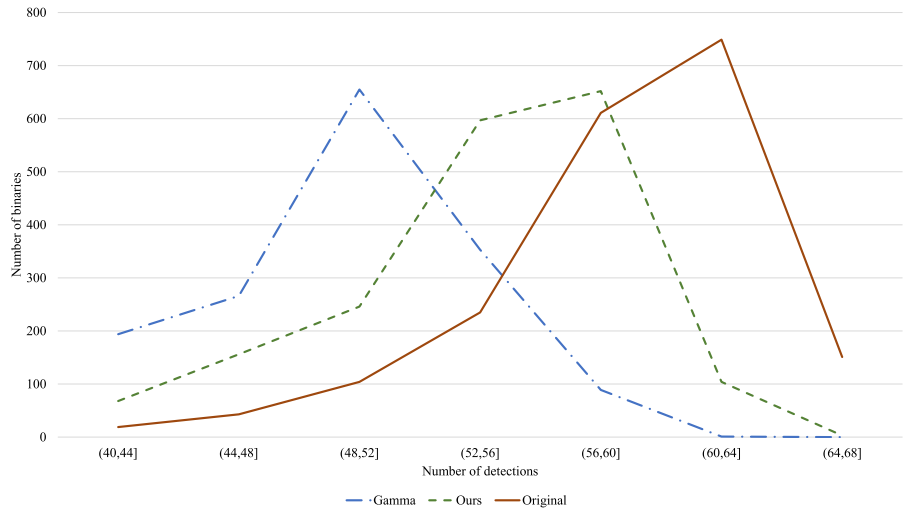


Fig. 9. Number of binaries detected by the same number of anti-viruses at VirusTotal (Chronicle, 2004).

Particularly, the Padding method is the one with the smallest performance (with an average number of 56.92 detections) and the Gamma method is the one with the largest performance (with an average number of 45.28 detections). Our proposal achieved an intermediate performance with an average number of 51.76 detections, which is very similar to the rest of the tested methods.

In Fig. 9, we illustrate the number of samples that were detected as malicious by the same number of anti-viruses for: the original samples, our proposal, and the method which further reduced the number of average detections (Gamma). Comparing the behavior of the anti-viruses in the platform over the original and modified samples for each method, we can see a shift in the graphic, illustrating the aforementioned reduction in the number of detections of the modified samples.

Observing these results, we can conclude that all compared techniques seem to perform similarly in this matter. However, it is important to notice that anti-viruses use several techniques simultaneously to detect malware. Then, we can not assure which of them are actually the ones which triggered the detection, since this information is not available. Furthermore, the dataset used in our evaluation is composed of well-known viruses, therefore it is probable that techniques based on signatures are responsible for classifying the samples as malware.

Since the differences obtained between our proposal and the original samples do not seem to be very large, we performed a Mann-Whitney *U* test to determine if the differences found were statistically significant. This test is a non-parametric test to compare two non-pairwise samples. The obtained results indicate that the difference in the number of detections before and after the modification of the samples is statistically significant with a *p*-value < 0.05.

4.4. Evaluation of adversarial retraining

In this section, we evaluate the utility of using the crafted AEs in the task of retraining a ML-based malware detector. In particular, we have used the original code provided by Anderson and Roth (2018) to perform the retraining of *MalConv*. From the 1997 AEs generated with our method, we have discarded those obtained from the samples in the preliminary dataset, resulting in 1898 AEs that will be used for retraining. Additionally, we have provided the same number of benign binaries for the training of the network, which have been randomly collected from the PE files present in a clean Windows 10 operating system with several programs installed (e.g., PyCharm IDE, Visual Studio Code, Google Chrome, etc.). In each case, we indicated to the network if the sample was benign or malicious. Notice that we have used the same proportion of benign/malicious samples as in the original dataset used in Anderson and Roth (2018).

Then, we have tested the retrained network by performing three different experiments. First, we have evaluated the detection rate of the AEs not used for training purposes. Notice that none of those AEs are classified as malware when using the original network. However, when using the retrained network, 95 out of 99 files are detected as malicious.

In our second experiment, we have applied our proposal to generate AEs from the original samples in the preliminary dataset, but targeting the retrained *MalConv* model instead of the original one. With the original network, our method was able to craft an AE from 99.00 % of the samples. However, when using the retrained *MalConv* network, the method was able to craft a new AE only from 21.00 % of the samples. Furthermore, the size needed to craft a successful AE was larger when targeting the retrained model (i.e., an average size of 5.80 % of the original sample in the case of the original network, versus 16.76 % in the case of the retrained one). Finally, the average running time increased from 685.25 s to 12,486.62 s.

In our third and final experiment, we have evaluated the retrained model over a set of 903 collected malicious samples that were not detected by *MalConv* originally and, therefore, were discarded from our dataset in Section 4.1. The new retrained model was able to detect 642 out of 903 samples, which represents 71.10 % of the tested samples.

The obtained results indicate that the AEs crafted from existing malware with our method can help to retrain ML-based malware detectors with the aim of making them more robust.

4.5. Evaluation with ransomware samples

Given the current threat landscape, we now evaluate the accuracy of our approach over a set of modern ransomware samples available in the VirusTotal Academic share[4]. We selected samples published in the VirusTotal Academic share during 2021, which were categorized as ransomware, and were in the correct format for our study. This resulted in a dataset of 451 ransomware PE files.

[4] VirusTotal Academic share is a dataset of malware samples collected and curated by VirusTotal, which is available for academic research after a vetting process.

Table 5
Comparison of the evasion obtained by the different methods over a dataset of ransomware samples published in VirusTotal Academic share. .

Method	Time (s)	Evaded	Total	Ev. rate (%)
Ours	3003.05	366	451	81.15 %
Gamma (Demetrio et al., 2021)	111.44	288	451	63.86 %
Full DOS (Demetrio et al., 2019)	11.03	265	451	58.76 %
Extend (Demetrio et al., 2020)	7.42	195	451	43.24 %
Partial DOS (Demetrio et al., 2019)	9.38	121	451	26.83 %
Shift (Demetrio et al., 2020)	10.18	92	451	20.40 %
Padding (Kolosnjaji et al., 2018)	9.20	0	451	00.00 %

Table 5 shows the results obtained with our proposal and with the state-of-the-art methods previously discussed in Section 4.3. For each of the considered methods we report four columns: the average time per sample (in seconds) needed by each algorithm (Time (s)); the number of not corrupted AEs which successfully evade *MalConv* (Evaded); the total number of samples tested (Total); and the evasion rate (Ev. Rate (%)). As it can be observed, some methods (our proposal, Gamma, Extend, and Padding) have reduced their effectiveness on this dataset, while others have shown a better performance (Full DOS, Partial DOS, and Shift). The evasion rate of Full DOS in comparison with the evaluation of Section 4.3 is particularly significant, with an improvement of 19.27 %. Although the top methods have reduced their effectiveness on this dataset, the difference between the two best methods, ours and Gamma, has increased from 11.74 % to 17.29 %. These results complement the evaluation of the methods presented in Section 4.3 and further validate the advantages of our method.

4.6. Limitations and threats to validity

Our results are promising but might be limited by either the size or the content of the dataset used for our experimentation. Despite our efforts to guarantee representativeness and accuracy in our collection methodology, the resulting dataset might still contain unknown biases. In this regard, we tried to analyze if our method performs particularly better or worse on specific types of malware. In particular, we analyzed the following properties to identify any particularities among the samples for which our method did not produce AEs: size, submission date to VirusTotal, number of sections, presence of virtual sections, presence of known packers, and type of malware. Unfortunately, we were unable to reach any conclusive findings. Similarly, after analyzing the samples for which our method does not successfully generate AEs, we were unable to identify any common characteristics among them. This lack of explainability is common to both detectors and automatic methods to generate AEs: our model also behaves as a black box and it is not possible to determine which part of the input perturbations produce the actual improvement in the results.

Despite the fact that MalConv has been commonly used for evaluation purposes on related work, some weaknesses have been highlighted that might affect the validity of the obtained results (Krčál et al., 2018; Le et al., 2018). These shortcomings might be addressed by using a different DL model. However, although there exist previous works available using ML approaches for malware detection, they are not focused on raw bytes or do not offer a pretrained model (Gibert et al., 2018, 2020; Qi et al., 2021). We leave this question as future work and make our code and data publicly available to facilitate its testing with other trained models.

The comparison we do with commercial AV engines in VirusTotal must be taken in light of known shortcomings of the use of VirusTotal for academic research, as it is pointed out in Peng et al. (2019) and Zhu et al. (2020). This includes both observational biases (e.g., different commercial AVs relying on the same underlying detection engine) and AV configuration issues, among others.

5. Conclusions

In this paper, we have proposed a general method to craft Adversarial Examples for Machine Learning-based malware detectors. In particular, we focus on models which operate on raw bytes. Our proposal is based on a black-box setting, where the algorithm can observe the output of the Machine Learning detector. The system receives a binary as input and produces a score as output. Specifically, we propose a two-phase method to design the AEs. First, we design a method to introduce unused spaces, known as code caves, inside a PE file which do not alter the functionality of the original binary. Moreover, the size of these unused spaces is dynamically determined. Second, the content of the introduced spaces is optimized with a Genetic Algorithm. Therefore, the importance of the proposed method is related to its ability to determine, for each individual code cave, the smallest size needed, the right location, and the adequate content, that will result in AEs of higher quality. Furthermore, we handle the previous tasks as an optimization problem and show that the use of efficient fine-tuned optimization techniques can solve them with a better evasion rate than previous methods. To do so, the proposed approach has been evaluated on a well-known state-of-the-art Deep Learning architecture, *MalConv*, achieving an evasion rate of 97.99 %. Moreover, we favorably compared our approach with seven methods available in the state of the art for the same task. We also showed that the increment in the size needed to craft an AE with the proposed procedure is less than 1 % in more than half of the tested samples. In addition, given the current threat landscape, we evaluated our method over a set of modern ransomware samples, achieving similar results.

Then, we tested the crafted AEs over a set of commercial antiviruses, obtaining an average decrease of 5 detections. This fact suggests the possibility of transferability of our proposal to the industry. Thus, this method could be used in combination with other approaches to further reduce the detection rate of commercial anti-viruses.

Finally, we used the generated AEs to retrain the original model, resulting in an increase of its detection rate. Then, we used the resulting model to reevaluate the performance of our proposal. In this experiment, we observed an increase in the difficulty of crafting successful AEs (either in time and size). Furthermore, the retrained model detected malware samples that were not detected by MalConv originally. These results indicate that retraining ML malware detectors with the generated AEs might increase the robustness of the ML models. However, the impact of the technique might be limited by the number of different techniques used for the generation of AEs and the number of ML models evaluated. An interesting future research line would be exploring the evaluation of Adversarial Retraining including other AE generation methods and further ML models.

Despite the fact that the particular technique proposed in this paper is applied for raw bytes, the general idea of performing a perturbation of a sample and, based on heuristic optimization, determining the size, location, and content of the perturbation can

be extended to target either other ML-based detectors or other file formats.

ML-based malware detectors in isolation are not currently the definitive solution in the malware detection domain. However, they are already playing an important role in the detection pipeline. In this context, the general method proposed in this paper to design AEs can be easily used to enhance the performance of anti-virus software. Specifically, it could help to increase their robustness via adversarial retraining.

To improve the reproducibility of our experiments and to foster future research in this area, we make publicly available both the source code and the data used in this work.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Javier Yuste: Conceptualization, Investigation, Data curation, Methodology, Software, Writing – original draft. **Eduardo G. Pardo:** Conceptualization, Investigation, Validation, Writing – original draft. **Juan Tapiador:** Supervision, Formal analysis, Writing – review & editing, Funding acquisition.

Acknowledgements

This research was supported by the Ministerio de Ciencia, Innovación y Universidades (Grant Refs. PGC2018-095322-B-C22 and PID2019-111429RB-C21), by the Region of Madrid grant CYNAMON-CM (P2018/TCS-4566), co-financed by European Structural Funds ESF and FEDER, and the Excellence Program EPUC3M17. The opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect those of any of the funders.

References

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

Aleshkin, A., Lesko, S., 2019. Predicting the growth of total number of users, devices and epidemics of malware in internet based on analysis of statistics with the detection of near-periodic growth features. In: 2019 XXI International Conference Complex Systems: Control and Modeling Problems (CSCMP), pp. 347–352.

Amsaleg, L., Bailey, J., Barbe, A., Erfani, S.M., Furon, T., Houle, M.E., Radovanović, M., Nguyen, X.V., 2021. High intrinsic dimensionality facilitates adversarial attack: theoretical evidence. IEEE Trans. Inf. Forensics Secur. 16, 854–865. doi:10.1109/TIFS.2020.3023274.

Anderson, H.S., Kharkar, A., Filar, B., Roth, P., 2017. Evading machine learning malware detection. Black Hat.

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

Anderson, R., Barton, C., Bölme, R., Clayton, R., Ganán, C., Grasso, T., Levi, M., Moore, T., Vasek, M., 2019. Measuring the changing cost of cybercrime. The 18th Annual Workshop on the Economics of Information Security.

Bazrafshan, Z., Hashemi, H., Fard, S.M.H., Hamzeh, A., 2013. A survey on heuristic malware detection techniques. In: The 5th Conference on Information and Knowledge Technology. IEEE, pp. 113–120.

Biggio, B., Corona, I., Maiorca, D., Nelson, B., Srndic, N., Laskov, P., Giacinto, G., Roli, F., 2013. Evasion attacks against machine learning at test time. In: ECML/PKDD (3). Springer, pp. 387–402.

Biggio, B., Fabio, R., 2018. Wild patterns: ten years af- ter the rise of adversarial machine learning. Pattern Recognit 84, 317,331.

Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H., 2008. Automatically identifying trigger-based behavior in malware. In: Botnet Detection. Springer, pp. 65–88.

Chakraborty, A., Alam, M., Dey, V., Chattopadhyay, A., Mukhopadhyay, D., 2018. Adversarial attacks and defences: a survey. arXiv preprint arXiv: 1810.00069.

Chen, L., Ye, Y., Bourlai, T., 2017. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In: 2017 European Intelligence and Security Informatics Conference (EISIC). IEEE, pp. 99–106.

Chronicle, 2004-. VirusTotal. https://www.virustotal.com/. [Online; accessed 13-June-2020].

Connolly, L.Y., Wall, D.S., 2019. The rise of crypto-ransomware in a changing cybercrime landscape: taxonomising countermeasures. Computers & Security 87, 101568.

Coull, S., Gardner, C., 2018. What are Deep Neural Networks Learning About Malware?https://www.fireeye.com/blog/threat-research/2018/12/what-are-deep-neural-networks-learning-about-malware.html. [Online; accessed 12-June-2020].

Das, S., 2019. A machine learning model for detecting respiratory problems using voice recognition. In: 2019 IEEE 5th International Conference for Convergence in Technology (I2CT). IEEE, pp. 1–3.

Demetrio, L., Biggio, B., 2021. Secml-malware: a python library for adversarial robustness evaluation of windows malware classifiers. arXiv preprint arXiv:2104.12848.

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., 2019. Explaining vulnerabilities of deep learning to adversarial malware binaries. arXiv preprint arXiv:1901.03583.

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., 2021. Functionality-preserving black-box optimization of adversarial windows malware. IEEE Trans. Inf. Forensics Secur. 16, 3469–3478. doi:10.1109/TIFS.2021.3082330.

Demetrio, L., Coull, S.E., Biggio, B., Lagorio, G., Armando, A., Roli, F., 2020. Adversarial EXEmples: a survey and experimental evaluation of practical attacks on machine learning for windows malware detection. arXiv preprint arXiv:2008.07125.

El-Bakry, H.M., 2010. Fast virus detection by using high speed time delay neural networks. Journal in computer virology 6 (2), 115–122.

Firdausi, I., Lim, C., Erwin, A., Nugroho, A.S., 2010. Analysis of machine learning techniques used in behavior-based malware detection. In: 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies, pp. 201–203.

Gandotra, E., Bansal, D., Sofat, S., 2014. Malware analysis and classification: a survey. Journal of Information Security 2014.

, 2010. In: Gendreau, M., Potvin, J.-Y. (Eds.), Handbook of metaheuristics, Vol. 2. Springer.

Gibert, D., Mateu, C., Planes, J., 2018. An end-to-end deep learning architecture for classification of malwares binary content. In: International Conference on Artificial Neural Networks. Springer, pp. 383–391.

Gibert, D., Mateu, C., Planes, J., 2020. Hydra: a multimodal deep learning framework for malware classification. Computers & Security 95, 101873.

Gibert, D., Mateu, C., Planes, J., 2020. The rise of machine learning for detection and classification of malware: research developments, trends and challenges. Journal of Network and Computer Applications 153, 102526. doi:10.1016/j.jnca.2019.102526.

Hashemi, A.S., Mozaffari, S., 2019. Secure deep neural networks using adversarial image generation and training with noise-gan. Computers & Security 86, 372–387.

Hojjatinia, S., Hamzenejadi, S., Mohseni, H., 2019. Android botnet detection using convolutional neural networks. arXiv preprint arXiv:1911.12457.

Huang, K., Siegel, M., Madnick, S., 2018. Systematically understanding the cyber attack business: a survey. ACM Computing Surveys (CSUR) 51 (4), 1–36.

Kaspersky, K., 2005. Hacker debugging uncovered (uncovered series). A-List Publishing.

Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F., 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In: 2018 26th European Signal Processing Conference (EUSIPCO). IEEE, pp. 533–537.

Kolosnjaji, B., Zarras, A., Webster, G., Eckert, C., 2016. Deep learning for classification of malware system call sequences. In: Australasian Joint Conference on Artificial Intelligence. Springer, pp. 137–149.

Kornblum, J., 2006. Identifying almost identical files using context triggered piecewise hashing. Digital Invest. 3, 91–97.

Kreuk, F., Barak, A., Aviv-Reuven, S., Baruch, M., Pinkas, B., Keshet, J., 2018. Deceiving end-to-end deep learning malware detectors using adversarial examples. arXiv preprint arXiv:1802.04528.

Krčál, M., Švec, O., Bálek, M., Jašek, O., 2018. Deep convolutional malware classifiers can learn from raw executables and labels only. ICLR.

Le, Q., Boydell, O., Mac Namee, B., Scanlon, M., 2018. Deep learning at the shallow end: malware classification for non-domain experts. Digital Invest. 26, S118–S126.

Lee, W.Y., Saxe, J., Harang, R., 2019. Seqdroid: Obfuscated Android Malware Detection Using Stacked Convolutional and Recurrent Neural Networks. In: Deep Learning Applications for Cyber Security. Springer, pp. 197–210.

Liu, X., Lin, Y., Li, H., Zhang, J., 2020. A novel method for malware detection on ml-based visualization technique. Computers & Security 89, 101682.

Millar, S., McLaughlin, N., Martinez del Rincon, J., Miller, P., Zhao, Z., 2020. Dandroid: A multi-view discriminative adversarial network for obfuscated android malware detection. In: Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy, pp. 353–364.

Miller, B.L., Goldberg, D.E., et al., 1995. Genetic algorithms, tournament selection, and the effects of noise. Complex systems 9 (3), 193–212.

Oliveto, P.S., Paixão, T., Heredia, J.P., Sudholt, D., Trubenová, B., 2018. How to escape local optima in black box optimisation: when non-elitism outperforms elitism. Algorithmica 80 (5), 1604–1633.

Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z.B., Swami, A., 2016. The limitations of deep learning in adversarial settings. In: 2016 IEEE European symposium on security and privacy (EuroS&P). IEEE, pp. 372–387.
Peng, P., Yang, L., Song, L., Wang, G., 2019. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In: Proceedings of the Internet Measurement Conference. Association for Computing Machinery, pp. 478–485.
Pierazzi, F., Pendlebury, F., Cortellazzi, J., Cavallaro, L., 2019. Intriguing properties of adversarial ml attacks in the problem space. arXiv preprint arXiv:1911.02142.
Qi, P., Zhang, Z., Wang, W., Yao, C., 2021. Malware detection by exploiting deep learning over binary programs. In: 2020 25th International Conference on Pattern Recognition (ICPR). IEEE, pp. 9068–9075.
Radwan, A.M., 2019. Machine learning techniques to detect maliciousness of portable executable files. In: 2019 International Conference on Promising Electronic Technologies (ICPET). IEEE, pp. 86–90.
Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.K., 2018. Malware detection by eating a whole exe. In: Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence.
Raff, E., Sylvester, J., Nicholas, C., 2017. Learning the pe header, malware detection with minimal domain knowledge. In: Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, pp. 121–132.
Roberts, M., 2020. VirusShare. https://virusshare.com/. [Online; accessed 12-June-2020].
Roth, P., Anderson, H., Cattell, S., 2019. Extending EMBER. https://www.endgame.com/blog/technical-blog/extending-ember. [Online; accessed 12-June-2020].
Sahay, S.K., Sharma, A., Rathore, H., 2020. Evolution of Malware and Its Detection Techniques. In: Information and Communication Technology for Sustainable Development. Springer, pp. 139–150.
Saxe, J., Berlin, K., 2015. Deep neural network based malware detection using two dimensional binary program features. In: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE). IEEE, pp. 11–20.
Shah, S., Jani, H., Shetty, S., Bhowmick, K., 2013. Virus detection using artificial neural networks. Int J Comput Appl 84 (5).
Sharif, M., Lucas, K., Bauer, L., Reiter, M.K., Shintre, S., 2019. Optimization-guided binary diversification to mislead neural networks for malware detection. arXiv preprint arXiv:1912.09064.
Simonyan, K., Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.
Singh, A., 2009. Portable Executable File Format. In: Identifying Malicious Code Through Reverse Engineering. Springer, pp. 1–15.
Smith, M.R., Johnson, N.T., Ingram, J.B., Carbajal, A.J., Haus, B.I., Domschot, E., Ramyaa, R., Lamb, C.C., Verzi, S.J., Kegelmeyer, W.P., 2020. Mind the Gap: On Bridging the Semantic Gap between Machine Learning and Malware Analysis. In: Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security. Association for Computing Machinery, New York, NY, USA, pp. 49–60. doi:10.1145/3411508.3421373.
Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R., 2014. Intriguing properties of neural networks. In: 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14–16, 2014, Conference Track Proceedings.
Szor, P., 2005. The art of computer virus research and defense. Pearson Education.
Wang, R., Zhu, Y., Tan, J., Zhou, B., 2017. Detection of malicious web pages based on hybrid analysis. Journal of Information Security and Applications 35, 68–74.
Xue, H., Sun, S., Venkataramani, G., Lan, T., 2019. Machine learning-based analysis of program binaries: a comprehensive study. IEEE Access 7, 65889–65912.
Yosifovich, P., Solomon, D.A., Ionescu, A., 2017. Windows internals, part 1: System architecture, processes, threads, memory management, and more. Microsoft Press.
Young, T., Hazarika, D., Poria, S., Cambria, E., 2018. Recent trends in deep learning based natural language processing. IEEE Comput Intell Mag 13 (3), 55–75.
Yuan, X., He, P., Zhu, Q., Li, X., 2019. Adversarial examples: attacks and defenses for deep learning. IEEE Trans Neural Netw Learn Syst 30 (9), 2805–2824.
Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., Wang, G., 2020. Measuring and modeling the label dynamics of online anti-malware engines. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 2361–2378.

Javier Yuste is a Ph.D. student at Universidad Rey Juan Carlos, Madrid, where he is part of the Group for Research in Algorithms For Optimization (GRAFO). His research interests focus on the applicability of Artificial Intelligence (AI) techniques to solve cybersecurity problems. In particular, the intersection between AI techniques and analysis, detection and evasion of malicious software.

Eduardo G. Pardo is an Associate Professor at Universidad Rey Juan Carlos (Madrid, Spain). He got his Ph.D. in 2011 in the field of heuristic optimization and he is co-author of many journal papers related to this topic. Additionally, Eduardo is founding member of the Group for Research in Algorithms For Optimization (GRAFO) at Universidad Rey Juan Carlos. Among his research interests he includes the development and application of efficient Artificial Intelligence techniques to real-world problems, where we can find problems related to security.

Juan Tapiador is a Professor of Computer Science with the Universidad Carlos III de Madrid, Spain, where he leads the Computer Security Lab. His research interests include binary analysis, systems security, privacy, surveillance, and cybercrime. He has served in the technical committee of conferences, such as USENIX Security, ACSAC, DIMVA, ESORICS, and AsiaCCS. He has been a recipient of the UC3M Early Career Award for Excellence in Research in 2013, the Best Practical Paper Award at the 41st IEEE Symposium on Security and Privacy (Oakland), the CNIL-Inria 2019 Privacy Protection Prize, and the 2019 AEPD Emilio Aced Prize for Privacy Research. His work has been covered by international media, including The Times, Wired, Le Figaro, ZDNet, and The Register.