machine learning & knowledge extraction

MDPI

Article

On Deceiving Malware Classification with Section Injection

Adeilson Antonio da Silva [1] and Mauricio Pamplona Segundo [2,*]

[1] Computer Institute, Federal University of Bahia (UFBA), Salvador 40170-110, Brazil
[2] Computer Science and Engineering, University of South Florida (USF), Tampa, FL 33620, USA
* Correspondence: mauriciop@usf.edu

Abstract: We investigate how to modify executable files to deceive malware classification systems. This work's main contribution is a methodology to inject bytes across a malware file randomly and use it both as an attack to decrease classification accuracy but also as a defensive method, augmenting the data available for training. It respects the operating system file format to make sure the malware will still execute after our injection and will not change its behavior. We reproduced five state-of-the-art malware classification approaches to evaluate our injection scheme: one based on Global Image Descriptor (GIST) + K-Nearest-Neighbors (KNN), three Convolutional Neural Network (CNN) variations and one Gated CNN. We performed our experiments on a public dataset with 9339 malware samples from 25 different families. Our results show that a mere increase of 7% in the malware size causes an accuracy drop between 25% and 40% for malware family classification. They show that an automatic malware classification system may not be as trustworthy as initially reported in the literature. We also evaluate using modified malware alongside the original ones to increase networks robustness against the mentioned attacks. The results show that a combination of reordering malware sections and injecting random data can improve the overall performance of the classification. All the code is publicly available.

Keywords: malware classification; adversarial examples; Deep Learning; Convolutional Neural Networks

Citation: da Silva, A.A.; Pamplona Segundo, M. On Deceiving Malware Classification with Section Injection. *Mach. Learn. Knowl. Extr.* **2023**, *5*, 144–168. https://doi.org/10.3390/make5010009

Academic Editor: Ramón Alberto Mollineda Cárdenas

Received: 14 November 2022
Revised: 11 January 2023
Accepted: 11 January 2023
Published: 16 January 2023

1. Introduction

Malware—a short term for malicious software—is described by Sikorski et al. [1] by their action:

> Any software that does something that causes harm to a user, computer, or network can be considered malware [...]

These applications, purposedly built with intentions of reading, copying, or modifying information from computer systems—often without user consent—pose a high threat for modern information systems [2–7]. The early detection of such malware is vital to minimize their effects on an organization or even among regular users.

In this work, we discuss strategies related to the classification (i.e., which kind of malware is it?) of malware samples using only their raw bytes as inputs to machine learning algorithms. These strategies can be seen as part of the static analysis of samples, an especially important stage in a malware detection pipeline, in which it is necessary to provide the classification without executing the file being analyzed. It is important to stress that these methodologies are not to be used as the sole strategy to detect malware samples but as the first one in a multi-step chain of procedures. Despite that, due to their fast execution times and lack of human interaction, they are still an integral part of such a pipeline [2,8,9].

We present here a straightforward way to modify a software file to deceive systems built to classify malware examples into families. Our method builds upon the idea of injecting bytes into the executable file [10]. We seek to insert bytes in various parts of a

malware. By doing so, we aim to deceive malware classifiers and preserve the original functionality while hindering the detection of injected data. To accomplish that, we create rules of injection that respect the file format of the operating system the malware will infect. We can not only define how many bytes we inject but also how they spread over the file. More importantly, we explore two approaches:

Random injection: inserting random bytes, so that we do not require any knowledge about the systems to be deceived

Adversarial injection: inserting bytes taken from families different from the sample being evaluated.

The classification approaches evaluated in this work are based on methods that learn straight from the raw bytes of the file, ranging from methodologies that reinterpret the sample as a grayscale image up to preprocessing each sample as a 1D vector in their execution [9,11–20]. We want to evaluate the vulnerability of these variants to the already known adversarial examples [21], which is an approach with increasing popularity in the literature, especially in the context of malware [10,12,19,20,22–25]. There are some limitations that must be observed, though, since the perturbations added to malware samples must be drawn from a discrete domain. It differs from other types of data, such as images. In addition, executable files have strict standards, which means byte ordering is relevant in some parts of the file. As mentioned earlier, we limited our manipulations to the expected standards in order to preserve the functionality of the malware samples.

In that sense, we investigate both sides of this problem: we explore an **attack** method that modifies input samples to deceive the classification algorithms, but we also evaluate the robustness of **defense** mechanisms to mitigate those attacks. Several methods have been proposed recently to reduce the effects of adversarial attacks in multiple domains [26–30]—by trying to distinguish a crafted input from a real one, reducing their effects on the network output by modifying the optimization during training, or even trying to totally remove the adversarial perturbations from the input. Nevertheless, this still remains an open problem in the malware domain. With this work, we attempt to provide some directions to the mitigation of adversarial attacks against malware classification systems.

The rest of this paper is presented as follows: in Section 3, we compare our methodologies to others present in the literature. In Section 4, we present how we generate and add data between sections of a Portable Executable (PE) file. Section 5 discusses the machine learning algorithms evaluated in this paper for malware classification. In Section 6, we discuss our evaluation strategies and their results, finishing with our conclusions in Section 7.

Our contributions can be summarized as follows:

1. We provide a framework to inject data into PE files that leverages all the alignments required to preserve its functionality. It can inject any sort of data (either random or from a different file) in multiple positions of the file, not only at the end (padding).
2. We evaluate how different deep neural networks architectures proposed for malware classification behave in multiclass classification scenarios. We want to assess the difficulties behind separating a given sample from other samples of the same kind.
3. We evaluate how the aforementioned networks behave when dealing with injected samples. Our goal here is to assess how our attacks impact the classification of these networks in regard to both the location and also the amount of injected data.
4. We evaluate different augmentation strategies for defending against our data injection scheme, amplifying the robustness of malware classification techniques using raw bytes.

2. Background

Modern operating systems, such as Linux and Microsoft Windows, use the concept of sections to read an executable file, load it to the memory, and run its instructions. It

is necessary to know and follow the file format specifications to be able to insert data in different parts of an executable and preserve its functionality. Since there are differences between the files accepted by each operating system, designing a system-agnostic injection scheme is impracticable. For this reason, we focus on the Microsoft Windows PE format, as it is the only one included in publicly available malware datasets [11,31].

As shown in Figure 1, PE sections provide information about the executable, such as its instructions (".text"), its variables(".data"), and resources it uses (".rsrc"). Following this layout, our strategy consists of injecting non-executable sections such as ".data" to the file. This way, the set of instructions does not change, and the only way to decide whether an injected section is in use or not is through execution.

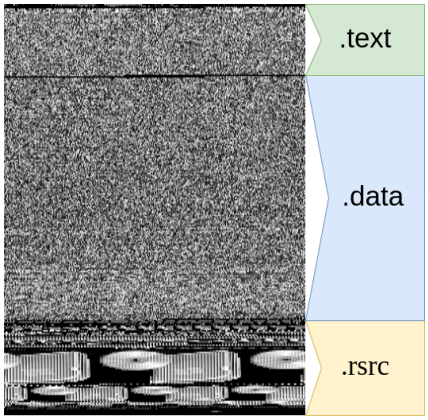


Figure 1. Illustration of the sections of a PE file.

3. Related Works

We start by presenting previous works which performed malware classification. We also discuss other methodologies for malware perturbation and how authors have tried to minimize these effects.

3.1. Malware Classification

Nataraj et al. [11] presented a method to transform software into images and classify them according to their malware family. In this context, a family is a set of software files with high similarity of instructions and behavior. Follow-up works explored this idea using different feature extraction (e.g., GIST, Local Binary Patterns, Scale-Invariant Feature Transform) and classification (e.g., Support Vector Machines, K-Nearest Neighbors (KNN)) methods [32,33].

The growth in Deep Learning research led to the exploration of neural networks for malware classification. Recent works applied different architectures for this task, either by extracting static features from the file (e.g., system calls, imported libraries, functions in use, function call graph) [13,34–38] or by using the raw bytes from the data as input [16,39,40]. Other strategies try to combine the extraction of static features with information taken from the dynamic analysis of samples; i.e., they are executed in a controlled environment, and the effects caused on the operational system are then used as features to classify the input [2,41,42]. Some of them achieve high classification accuracy by training Convolutional Neural Networks (CNNs) from scratch [18,33,41] or by using prior knowledge from a CNN pre-trained on a large dataset [9,14,15] such as ImageNet [43]. Malware detection was also exploited in the form of a binary classification by considering all malware files as one class and samples of benign software as the other one [15,16,37,38,41].

These networks, however, are vulnerable to adversarial attacks, which means that tampering with the data structure of a malware sample before it is analyzed can lead to substantial misclassification, completely overruling the original purpose of an automatic classification system. In this work, we explore different architectures—KNN+GIST as proposed by Nataraj et al. (2011) [11], CNN, CNN-LSTM and CNN BiLSTM as proposed

by Le et al. (2018) [17] and MalConv as proposed by Raff (2017) [16]—and how they behave against crafted adversarial samples. Section 5 provides the reasoning behind this choice. In this work, we evaluate the two aspects of adversarial training by attacking these models with handcrafted samples and also retraining them with augmented data to make them more robust against the injected samples.

It is worth mentioning that there are few relevant public datasets for training malware classifiers, which makes comparing different works a more subtle task. Malimg [11], BIG 2015 [31] and EMBER [36] are the most notable ones. Since our injection method require reading the file header, the BIG 2015 [31] dataset is not possible because the samples have their headers stripped. EMBER [36], on the other hand, does not provide raw byte values straight away. Since they provide SHA-256 values taken from file contents, a reverse search in malware indexing services is needed in order to retrieve their raw bytes. In Table 1, we aggregate state-of-the-art methods for malware detection and classification by their technique and the dataset it used.

Table 1. Summary of different malware classification techniques.

Author	Technique	Dataset
Nataraj et al. (2011) [11]	GIST + KNN	malimg [11]
Pascanu et al. (2015) [34]	Echo State Network (ESN) + Logistic Regression	Private
Athiwaratkun and Stokes (2017) [13]	LSTM + Multilayer Perceptron (MLP)	Private
Yue (2017) [14]	CNN	malimg [11]
Raff (2017) [16]	Embedding + CNN	Private
Anderson (2018) [36]	Embedding + CNN	Ember [36]
Su et al. (2018) [18]	CNN	Private
HaddadPajouh et al. (2018) [40]	LSTM	Private
Liu et al. (2018) [33]	Multilayer SIFT	malimg [11], BIG 2015 [31]
Agarap and Pepito (2018) [32]	Gated Recurrent UNIT (GRU) + Support Vector Machines (SVM)	malimg [11]
Le (2018) [17]	CNN-BiLSTM	BIG 2015 [31]
Chen (2018) [15]	Inception-V1 [44]	malimg [11], BIG 2015 [31]
Vinayakumar et al. (2019) [41]	CNN	malimg [11], Ember [36], Private
Chen (2020) [9]	Inception-V1 [44]	Private
Gao et al. (2022) [38]	Graph Isomorphism Network (GIN)	Private

3.2. Malware Injection

Adversarial attacks consist of adding tiny changes to the input data to alter its classification result and are usually not easily perceived by humans. However, arbitrarily modifying software files without changing its behavior is impossible. Even verifying if a modification does not affect a software's response is an undecidable problem. Thus, if someone arbitrarily alters a malware to change its classification results, there is a chance it will no longer pose a threat to the system. Despite that fact, there exists in the literature some possible attacks that retain their functionalities. They are illustrated in Figure 2.

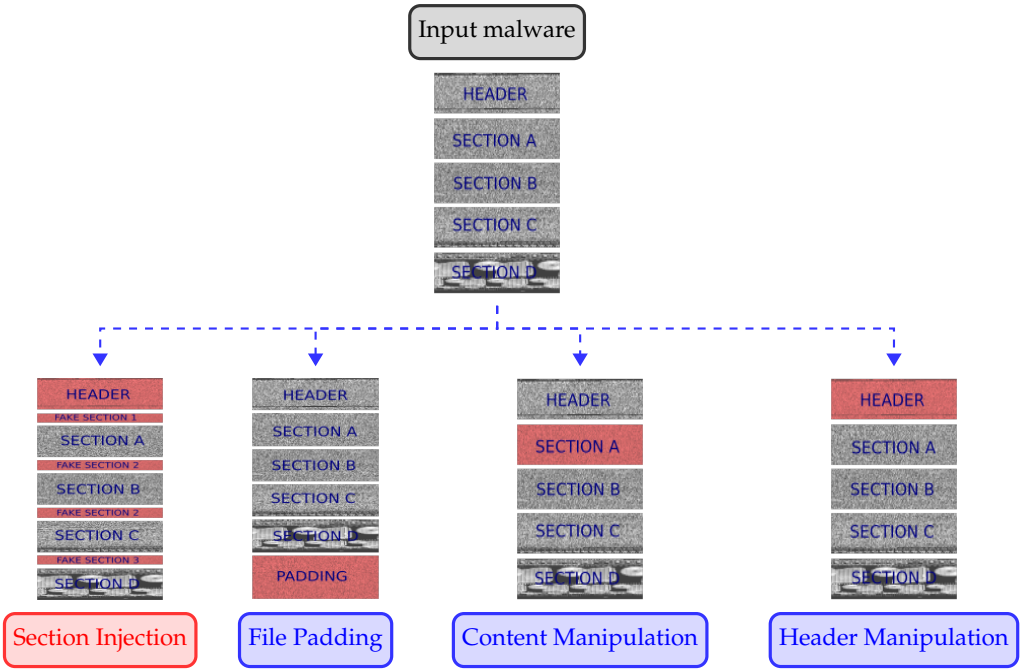


Figure 2. Illustration of the differences among attacks to image-based malware classifiers. The leftmost (red) square displays our approach. Blue squares displays previous attacks.

Different works exploited adversarial attacks in the malware domain. Grosse et al. [12] and Al-Dujaili et al. [22] extracted static features from malware files and used the Fast Gradient Sign Method (FGSM) [21] to modify these feature vectors and form adversarial samples. Notwithstanding, these approaches do not guarantee that it is possible to alter the malware file to produce the adversarial feature vector while maintaining the original functionality. Therefore, they may not have a practical use.

Anderson et al. [10] explores a black box attack against a reinforcement learning model, where the agent actions are taken from a list of modifications that includes manipulating existing bytes but also adding ones between sections or even creating new sections. No further information is provided regarding the constraints on these injections. It fits the "Section Injection" and "Content Manipulation" categories illustrated in Figure 2. It achieves evasion rates up to 16% against a Gradient Boost Decision Tree (GBDT) [36] model.

Khormali et al. [19] focused on injecting bytes to the executable files' end, which is an unreachable area during execution. It fits the "File Padding" category illustrated in Figure 2. As the operating system will not execute it and not even read it in some cases, it does not affect the malware behavior. These bytes can either be generated by FGSM or be parts of other malware. Nevertheless, extra bytes at the end of the file may be easy to detect and discard before the classification. This approach requires access to the model or training data used by the classification system, which may not be available in a real attacking scenario.

Demetrio et al. [23] propose a black-box attack called GAMMA (Genetic Adversarial Machine Learning Malware Attack), which is a method that queries a given malware classifier and based on the output, draws from a set of functionality-preserving manipulations that changes malware samples iteratively. GAMMA is evaluated against two malware classifiers, Malconv [16]—a shallow neural network—and GBDT [36]. Its proposed methods fit all the categories illustrated in Figure 2, despite not detailing how some of those are achieved.

Lucas et al. [25] also employ functionality-preserving techniques. They extend binary rewriting techniques such as in-place randomization (IPR) [45]—where the binary is disassembled and some of its instructions are rewritten—and code displacement (Disp) [46]—where the disassembled version is also used but with the intent of moving instructions between

sections, fitting into the "Content Manipulation" category illustrated in Figure 2. They apply these attacks in an interactive manner and evaluate them against three neural networks, achieving a misclassification rate of over 80% in some scenarios.

Benkraouda et al. [20] proposes a framework that mixes a mask generator to highlight the bytes that are possible to manipulate while retaining executability, adversarial example generation using a Carlini–Wagner (CW) attack [47] and an optimization step that iteratively modifies the masked bytes by comparing the generated adversarial data to a set of known instructions. It fits the "Content Manipulation" category illustrated in Figure 2. The attack is evaluated against a three-layer CNN, achieving an attack success rate of up to 81.8%. A shortcoming of this method is the time it takes to generate its samples, reaching over six hours for a single sample in some cases.

Demetrio et al. [24] introduce the RAMEN framework, an extensive library with multiple attacks for malware classification. They present three novel attacks—Full DOS, Extend and Shift—all of them capable of modifying the binary sample while keeping its functionality. The novel attacks are evaluated against MalConv [16], DNN with Linear (DNN-Lin) and ReLU (DNN-ReLU) [48] and GBDT [36], being misclassified by the neural networks but not being able to evade the Decision Tree since it does not rely only on static data.

Our attack scheme—Section Injection—is also explored by Anderson et al. [10] and Demetrio et al. [23] as one possible method in their pipelines, but no further information is provided regarding the constraints for this injection. It can also be seen as an ensemble of the *Extend* and *Shift* methods proposed by Demetrio et al. [24] and the *padding* methods discussed by Khormali et al. [19]. The byte modifications presented by Lucas et al. [25] can also be integrated in our method, leading to the injection of perturbed sections instead of random ones.

Regarding the data used to evaluate the attacks, most of the works listed here used some sort of private dataset either by collecting samples from malware hosting services or expanding public ones—Benkraouda et al. [20] merged malimg [11] and benign samples from the Architecture Object Code Dataset (AOCD) [49], while Khormali et al. [19] used BIG 2015 [31] and also formed a private IoT dataset. A summary of the functionality-preserving attacks can be found in Table 2.

Table 2. Summary of functionality-preserving attacks against PE malware classification.

Author	Methods	Targets	Dataset
Anderson et al. [10]	Set of Manipulations	GBDT [36]	Private
Khormali et al. [19]	Padding	3-layer CNN	BIG 2015 [31] + Private IoT dataset
Demetrio et al. [23]	Set of Manipulations	MalConv [16], GBDT [36]	Private
Demetrio et al. [24]	Partial DOS, Full DOS, Extend, Shift, FGSM, Padding	MalConv [16], DNN [48], GBDT [36]	Private
Lucas et al. [25]	IPR, Disp	AvastNet [50], MalConv [16], GBDT [36]	Private
Benkraouda et al. [20]	Adversarial Generation + Optimization	CNN [19,51]	Private (combination of malimg [11] and AOCD [49])

4. Data Injection

To comply with a realistic usage scenario, we inject one or more sections filled with arbitrary bytes before any processing is completed for classification purposes, as illustrated in Figure 3. We explain how the proposed injection process works in the following sections, and we show how we built the malware classifiers used in our experiments in Section 5.

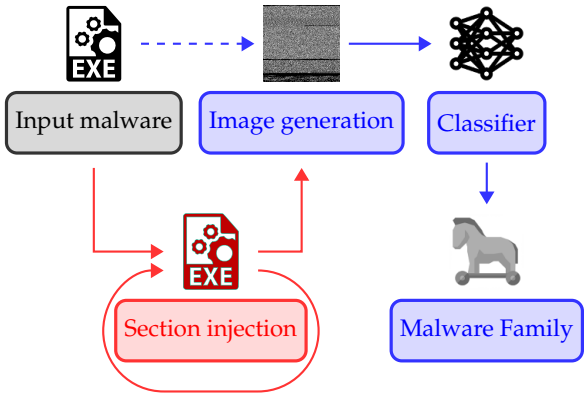


Figure 3. Flowchart of an image-based malware classification system (blue lines). Red lines replace the dashed blue line in our data injection scheme.

4.1. File Header

The first step in our injection scheme is to obtain information about the input file by reading its header. Table 3 lists the flags that are relevant to us. After inserting a new section, we need to increment the flag $NumberOfSections$ and update the flag $SizeOfImage$ accordingly to preserve the malware functionality. We pick the injected section's index k by drawing a number in the interval $[0, NumberOfSections]$. Sections 0 to $k-1$ remain in place, and sections k to $NumberOfSections-1$ are shifted one position forward so that we can insert the new section in k-th place.

Table 3. Flags in the header of PE files.

Flag Name	Description
NumberOfSections	Number of sections in the file
FileAlignment	Section size in bytes is a multiple of this flag
SectionAlignment	Memory address of a section is a multiple of this flag
SizeOfImage	Memory size of all sections in bytes
ImageBase	Address of the first byte when the file is loaded to memory (default value is 0x00400000)

4.2. Section Header

A section header is composed of 40 contiguous bytes. These bytes specify what the loader needs to handle this section. Table 4 shows the bytes that we fill when creating a new section. We refer to a flag of the i-th section as $Flag_i$.

Table 4. Flags in section headers of PE files.

Flag Name	Size	Description
Name	8 bytes	Section name
VirtualSize	4 bytes	Section size in bytes on memory
VirtualAddress	4 bytes	Section offset on memory relative to *ImageBase*
SizeOfRawData	4 bytes	Section size in bytes on disk
PointerToRawData	4 bytes	Section offset on disk relative to the beginning of the file
Characteristics	4 bytes	Section characteristics like usage and permissions

First, we generate eight random printable characters (ASCII table values between 33 and 126) as $Name_k$. After that, we set $SizeOfRawData_k$ using Equation (1):

$$SizeOfRawData_k = \lceil \frac{N}{FileAlignment} \rceil \times FileAlignment \quad (1)$$

with N being the number of bytes we want to add. We always set N as a multiple of $FileAlignment$ so that null padding is unnecessary. $FileAlignment$ is usually 512 bytes, but it varies according to compilation options.

$PointerToRawData_k$ is set as in Equation (2) if the k-th section is the last one. Otherwise, it is set as in Equation (3), and we add $SizeOfRawData_k$ to $PointerToRawData_i$, $\forall i > k$.

$$PointerToRawData_k = PointerToRawData_{k-1} + SizeOfRawData_{k-1} \quad (2)$$

$$PointerToRawData_k = PointerToRawData_{k+1} \quad (3)$$

On memory, we inject sections after every other section to avoid having to update instructions that use memory offsets and preserve the execution path. We set $VirtualAddress_k$ using Equation (4):

$$VirtualAddress_k = \lceil \frac{VirtualAddress_L + VirtualSize_L}{SectionAlignment} \rceil \times SectionAlignment \quad (4)$$

where L is the index of the last section on memory. This way, we correctly align the injected section according to $SectionAlingment$.

$VirtualSize_k$ is set to 0, as we do not want to take memory space. Thus, multiple runs of this injection process produce sections pointing to the same address. In our tests, this does not affect execution. We finish our header by setting $Characteristics_k$ as a read-only section with initialized data.

4.3. Injected Data

In our work, the injected data are a sequence of random bytes. As we have control of the section structure, we could insert pieces from other executables or adversarial examples created using FGSM as other works in the literature [19]. However, we do not do that because we assume we have no access to models and training data used by malware classifiers. Our results show that our simple strategy is enough to affect the performance of a state-of-the-art malware classification approach substantially.

4.4. Workarounds

We found some challenges when applying this method to an arbitrary PE file. Instead of constraining the input files, we dealt with the problems as they appeared. Some malware instances, usually packed or obfuscated, have multiple contiguous virtual sections that do not exist on disk, only on memory. For those cases, we had to adjust the *PointerToRawData* in injected data to make sure it points to a valid physical section. Furthermore, malware sections are not always correctly aligned with the *FileAlignment* flag. To avoid fixing existing sections, we only inject data before correctly aligned ones.

5. Malware Classification

As can be seen in Figure 3, this process is divided into two parts: image generation and classification. The former is described in Section 5.1. The latter is carried out with various approaches:

1. GIST + KNN [11], which holds state-of-the-art performance for handcrafted methods [52];
2. Le-CNN, Le-CNN-LSTM, Le-CNN-BiLSTM [17], three similar models that uses resizing of the input data to a fixed number of bytes [53];

3. MalConv [16], a model that truncates the first 1MB and performs classification with it [54]).

Those architectures were chosen because their code base is available publicly, and they meet the hardware specifications on the computers used for the experiments (16 GB RAM, 1 TB hard disk, NVIDIA® GeForce™ RTX 3060 GPU, Intel® Core™ i7-11800H @ 2.30 GHz). Since we want a direct comparison with the original works, we used similar training protocols. They are, respectively, described in Sections 5.2.1 and 5.2.2.

5.1. Image Generation

We transform an executable into an image following Chen's adaptation [15] of Nataraj et al.'s specifications [11]. We treat every byte as a grayscale pixel, and we break the file into image rows by using a fixed width, which is set according to the file size (see Table 5). We discard the last row if it is incomplete. The result is illustrated in Figure 1.

Table 5. Image width based on the executable size [11].

Size (kB)	Width (px)
<10	32
10–30	64
30–60	128
60–100	256
100–200	384
200–500	512
500–1000	768
1000–2000	1024
>2000	2048

5.2. Classification

5.2.1. GIST + KNN

We reproduced Nataraj et al.'s approach [11] to the best of our abilities. To do so, we resize our images to 64×64 pixels, extract 320-dimensional GIST descriptors, and then classify it using KNN with $K = 3$.

5.2.2. CNNs

Several types of neural networks were explored to classify malware files [13–18,34,35,39,40]. However, to the best of our knowledge, CNNs are the ones with the highest accuracy. In this work, we chose different CNN strategies to understand how they perform against data injection:

1. Le et al. [17] present three models. A simple model with three 1D-CNN layers before a fully connected layer is referred to as Le-CNN. A second model with an LSTM layer before the fully connected one is referred to as Le-CNN-LSTM. A third model with a bidirectional LSTM before the fully connected layer is referred to as Le-CNN-BiLSTM. For all of them, we employ the same input size of 10 k bytes, a batch size of 512, and train the model for at most 60 epochs (early stopping if the accuracy does not improve for 10 epochs). Optimization is performed with the Adam algorithm [55] with a learning rate of 1×10^{-4}.

2. Raff et al. [16] present the model referred to as MalConv. This model employs a gated convolution network, i.e., an embedding layer followed by two separate 1D-CNN layers that are multiplied and passed on for two fully connected layers. For this model, we use training protocol similar to Lucas et al. [23]: an input size of 1 MB and training for a total of 10 epochs without early stopping with a batch size of 16 due to memory constraints. Optimization is performed using the Stochastic Gradient Descent (SGD) algorithm with a Nesterov momentum [56] of 9×10^{-1}, weight decay of 1×10^{-3}, and a learning rate of 1×10^{-2}.

All those models use some combination of 1D convolutions and pooling layers to both reduce the dimensionality and also introduce some translation invariance to the model—i.e., being able to detect a feature even if it appears in a different position on the data [57]. It is beneficial in the malware domain because even though the set of instructions can be considered small (bytes 0–255), the context variance is remarkably high, and a given array of bytes can appear in many locations within the file having different meanings.

6. Results

Henceforward, we display our experimental results for different attack and defense scenarios. In Section 6.1, we provide an overview of the chosen dataset and its structure. In Section 6.2, we discuss the reasons for using metrics such as precision–recall over other metrics currently used in other malware injection/classification works, such as accuracy and ROC. Sections 6.3 and 6.4 are the first evaluations on attacking classification models with modified malware samples, and they provide the first insight on how impactful data injection in the performance of our trained models is. In Section 6.5, we try to assess the real importance of the file header as a feature for the classification models; if we strip the header from input files, can they still be correctly classified? In Section 6.6, we try to provide defense mechanisms against the attacks discussed previously in three different fronts: using injected data in the training set in Section 6.6.1, creating a binary dataset by inflating the dataset with benign data in Section 6.6.2 and also finetuning larger models in Section 6.6.3.

6.1. Dataset

The forthcoming experiments were made upon the malimg [11] dataset to evaluate malware classification before and after code injection. Table 6 displays the distribution of samples across all classes. It has 9339 malware samples from 25 families, and the average size of a sample is approximately 176 kB. In this dataset, most of the samples—7475 of them to be precise—have a *FileAlignment* flag of 512 bytes. The second most common value for this flag is 4096 bytes, with 1674 samples, and in third place, 1024 bytes with 190 samples. No other values were found for the *FileAlignment* flag in this dataset.

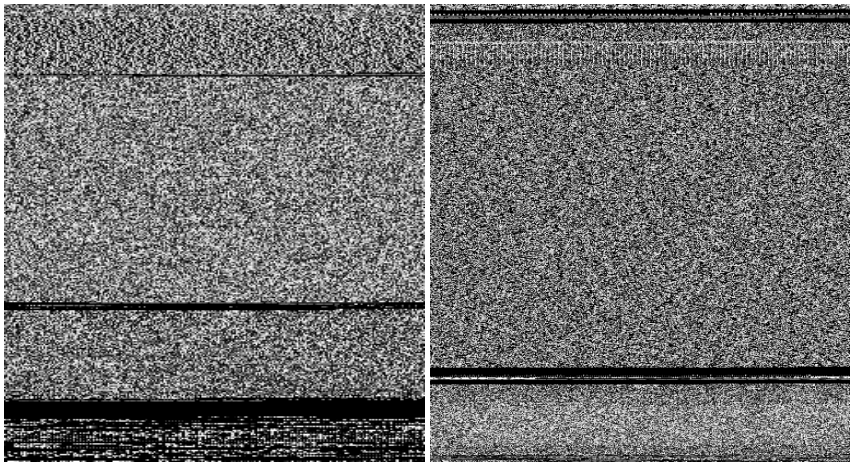
In Figure 4, we display some visual contrast among classes with the most and the least number of samples—Allaple.A and Skintrim.N, respectively—and also with the higher and the smallest average size—VB.AT and Agent.FYI, respectively. The same aspect ratio is kept for all images to highlight the resolution differences created during image generation, as explained in Section 5.1. We can see that the class dissimilarities are mostly represented by lower density sections, i.e., image areas with a higher number of black pixels. Texture can be seen in some samples, such as at the bottom of Figure 4c, which is usually caused by resource sections.

Another feature of this dataset is that most samples present a similar structure to the one illustrated in Figure 1, having *.text*, *.data* and *.rsrc* sections as the most prevalent ones. A few classes, namely Alueron.gen!J, Lolyda.AA1 and Lolyda.AT, present rather uncommon sections either obfuscated or generated with non UTF-8 characters.

Table 6. Samples distribution and average size in malimg [11] dataset.

#	Family	# Samples	Average Size (kB)
1	Adialer.C	122	209.82
2	Agent.FYI	116	16.07
3	Allaple.A	2949	72.64
4	Allaple.L	1591	57.75
5	Alueron.gen!J	198	101.27
6	Autorun.K	106	524.54
7	C2LOP.P	146	386.92
8	C2LOP.gen!g	200	524.04
9	Dialplatform.B	177	13.98
10	Dontovo.A	162	34.50
11	Fakerean	381	110.62
12	Instantaccess	431	173.07
13	Lolyda.AA1	213	27.43
14	Lolyda.AA2	184	35.13
15	Lolyda.AA3	123	244.80
16	Lolyda.AT	159	24.66
17	Malex.gen!J	136	82.96
18	Obfuscator.AD	142	162.82
19	Rbot!gen	158	241.04
20	Skintrim.N	80	192.98
21	Swizzor.gen!E	128	336.74
22	Swizzor.gen!I	132	320.77
23	VB.AT	408	666.80
24	Wintrim.BX	97	408.74
25	Yuner.A	800	524.54
Total	**25 families**	**9339 samples**	**176.29kB size**

(**a**) Allaple.A (**b**) Skintrim.N



(**c**) VB.AT (**d**) Agent.FYI

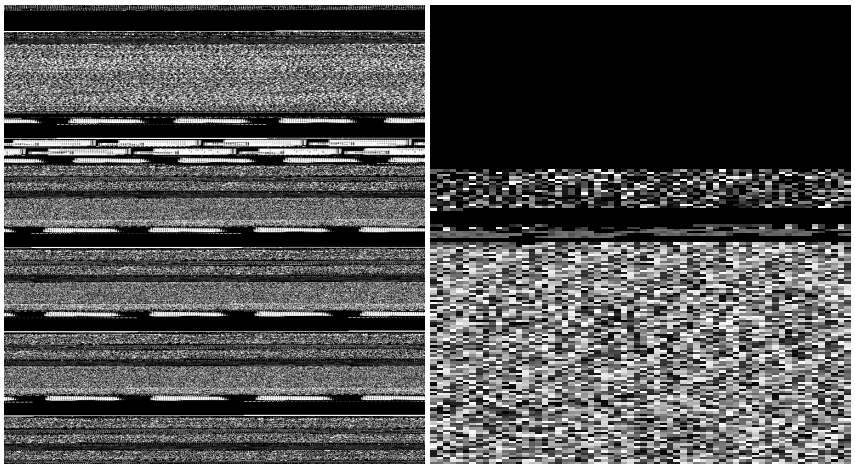


Figure 4. Illustration of samples from the class with most (**a**) and the least (**b**) number of samples. In addition, classes with a higher (**c**) and smaller (**d**) average size. Images were rescaled to use the same aspect ratio.

6.2. Metrics

There are many different metrics and visualization techniques in the literature being used to evaluate the machine learning algorithm's performance on adversarial examples, each of them with a better use case or a more singular depiction of specific methods. In this work, we decided to use the following approaches:

- **Accuracy curves**: where each data point represents the accuracy (e.g., percentage of correctly classified samples over total number of samples) of the network in a given scenario—as described by Equation (5), where True Positive (*TP*), True Negative (*TN*), False Positive (*FP*) and False Negative (*FN*) are used.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5}$$

- **Confusion Matrices**: used to understand how similar the classes are before and after the injection of data, which might give a clue on the weights given to each class by the evaluated algorithms.

- **Precision–Recall curves**: as mentioned in Section 6.1, we are dealing with a highly imbalanced dataset, with some classes having an order of magnitude more examples than others. In those scenarios, precision–recall curves offer a better visualization on the true performance of the models, since it computes the capacity of the model in correctly classifying the target class when it has way fewer samples than the negative class. It is possible for a classifier to achieve high accuracies by learning to categorize based only on the major class if the positive to negative ratio is too low [58,59]. We also compute the average precision (AP) as described by Equation (6). It can be understood as the area under the precision–recall curve. These values are obtained by computing precision (P) and recall (R) over a range of thresholds (n), using the algorithm's output probabilities.

$$\mathrm{AP} = \sum_{n} (R_n - R_{n-1}) P_n \tag{6}$$

Another relevant factor that might impact the overall results is the data distribution. We randomly split the dataset into three parts: training (80%), validation (10%), and test (10%). We use the training and validation sets to perform the CNN training and combine them as a single gallery for the KNN search. This is important to evaluate the true generalization power of the classifier and reduce its chance of overfitting the data by simply replicating what it sees during the training phase [57]. No hyperparameter tuning is performed using the test set, simulating a real set of unseen examples. In that sense, we would be able to detect overfitting as a huge performance drop in the test set among different methods.

6.3. Injection Attacks with Random Data

To evaluate malware classification before and after code injection, we use the malimg [11] dataset. It has 9339 malware samples from 25 families. We randomly split the dataset into three parts: training (80%), validation (10%), and test (10%). We use the training and validation sets to perform the CNN training and combine them as a single gallery for the KNN search.

For testing, we insert m new sections with $n \times FileAlignment$ bytes at random parts of each test malware, with m and n varying from 1 to 5, totaling 25 different injection scenarios. We repeat training/testing experiments three times for each model and show the average results in Figure 5.

We can see that the way we inject multiple sections affects the results. For instance, despite the amount of data being the same, four sections with $FileAlignment$ bytes impact more the performance than two sections with $2 \times FileAlignment$ or one section with $4 \times FileAlignment$ bytes. Thus, dividing a portion of data into more parts and spreading them over the file is more effective in deceiving the classifier than having a few large sections.

The biggest drop occurred when we injected five sections with $5 \times FileAlignment$ bytes. As most samples have $FileAlignment = 512$ and the average malware size is 177 kB, our injection approach accounts for an approximate 7% increase in file size and misclassification rates ranging between 25% and 40%. Figure 6 illustrates the misclassification differences between the test set with original samples and a set with injected samples.

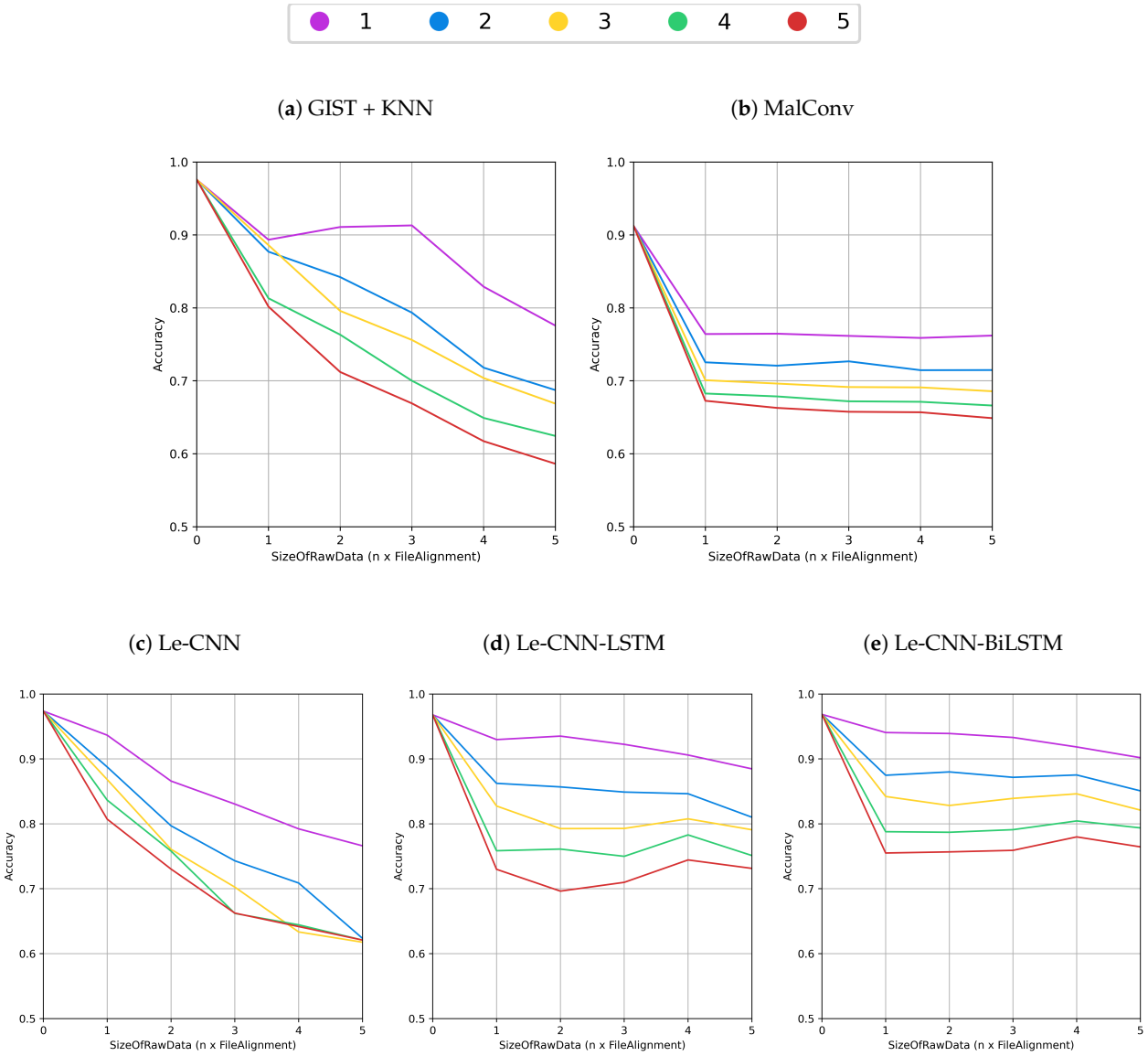
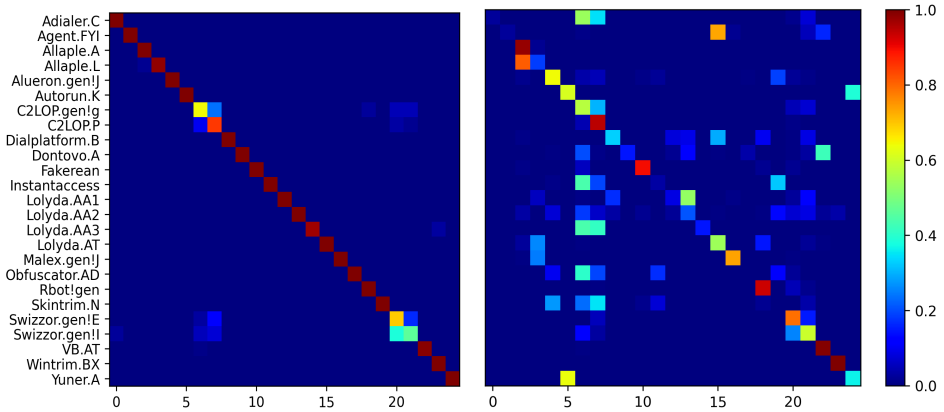


Figure 5. Average accuracy of malware classification under different injection scenarios. Different colors represent the number of injected sections.

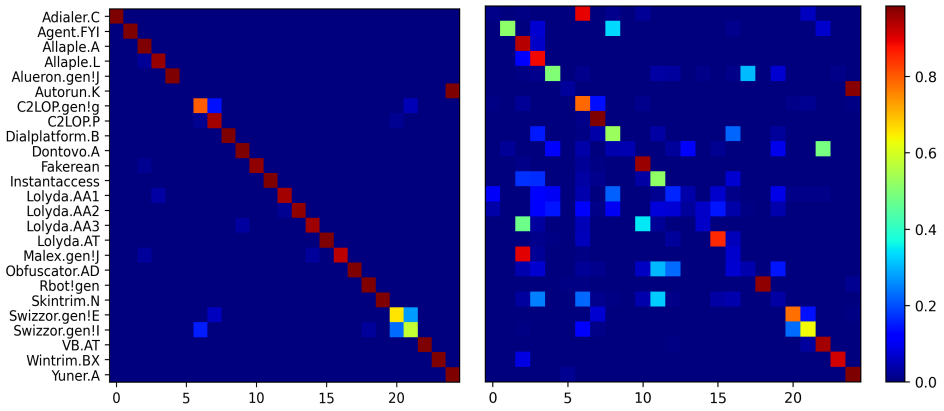
It is worth noting that some of these families share similar traits. For instance, families Autorun.K, Malex.gen!J, Rbot!gen, VB.AT and Yuner.A are all packed using UPX packer. Some families are variants of the same kind of malware, such as C2LOP.P and C2Lop.gen!g, Swizzor.gen!I and Swizzor.gen!E. It is expected that confusion concentrates around those variants [11].

We can see that all models fail to correctly classify these variants, even before data injection. The Le-CNN-BiLSTM model, as seen in Figure 6c,d does not learn how to correctly identify samples from a packed family, e.g., Autorun.K, incorrectly predicting them as Yuner.A. One behavior is clear in KNN and MalConv models: their tendencies to incorrectly predict samples as belonging to classes "Autorun.K", "C2LOP.gen!g" and "C2LOP.P". Those families share samples with high average sizes, at 524.54 kB, 386.92 kB and 524.04 kB, respectively. Since Le-CNN-BiLSTM resizes everything to 10 k bytes, this error is less prevalent with this model. In the same manner, MalConv has these classes as the ones with less misclassifications in the injected set. Considering its 1MB input, those are the samples where padding is used the least.

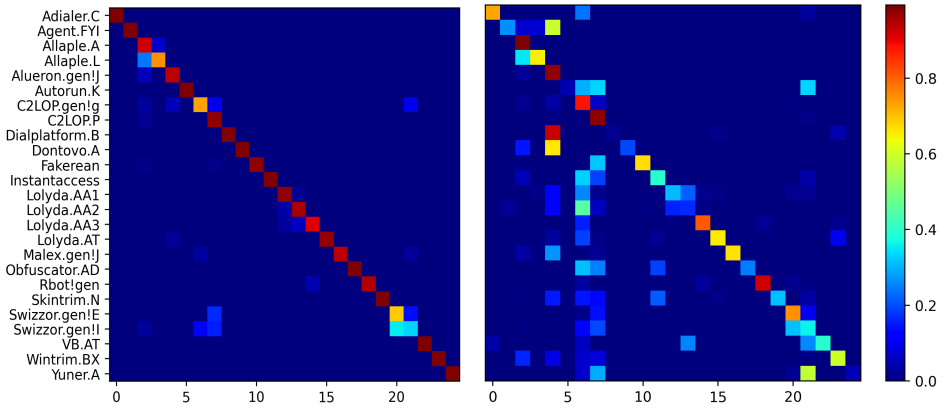
In Figure 7, we can see how the trained models lose precision after section injection. Due to the imbalanced nature of the dataset, this is illustrated by precision–recall curves.



(**a**) KNN+GIST—Original (**b**) KNN+GIST—Injected

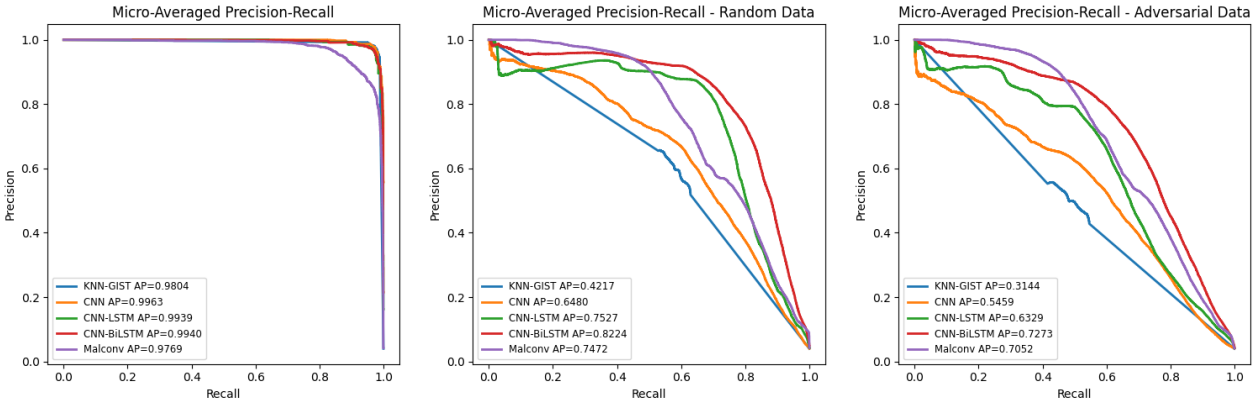


(**c**) CNN-BiLSTM—Original (**d**) CNN-BiLSTM—Injected



(**e**) MalConv—Original (**f**) MalConv—Injected

Figure 6. Confusion matrix for malware classification using KNN+GIST, Le-CNN-BiLSTM and MalConv in the original test set (**a**,**c**,**e**) and (**b**,**d**,**f**) when 5 sections of $5 \times FileAlignment$ bytes are injected.



(**a**) Original (**b**) Injected with random data (**c**) Injected with adversarial data

Figure 7. Precision–recall curves in the original test set (**a**) and when 5 sections of 5 × *FileAlignment* bytes are injected, (**b**) with random bytes and adversarial bytes in (**c**). Each color represents a different model.

We can see that the handcrafted method is the least precise in this scenario, being followed by Le-CNN, MalConv, Le-CNN-LSTM and Le-CNN-BiLSTM, respectively.

6.4. Injection Attacks with Adversarial Data

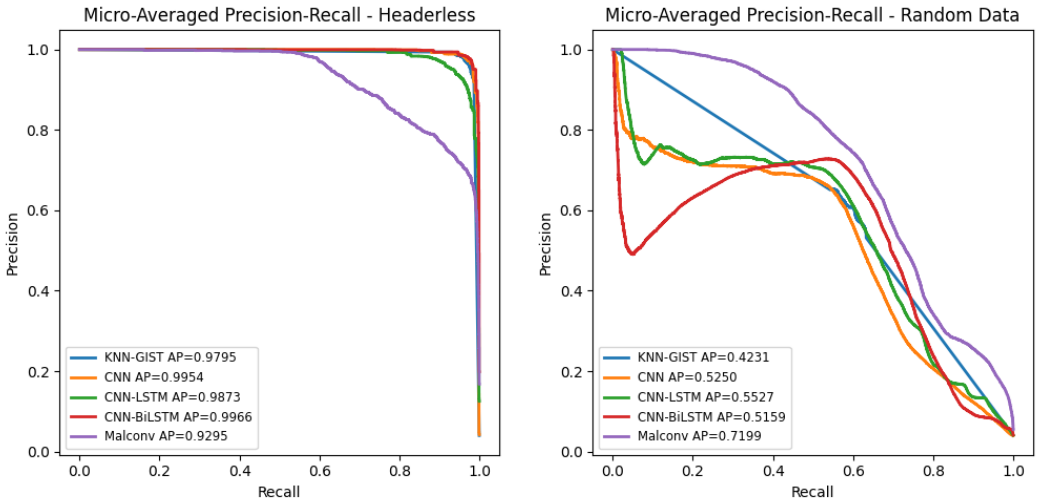
What if instead of adding random data we use bytes that appear in samples from other classes? We evaluate this kind of attack in this section, this time focusing on the most impactful injection scenario, i.e., 5 sections with 5 × *FileAlignment*. Figure 7c displays the difference that injecting with adversarial data imposes.

Comparing with the random injection results seen in Figure 7b, we can see that all models had their average precision decreased—KNN + GIST by 25.44%, Le-CNN by 15.75%, Le-CNN-LSTM by 15.91%, Le-CNN-BiLSTM by 11.56% and MalConv by 5.62%. That might be an indication that MalConv is learning more discriminative features from the samples, and it is deceived for reasons other than the kind of data being injected, since it becomes the model with the highest average precision despite losing more accuracy than Le-CNN-BiLSTM (Figure 5).

6.5. Evaluating Samples without Header

Here, we evaluate the possibility of training our models stripping the header of the samples, similarly to what is employed in BIG 2015 [31]. Figure 8 illustrates the results for samples without the header.

All models rely heavily on the samples header in order to perform classification, losing precision even before data injection as seen in Figure 8a. Only Le-CNN-BiLSTM increased its precision by 0.0026 in this scenario. All models became less robust to data injection, losing precision significantly when compared to complete executables in Figure 7b. Despite that, MalConv is the only model with similar average precision to previous scenarios.



(**a**) Original (**b**) Injected

Figure 8. Precision–recall curves in the original test set (**a**,**b**) when 5 sections of 5 × *FileAlignment* bytes are injected, both versions without file header.

6.6. Defending against Data Injection

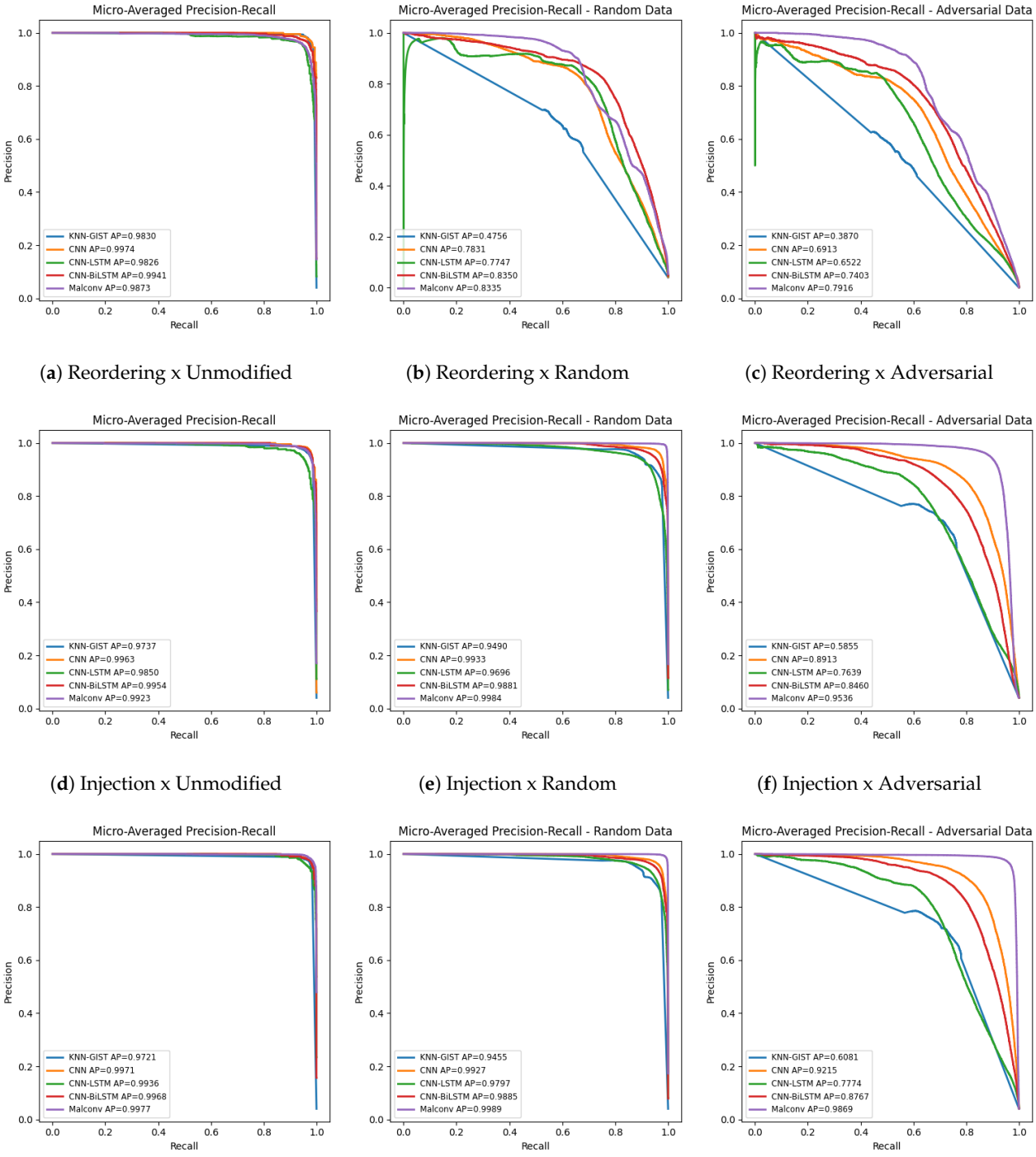
Multiple strategies were evaluated to make these models more robust against data injection by focusing on the data available during training.

6.6.1. Augmentation

A solution proposed in the literature [60–62] is to augment the data used for training. Three strategies were initially evaluated:

1. **Section reordering:** Since our injection scheme adds new sections in a random position among the existing one, the first augmentation idea was to reorder the sections on the training section. By doing this, we wanted to check if the model could be more robust against data injection without seeing them during training. As shown by Figure 9a–c, this strategy increased a bit the performance of all models when compared to the vanilla results shown by Figure 7.
2. **Training with injected data:** Since data are injected in the test set, a possibility was to include injected samples with random data in the training set as well. By comparing Figure 9d–f against Figure 7, we can see that all models became less vulnerable against random data injection but still struggle against adversarial data. MalConv benefitted the most in this scheme.
3. **Reorder+Injection:** Augmenting the training set with both injected and reordered samples, shown in Figure 9g–i, was also evaluated. Comparing with the original results in Figure 7, we can see that this may be a good defense strategy as well.

As shown by Figure 9, some models were improved by these augmentation strategies, even though they are still vulnerable.



(**a**) Reordering x Unmodified (**b**) Reordering x Random (**c**) Reordering x Adversarial

(**d**) Injection x Unmodified (**e**) Injection x Random (**f**) Injection x Adversarial

(**g**) Reordering + Injection x Unmodified (**h**) Reordering + Injection x Random (**i**) Reordering + Injection x Adversarial

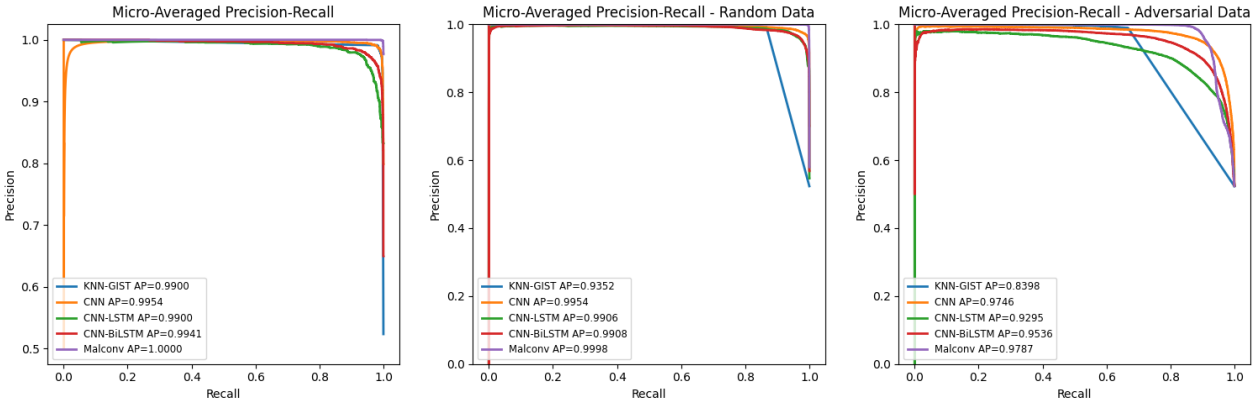
Figure 9. Precision–recall curves for tests with augmentation. (**a**,**d**,**g**) display results on test sets with original samples. (**b**,**e**,**h**) display results for datasets injected with random data. (**c**,**f**,**i**) display results for injection with adversarial data. Five sections of 5 × *FileAlignment* bytes are injected in all cases. Each color represents a different model.

6.6.2. Binary Data

All experiments mentioned here were also performed in a binary dataset. We collected samples from a clean Windows 10 Virtual Machine to form the "benign" class and kept every sample from the malimg [11] dataset as the "malware" class. For these tests, we

evaluated the model's performance on the original test set and against malware-only versions of the dataset injected with both random and adversarial data.

In this version of the dataset, the models were barely affected by data injection. We believe something similar to that mentioned by Raff et al. [16] also happened in our dataset: models were learning "Microsoft vs. non-Microsoft" instead of "Benign vs. Malign", as shown by Figure 10. The models did not really learn to differentiate between a malware and a benign file but rather whether a given executable is from Microsoft's library or not, hence why they seem more robust against injection attacks.



(**a**) Results on unmodified binary dataset

(**b**) Results on random injected malware

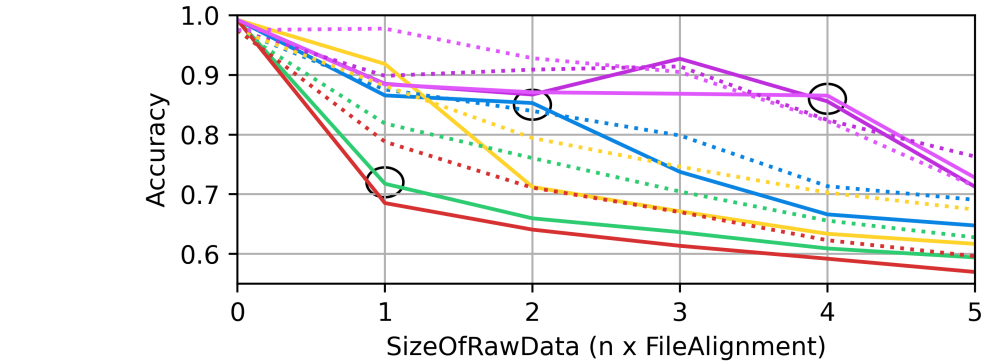
(**c**) Results on adversarial-injected malware

Figure 10. Precision–recall curves when 5 sections of 5 × *FileAlignment* bytes are injected, (**b**) with random bytes and adversarial bytes in (**c**). Each color represents a different model.

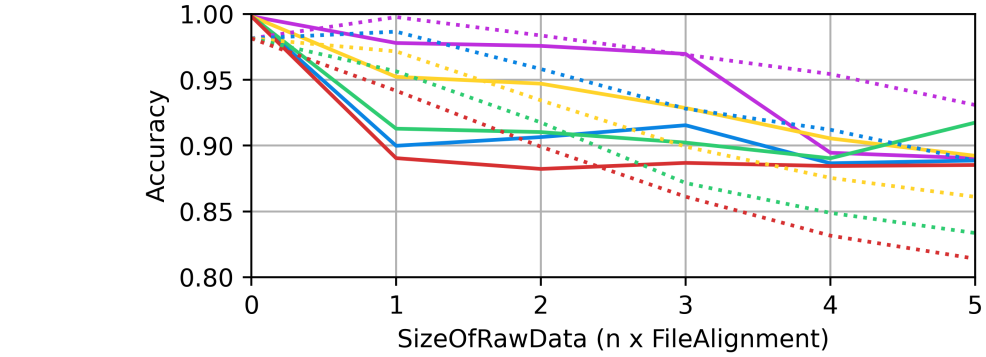
6.6.3. Scaling Models

Some of the challenges involved in building more robust models are closely related to the available data—highly imbalanced number of samples, sample size variation within and across families, the packing and obfuscation of samples—but those are not the only concerns. Increasing the architecture size does not necessarily lead to more robust models.

To verify that, we follow Chen's approach [9,15], which consists of fine-tuning a pre-trained CNN to classify malware families. We use the Inception-V3 architecture [44] pre-trained on the ImageNet dataset [43]. For that, we resize our images to 299 × 299 pixels and transform it into a 3-channel (RGB) image by replicating the grayscale channel. Then, we split our training into two phases. First, we recreate the last layer with the correct number of classes and optimize it while keeping the rest of the network frozen. We stop this training phase when the validation accuracy does not improve for ten epochs (patience). Then, we resume training for all layers with a 30-epoch patience. In both phases, we split training data into mini-batches of 64 images and use Adam optimizer for backpropagation with a learning rate of 10^{-4}. We can see that such a model is also vulnerable to section injection, as seen in a preliminary comparison against KNN+GIST, as illustrated in Figure 11.



(**a**) Malware classification (malware family)



(**b**) Malware detection (benign vs. malign)

Figure 11. Average accuracy of (**a**) malware classification and (**b**) malware detection under different injection scenarios. Solid lines show results for Inception architecture, and dashed lines show results for GIST + KNN. Distinct colors represent the number of injected sections.

A straightforward observation in Figure inception-resultsa is that this model presents the same behavior as the previous one in Section 6.3: the classification error increases with the amount of injected bytes, and spreading the injected data is more effective in deceiving the classifier than larger sections. For instance, despite the amount of data being the same, four sections with *FileAlignment* bytes impact more the performance than two sections with $2 \times$ *FileAlignment* or one section with $4 \times$ *FileAlignment* bytes (see the circles in Figure inception-resultsa). The location does not seem to matter when adding a single section, though, as the results for random place injection are equivalent to always inserting the section at the end of the file.

In Figure inception-resultsb, we can also observe the same behavior described for the other malware detectors in Section 6.6.2. Neither the volume of injected data nor its dispersion through files considerably affects the CNN performance, which is more robust than GIST + KNN in this experiment. Nonetheless, both CNN and GIST + KNN lost at least 10% accuracy in the worst case, which is not an acceptable margin for a protection measure. Our experiments highlight how risky it is to rely on image-based methods for malware detection and classification by showing how easily one can trick them.

Current results point in the direction of combining text processing techniques with convolutional layers, as made by MalConv [16] with its embedding layer and Le-CNN-BiLSTM [17] with the recurrent layer after convolutional ones. An open challenge regarding these approaches is related to their input sizes: MalConv truncates data larger than a given size, which requires choosing between discarding relevant data and using more

computational resources to process larger samples; CNN-BiLSTM interpolates its input to a fixed size, possibly removing relevant byte relationships in some regions of the file.

7. Conclusions

In this work, we proposed a new method to inject data into malware files to change its classification when analyzed by an automatic malware classification system. With a mere 7% file size increase, we dropped the accuracy of five classifiers on par with the state-of-the-art—namely GIST + KNN [11], MalConv [16] and Le-CNN [17] and two other variations—between 25% and 40%. The obtained results seem promising, and we think this method can be improved to be robust enough for a larger scale of scenarios. There are some points researchers using this method need to be aware of:

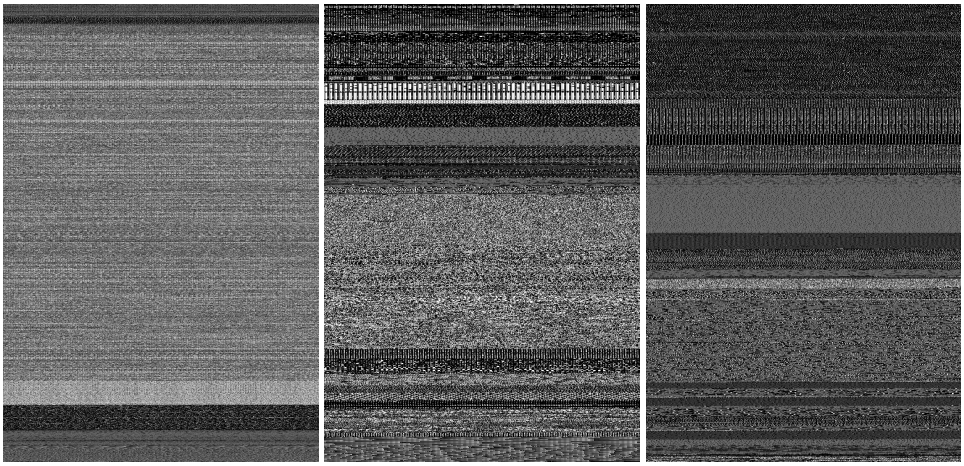
- The usage of CNNs is gaining momentum in this research field literatures [9,14,15,17–19]. This work shows a simple technique that can make the accuracy in such CNNs drop in almost 50% by adding small perturbations to a malware file. We could observe that methods such as Gated CNN [16] or combining CNN with LSTM [17] can be more robust against the data injection presented here.
- A deeper understanding of how the operating system loads executable files to memory usually helps malware creators. During preliminary tests, we saw that some file format rules are flexible, and malware authors do not follow all of them. It includes files with section headers missing or some sections not aligned to the required flags. We tried our best to keep our generated examples in accordance with the format specified. Malware creators might not have this mentality, so that should be considered when building neural networks with the purpose of detecting malware files that rely on static features from the file.
- Our results show that data dispersion might be just as important as the amount of data being injected. We can use this idea to conduct a more directed attack using our method together with the method proposed by Khormali et al. (2019) [19], injecting FSGM-generated sections in any position of the file.

Challenges and Future Directions

As mentioned in Section 6.6, augmenting the training set with injected samples might not be enough to prevent section injection attacks nor only increasing architecture size. Further investigation is required on how to transform the input for the models in such a way that only relevant data for the classification are kept. Current experiments point in the direction that instead of relying on a fixed preprocessing method—such as truncating or interpolating—more dynamic approaches should be investigated, such as Attention-based methods.

Another direction worthy of new experiments is the *interpretability* of these results. We believe that the highly structured pattern in some samples is what makes them more discriminative, hence why injecting patterns from opposite classes is hard to defend against in all scenarios. The same goes for binary classification, where the models were seemingly more robust against adversarial samples. The dataset construction needs to be explored with more diversity to avoid introducing any kind of bias during sample selection, which is a possible issue with our version. Figure 12 illustrates the variability of patterns (or lack thereof) that might be the main discriminative feature for the studied models.

(**a**) Benign samples



(**b**) Malware samples

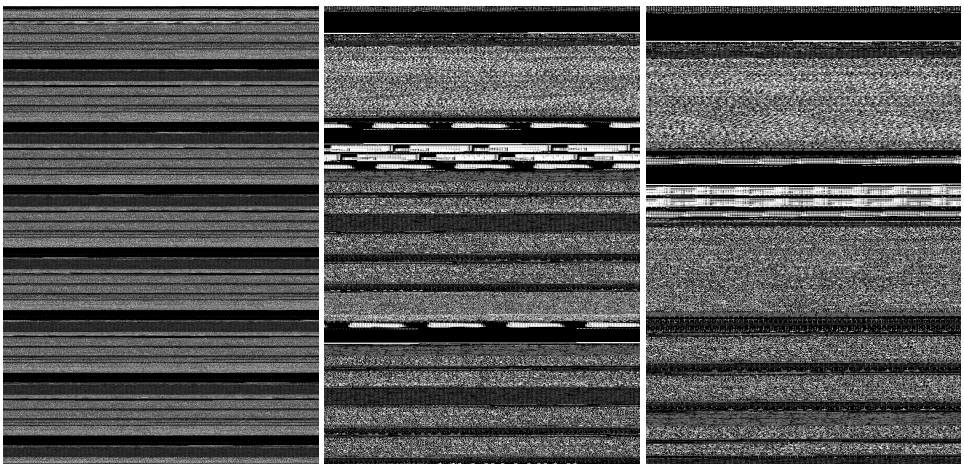


Figure 12. Illustration of the patterns variations exhibited in benign (**a**) and in malware (**b**) samples. Some patterns are highly structured and appear multiple times within the file. Some others are relatively small or appear only a few times in the sample's body.

Author Contributions: Conceptualization, A.A.d.S. and M.P.S.; methodology, A.A.d.S.; software, A.A.d.S.; validation, A.A.d.S. and M.P.S.; formal analysis, M.P.S.; investigation, A.A.d.S. and M.P.S.; resources, A.A.d.S.; data curation, A.A.d.S.; writing—original draft preparation, A.A.d.S.; writing—review and editing, M.P.S.; visualization, A.A.d.S. and M.P.S.; supervision, M.P.S.; project administration, A.A.d.S. and M.P.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The authors code is available at https://github.com/adeilsonsilva/malware-injection (accessed on 10 January 2023). Restrictions apply to the availability of the used datasets and related code [52–54].

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AOCD	Architecture Object Code Dataset
BiLSTM	Bidirectional Long Short-Term Memory
CNN	Convolutional Neural Networks
CW	Carlini–Wagner
FN	False Negaive
FP	False Positive
FGSM	Fast Gradient Sign Method
GAMMA	Genetic Adversarial Machine learning Malware Attack
GBDT	Gradient Boost Decision Tree
GIST	Global Image Descriptor
IPR	In-Place Randomization
KNN	K-Nearest Neighbors
LSTM	Long Short-Term Memory
PE	Portable Executable
SGD	Stochastic Gradient Descent
TN	True Negative
TP	True Positive

References

1. Sikorski, M.; Honig, A. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st ed.; No Starch Press: San Francisco, CA, USA, 2012.
2. Aboaoja, F.A.; Zainal, A.; Ghaleb, F.A.; Al-rimy, B.A.S.; Eisa, T.A.E.; Elnour, A.A.H. Malware Detection Issues, Challenges, and Future Directions: A Survey. *Appl. Sci.* **2022**, *12*, 8482. [CrossRef]
3. Naseer, M.; Rusdi, J.F.; Shanono, N.M.; Salam, S.; Muslim, Z.B.; Abu, N.A.; Abadi, I. Malware detection: Issues and challenges. In *Journal of Physics: Conference Series*; IOP Publishing: Bristol, UK, 2021; Volume 1807, p. 012011.
4. Alenezi, M.N.; Alabdulrazzaq, H.; Alshaher, A.A.; Alkharang, M.M. Evolution of malware threats and techniques: A review. *Int. J. Commun. Netw. Inf. Secur.* **2020**, *12*, 326–337. [CrossRef]
5. Li, Y.; Caragea, D.; Hall, L.; Ou, X. Experimental Study of Machine Learning based Malware Detection Systems' Practical Utility. In *Hicss Symposium On Cybersecurity Big Data Analytics*; 2020. Available online: https://par.nsf.gov/biblio/10178634-experimental-study-machine-learning-based-malware-detection-systems-practical-utility (accessed on 10 January 2023).
6. Microsoft Corporation. *Microsoft Security Intelligence Report Volume 24*; Technical Report; Microsoft Corporation: Redmond, WA, USA, 2019. Available online: https://www.microsoft.com/security/blog/2019/02/28/microsoft-security-intelligence-report-volume-24-is-now-available/ (accessed on 11 October 2022).
7. Symantec Corporation. *Internet Security Threat Report Volume 24*; Technical Report; Symantec Corporation: Tempe, AZ, USA, 2019. Available online: https://docs.broadcom.com/doc/istr-24-2019-en (accessed on 11 October 2022).
8. Microsoft 365 Defender Threat Intelligence Team. Microsoft Researchers Work with Intel Labs to Explore New Deep Learning Approaches for Malware Classification. 2020. Available online: https://www.microsoft.com/security/blog/2020/05/08/microsoft-researchers-work-with-intel-labs-to-explore-new-deep-learning-approaches-for-malware-classification/ (accessed on 19 February 2022).
9. Chen, L.; Sahita, R.; Parikh, J.; Marino, M. STAMINA: Scalable deep learning approach for malware classification. *Intel White Paper* **2020**, *1*, 3.
10. Anderson, H.S.; Kharkar, A.; Filar, B.; Roth, P. Evading machine learning malware detection. *Black Hat* **2017**, *2017*. Available online: https://www.blackhat.com/docs/us-17/thursday/us-17-Anderson-Bot-Vs-Bot-Evading-Machine-Learning-Malware-Detection-wp.pdf (accessed on 10 January 2023).
11. Nataraj, L.; Karthikeyan, S.; Jacob, G.; Manjunath, B.S. Malware Images: Visualization and Automatic Classification. In Proceedings of the 8th International Symposium on Visualization for Cyber Security, Pittsburgh, PA, USA, 20 July 2011; ACM: New York, NY, USA, 2011; VizSec '11; pp. 4:1–4:7. [CrossRef]
12. Grosse, K.; Papernot, N.; Manoharan, P.; Backes, M.; McDaniel, P. Adversarial perturbations against deep neural networks for malware classification. *arXiv* **2016**, arXiv:1606.04435.
13. Athiwaratkun, B.; Stokes, J.W. Malware classification with LSTM and GRU language models and a character-level CNN. In Proceedings of the 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), New Orleans, LA, USA, 5–9 March 2017; pp. 2482–2486.
14. Yue, S. Imbalanced malware images classification: A CNN based approach. *arXiv* **2017**, arXiv:1708.08042.
15. Chen, L. Deep transfer learning for static malware classification. *arXiv* **2018**, arXiv:1812.07606.
16. Raff, E.; Barker, J.; Sylvester, J.; Brandon, R.; Catanzaro, B.; Nicholas, C. Malware detection by eating a whole exe. *arXiv* **2017**, arXiv:1710.09435.

17. Le, Q.; Boydell, O.; Namee, B.M.; Scanlon, M. Deep learning at the shallow end: Malware classification for non-domain experts. *Digit. Investig.* **2018**, *26*, S118–S126. [CrossRef]
18. Su, J.; Vasconcellos, V.D.; Prasad, S.; Daniele, S.; Feng, Y.; Sakurai, K. Lightweight Classification of IoT Malware Based on Image Recognition. In Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, 23–27 July 2018; Volume 2; pp. 664–669. [CrossRef]
19. Khormali, A.; Abusnaina, A.; Chen, S.; Nyang, D.; Mohaisen, A. COPYCAT: Practical adversarial attacks on visualization-based malware detection. *arXiv* **2019**, arXiv:1909.09735.
20. Benkraouda, H.; Qian, J.; Tran, H.Q.; Kaplan, B. Attacks on Visualization-Based Malware Detection: Balancing Effectiveness and Executability. In *International Workshop on Deployable Machine Learning for Security Defense*; Springer: Cham, Switerlands, 2021; pp. 107–131.
21. Goodfellow, I.J.; Shlens, J.; Szegedy, C. Explaining and Harnessing Adversarial Examples. In Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, 7–9 May 2015.
22. Al-Dujaili, A.; Huang, A.; Hemberg, E.; O'Reilly, U. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. In Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA, 24 May 2018; pp. 76–82. [CrossRef]
23. Demetrio, L.; Biggio, B.; Lagorio, G.; Roli, F.; Armando, A. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv* **2019**, arXiv:1901.03583.
24. Demetrio, L.; Coull, S.E.; Biggio, B.; Lagorio, G.; Armando, A.; Roli, F. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Trans. Priv. Secur. (TOPS)* **2021**, *24*, 1–31. [CrossRef]
25. Lucas, K.; Sharif, M.; Bauer, L.; Reiter, M.K.; Shintre, S. Malware Makeover: Breaking ML-based static analysis by modifying executable bytes. In Proceedings of the Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security, Virtual Event, 7–11 June 2021; pp. 744–758.
26. Madry, A.; Makelov, A.; Schmidt, L.; Tsipras, D.; Vladu, A. Towards deep learning models resistant to adversarial attacks. *arXiv* **2017**, arXiv:1706.06083.
27. Shafahi, A.; Najibi, M.; Ghiasi, M.A.; Xu, Z.; Dickerson, J.; Studer, C.; Davis, L.S.; Taylor, G.; Goldstein, T. Adversarial training for free! In Proceedings of the Advances in Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019; Volume 32.
28. Zhang, J.; Dong, Y.; Liu, B.; Ouyang, B.; Zhu, J.; Kuang, M.; Wang, H.; Meng, Y. The art of defense: Letting networks fool the attacker. *arXiv* **2021**, arXiv:2104.02963.
29. Ho, C.H.; Vasconcelos, N. DISCO: Adversarial Defense with Local Implicit Functions. *arXiv* **2022**, arXiv:2212.05630.
30. Yoo, K.; Kim, J.; Jang, J.; Kwak, N. Detection of Word Adversarial Examples in Text Classification: Benchmark and Baseline via Robust Density Estimation. *arXiv* **2022**, arXiv:2203.01677.
31. Ronen, R.; Radu, M.; Feuerstein, C.; Yom-Tov, E.; Ahmadi, M. Microsoft Malware Classification Challenge. 2018. Available online: http://xxx.lanl.gov/abs/1802.10135 (accessed on 10 January 2023).
32. Agarap, A.F.; Pepito, F.J.H. Towards Building an Intelligent Anti-Malware System: A Deep Learning Approach using Support Vector Machine (SVM) for Malware Classification. *arXiv* **2018**, arXiv:1801.00318.
33. Liu, Y.s.; Lai, Y.K.; Wang, Z.H.; Yan, H.B. A New Learning Approach to Malware Classification Using Discriminative Feature Extraction. *IEEE Access* **2019**, *7*, 13015–13023. [CrossRef]
34. Pascanu, R.; Stokes, J.W.; Sanossian, H.; Marinescu, M.; Thomas, A. Malware classification with recurrent networks. In Proceedings of the 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), South Brisbane, QLD, Australia, 19–24 April 2015; pp. 1916–1920. [CrossRef]
35. Saxe, J.; Berlin, K. Deep neural network based malware detection using two dimensional binary program features. In Proceedings of the 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), Fajardo, PR, USA, 20–22 October 2015; pp. 11–20. [CrossRef]
36. Anderson, H.S.; Roth, P. Ember: An open dataset for training static pe malware machine learning models. *arXiv* **2018**, arXiv:1804.04637.
37. İbrahim, M.; Issa, B.; Jasser, M.B. A Method for Automatic Android Malware Detection Based on Static Analysis and Deep Learning. *IEEE Access* **2022**, *10*, 117334–117352. [CrossRef]
38. Gao, Y.; Hasegawa, H.; Yamaguchi, Y.; Shimada, H. Malware Detection by Control-Flow Graph Level Representation Learning With Graph Isomorphism Network. *IEEE Access* **2022**, *10*, 111830–111841. [CrossRef]
39. Raff, E.; Zak, R.; Cox, R.; Sylvester, J.; Yacci, P.; Ward, R.; Tracy, A.; McLean, M.; Nicholas, C. An investigation of byte n-gram features for malware classification. *J. Comput. Virol. Hacking Tech.* **2018**, *14*, 1–20. [CrossRef]
40. HaddadPajouh, H.; Dehghantanha, A.; Khayami, R.; Choo, K.K.R. A deep Recurrent Neural Network based approach for Internet of Things malware threat hunting. *Future Gener. Comput. Syst.* **2018**, *85*, 88–96. [CrossRef]
41. Vinayakumar, R.; Alazab, M.; Soman, K.; Poornachandran, P.; Venkatraman, S. Robust intelligent malware detection using deep learning. *IEEE Access* **2019**, *7*, 46717–46738. [CrossRef]
42. Uysal, D.T.; Yoo, P.D.; Taha, K. Data-driven malware detection for 6G networks: A survey from the perspective of continuous learning and explainability via visualisation. *IEEE Open J. Veh. Technol.* **2022**, *4*, 61–71. [CrossRef]

43. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009.
44. Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 2818–2826.
45. Pappas, V.; Polychronakis, M.; Keromytis, A.D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In Proceedings of the IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012.
46. Koo, H.; Polychronakis, M. Juggling the gadgets: Binary-level code randomization using instruction displacement. In Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS), Xi'an China, 30 May 2016–3 June 2016.
47. Carlini, N.; Wagner, D. Towards Evaluating the Robustness of Neural Networks. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 39–57. [CrossRef]
48. Coull, S.E.; Gardner, C. Activation analysis of a byte-based deep neural network for malware classification. In Proceedings of the 2019 IEEE Security and Privacy Workshops (SPW). IEEE, San Francisco, CA, USA, 19–23 May 2019; pp. 21–27.
49. Clemens, J. Automatic classification of object code using machine learning. *Digit. Investig.* **2015**, *14*, S156–S162. [CrossRef]
50. Krčál, M.; Švec, O.; Bálek, M.; Jašek, O. Deep convolutional malware classifiers can learn from raw executables and labels only. In Proceedings of the 6th International Conference on Learning Representations (ICLR 2018), Vancouver, BC, Canada, 30 April–3 May 2018. Available online: https://openreview.net/pdf?id=HkHrmM1PM (accessed on 10 January 2023).
51. Kolosnjaji, B.; Zarras, A.; Webster, G.; Eckert, C. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*; Springer: Cham, Switerlands, 2016; pp. 137–149.
52. Sarvam. Supervised Classification with k-fold Cross Validation on a Multi Family Malware Dataset. 2014. Available online: https://sarvamblog.blogspot.com/2014/08/supervised-classification-with-k-fold.html (accessed on 10 January 2023).
53. CeADAR Ireland. Deep Learning at the Shallow End: Malware Classification for Non-Domain Experts. 2018. Available online: https://bitbucket.org/ceadarireland/deeplearningattheshallowend/src/master (accessed on 10 January 2023).
54. Elastic. Elastic Malware Benchmark for Empowering Researchers. 2017. Available online: https://github.com/elastic/ember (accessed on 10 January 2023).
55. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. *arXiv* 2014, arXiv:1412.6980.
56. Sutskever, I.; Martens, J.; Dahl, G.; Hinton, G. On the importance of initialization and momentum in deep learning. In Proceedings of the 30th International Conference on Machine Learning, PMLR, Atlanta, GA, USA, 17–19 June 2013; Volume 28, pp. 1139–1147.
57. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016. Available online: http://www.deeplearningbook.org (accessed on 10 January 2023).
58. Davis, J.; Goadrich, M. The relationship between Precision-Recall and ROC curves. In Proceedings Of The 23rd International Conference On Machine Learning, Pittsburgh, PA, USA, 25–29 June 2006; pp. 233–240.
59. Saito, M. The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. *PLOS ONE* **2015**, *10*, e0118432. [CrossRef]
60. Catak, F.O.; Ahmed, J.; Sahinbas, K.; Khand, Z.H. Data augmentation based malware detection using convolutional neural networks. *PeerJ Comput. Sci.* **2021**, *7*, e346. [CrossRef]
61. Perez, L.; Wang, J. The effectiveness of data augmentation in image classification using deep learning. *arXiv* **2017**, arXiv:1712.04621.
62. Taylor, L.; Nitschke, G. Improving deep learning with generic data augmentation. In Proceedings of the 2018 IEEE Symposium Series on Computational Intelligence (SSCI), Bangalore, India, 18–21 November 2018; pp. 1542–1547.