

PGCOMP - Programa de Pós-Graduação em Ciência da Computação  
Universidade Federal da Bahia (UFBA)  
Av. Adhemar de Barros, s/n - Ondina  
Salvador, BA, Brasil, 40170-110

<http://pgcomp.dcc.ufba.br>  
[pgcomp@ufba.br](mailto:pgcomp@ufba.br)

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 GIST+KNN, three CNN variations and one Gated CNN. We performed our experiments on a public dataset with 9,339 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 malwares alongside the original ones to increase networks robustness against mentioned attacks. Results show that a combination of reordering malware sections and injecting random data can improve overall performance of the classification. Code available at (<https://github.com/adeilsonsilva/malware-injection>).

# On deceiving malware classification with section injection: attack and defense using deep neural networks

Adeilson Antonio da Silva

Dissertação de Mestrado

Universidade Federal da Bahia

Programa de Pós-Graduação em  
Ciência da Computação

Novembro | 2022

MSC | 147 | 2022

On deceiving malware classification with section injection: attack and defense using deep neural networks

Adeilson Antonio da Silva

UFBA





Universidade Federal da Bahia  
Instituto de Computação

Programa de Pós-Graduação em Ciência da Computação

**ON DECEIVING MALWARE  
CLASSIFICATION WITH SECTION  
INJECTION: ATTACK AND DEFENSE  
USING DEEP NEURAL NETWORKS**

Adeilson Antonio da Silva

DISSERTAÇÃO DE MESTRADO

Salvador  
22 de Novembro de 2022



ADEILSON ANTONIO DA SILVA

**ON DECEIVING MALWARE CLASSIFICATION WITH SECTION  
INJECTION: ATTACK AND DEFENSE USING DEEP NEURAL  
NETWORKS**

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Maurício Pamplona Segundo

Salvador  
22 de Novembro de 2022

Sistema de Bibliotecas - UFBA

Silva, Adeilson Antonio da.

On deceiving malware classification with section injection: attack and defense using deep neural networks / Adeilson Antonio da Silva – Salvador, 2022.

47p.: il.

Orientador: Prof. Dr. Prof. Dr. Maurício Pamplona Segundo.

Dissertação (Mestrado) – Universidade Federal da Bahia, Instituto de Computação, 2022.

1. Redes Neurais. 2. Análise de Malware. 3. Aprendizado de Máquina.  
I. Segundo, Maurício Pamplona. II. Universidade Federal da Bahia. Instituto de Computação. III Título.

CDD – XXX.XX

CDU – XXX.XX.XXX

**MINISTÉRIO DA EDUCAÇÃO**  
**UNIVERSIDADE FEDERAL DA BAHIA**  
**INSTITUTO DE COMPUTAÇÃO**  
**PGCOMP - Programa de Pós-Graduação em Ciência da Computação**  
<http://pgcomp.ufba.br>


---

*“On deceiving malware classification with section injection: attack and defense using deep neural networks”*

Adeilson Antônio da Silva

Dissertação apresentada ao  
Colegiado do Programa de Pós-Graduação em Ciência  
da Computação na Universidade Federal da Bahia,  
como requisito parcial para obtenção do Título de  
Mestre em Ciência da Computação.

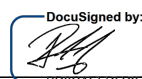
**Banca Examinadora**

DocuSigned by:  


FB3AA83C25B9414...

---

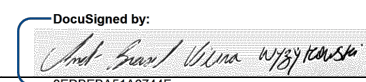
Prof. Dr. Maurício Pamplona Segundo (Orientador PGCOMP)

DocuSigned by:  


BD8CAEFD5D97451...

---

Prof. Dr. Karl Apaza Agüero (PGCOMP)

DocuSigned by:  


8EDBEBA51A6744F...

---

Prof. Dr. André Brasil Vieira Wyzykowski (Michigan State  
University)



*A todos aqueles de pele preta, arrancados de suas terras,  
sem direito a indenização.*





## **AGRADECIMENTOS**

Agradeço a todos que tornaram esse sonho possível: minha mãe, Mary Márcia, e meu pai, Adilson Silva, a quem tudo devo sem ter como retribuir. Meu melhor amigo, Bidu, que esteve sempre ao meu lado com sua alegria e seu amor fraterno. Minha melhor amiga, Caroline Ferraz, que emprestou seus ouvidos para aturar meus lamentos e sempre me deu forças para não desistir. Meu orientador, Maurício Pamplona Segundo, que confiou em mim e nas minhas ideias, dedicando seu tempo e seu conhecimento para me auxiliar nesse processo.

Desde quando comecei a sonhar em trabalhar com pesquisa tinha noção das dificuldades que enfrentaria num país com baixo investimento em ciência. Não imaginei, entretanto, ter de enfrentar uma pandemia que tornou todo processo ainda mais longo e complicado. Resolvi enfrentar essas adversidades junto aos meus e pelos meus. E graças a eles eu consegui chegar até aqui.



*A ciência, na minha opinião, é uma ferramenta absolutamente essencial para qualquer sociedade que tenha a esperança de sobreviver bem no próximo século com seus valores fundamentais intactos - não apenas como é praticada pelos seus profissionais, mas a ciência compreendida e adotada por toda a comunidade humana. E se os cientistas não realizarem essa tarefa, quem o fará?*

—CARL SAGAN (O Mundo assombrado pelos demônios)



## RESUMO

Neste trabalho investigamos como modificar arquivos executáveis de software com o intuito de enganar sistemas automatizados de classificação de malwares. A principal contribuição deste trabalho consiste em uma metodologia para injetar bytes em um arquivo aleatoriamente e utilizar isso como ataque para reduzir a acurácia da classificação, mas também como um método de defesa, aumentando a quantidade de dados disponíveis durante o treino desses sistemas. A injeção mencionada respeita o formato de arquivos do sistema operacional, de forma a garantir que o malware ainda será executável depois das modificações e não terá seu comportamento modificado. Nós reproduzimos cinco abordagens diferentes do estado da arte para classificação de malwares e avaliamos nosso esquema de injeção de dados: um baseado em GIST+KNN, três variações de CNN e uma Gated CNN. Nossos experimentos foram feitos utilizando um dataset disponível publicamente com 9339 exemplares de malware de 25 famílias diferentes. Nossos resultados mostram que um simples aumento de 7% no tamanho do malware pode causar uma diminuição entre 25% e 40% na classificação de famílias. Eles mostram também que um sistema automatizado de classificação pode não ser tão confiável quanto inicialmente reportado na literatura. Nós avaliamos também a utilização de malwares modificados em conjunto aos originais para aumentar a robustez da rede contra os ataques mencionados. Os resultados apontam que uma combinação da reordenação das seções dos malwares com a injeção de dados pode resultar em uma melhora no desempenho da classificação. Os códigos utilizados estão disponíveis em <https://github.com/adeilsonsilva/malware-injection>.



## 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 GIST+KNN, three CNN variations and one Gated CNN. We performed our experiments on a public dataset with 9,339 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 malwares alongside the original ones to increase networks robustness against mentioned attacks. Results show that a combination of reordering malware sections and injecting random data can improve overall performance of the classification. Code available at <https://github.com/adeilsonsilva/malware-injection>.





# CONTENTS

<b>Chapter 1—Introduction</b>	1
<b>Chapter 2—Background</b>	5
2.1 Malware Analysis . . . . .	5
2.2 File format . . . . .	6
2.3 Malware feature extraction . . . . .	7
<b>Chapter 3—Related Works</b>	9
3.1 Malware classification . . . . .	9
3.2 Malware injection . . . . .	11
<b>Chapter 4—Methodology</b>	15
4.1 Data injection . . . . .	16
4.1.1 File header . . . . .	16
4.1.2 Section header . . . . .	16
4.1.3 Injected data . . . . .	17
4.1.4 Workarounds . . . . .	18
4.2 Malware classification . . . . .	18
4.2.1 Image generation . . . . .	19
4.2.2 GIST+K-Nearest Neighbours (KNN) . . . . .	19
4.2.3 Convolutional Neural Networks (CNN) and Long Short-Term Mem- ory (LSTM) . . . . .	20
4.2.4 Gated CNN . . . . .	25
<b>Chapter 5—Results</b>	27
5.1 Dataset . . . . .	27
5.2 Metrics . . . . .	29
5.3 Injection Attacks with Random Data . . . . .	30
5.4 Injection Attacks with Adversarial Data . . . . .	32
5.5 Evaluating Samples without Header . . . . .	32
5.6 Defending Against Data Injection . . . . .	35
5.6.1 Augmentation . . . . .	35
5.6.2 Binary Data . . . . .	35
5.6.3 Scaling models . . . . .	37

**Chapter 6—Conclusions and Future Work**

41

## LIST OF FIGURES

2.1	Illustration of the sections of a PE32 executable file. . . . .	6
3.1	Illustration of the differences among attacks to image-based malware classifiers. The leftmost (red) square display our approach. Blue squares displays previous attacks. . . . .	11
4.1	Flowchart of an image-based malware classification system (blue lines). Red lines replace the dashed blue line in our data injection scheme. . . .	15
4.2	Image description using GIST feature extractor. . . . .	20
4.3	Classifying samples using K-Nearest Neighbors algorithm. . . . .	21
4.4	Illustration of the data flow in Le <i>et al</i> (LE et al., 2018) architectures. Transforming malware data to a fixed length. Feature extraction and dimensionality reduction through 1D CNN and 1D pooling layers. Since the input is scaled to a fixed size of 10K bytes, after these operations we have a matrix of size $78 \times 90$ . We can either unroll to a 1D feature vector and classify it using fully connected layers or consider it as smaller subsequences and use recurrent layers. . . . .	21
4.5	An example of a single a Long Short-Term Memory cell. . . . .	22
4.6	Operations performed at the “forget gate”. . . . .	23
4.7	Operations performed at the “input gate”. . . . .	24
4.8	Operations performed at the “output gate”. . . . .	24
4.9	Illustration of the embedding vector learning 4.9a and the transformation it goes through in the gating phase 4.9b. . . . .	25
5.1	Illustration of samples from the class with most (5.1a) and the least (5.1b) number of samples. Also, class with higher (5.1c) and with smaller (5.1d) average size. Images were rescaled to use the same aspect ratio. . . . .	29
5.2	Average accuracy of malware classification under different injection scenarios. Distinct colors represent the amount of injected sections. . . . .	31
5.3	Confusion matrices for malware classification using KNN+GIST, Le-CNN-BiLSTM and MalConv in the original test set 5.3a,5.3c,5.3e and 5.3b,5.3d,5.3f when 5 sections of $5 \times FileAlignment$ bytes are injected. . . . .	33
5.4	Precision Recall curves in the original test set 5.4a and when 5 sections of $5 \times FileAlignment$ bytes are injected, 5.4b with random bytes and adversarial bytes in 5.4c. Each color represents a different model. . . . .	34
5.5	Precision Recall curves in the original test set 5.5a and 5.5b when 5 sections of $5 \times FileAlignment$ bytes are injected, both versions without file header. . . . .	34

5.6	Precision Recall curves for tests with augmentation. 5.6a, 5.6d and 5.6g displays results on test sets with original samples. 5.6b, 5.6e and 5.6h displays results for datasets injected with random data. 5.6c, 5.6f and 5.6i display results for injection with adversarial data. 5 sections of $5 \times FileAlignment$ bytes are injected in all cases. Each color represents a different model. . . . .	36
5.7	Precision Recall curves when 5 sections of $5 \times FileAlignment$ bytes are injected in a binary dataset, 5.7b with random bytes and adversarial bytes in 5.7c. Each color represents a different model. . . . .	37
5.8	Average accuracy of 5.8a malware classification and 5.8b malware detection under different injection scenarios. Solid lines show results for Inception architecture, and dashed lines for GIST+KNN. Distinct colors represent the number of injected sections. . . . .	38

## LIST OF TABLES

3.1	Summary of different malware classification techniques. . . . .	10
3.2	Summary of functionality preserving attacks against PE32 malware classification. . . . .	14
4.1	Flags in the header of PE32 files. . . . .	16
4.2	Flags in section headers of PE32 files. . . . .	17
4.3	Image width based on the executable size (NATARAJ et al., 2011). . . . .	19
5.1	Samples distribution and average size in maling dataset. . . . .	28



## LIST OF ACRONYMS

<b>AOCD</b>	Architecture Object Code Dataset . . . . .	13
<b>CNN</b>	Convolutional Neural Networks . . . . .	xv
<b>CW</b>	Carlini-Wagner . . . . .	12
<b>FGSM</b>	Fast Gradient Sign Method . . . . .	11
<b>FN</b>	False Negative . . . . .	30
<b>FP</b>	False Positive . . . . .	30
<b>GAMMA</b>	Genetic Adversarial Machine learning Malware Attack . . . . .	12
<b>GBDT</b>	Gradient Boost Decision Tree . . . . .	12
<b>IPR</b>	In-place Randomization . . . . .	12
<b>KNN</b>	K-Nearest Neighbours . . . . .	xv
<b>LSTM</b>	Long Short-Term Memory . . . . .	xv
<b>PE</b>	Portable Executable . . . . .	3
<b>TP</b>	True Positive . . . . .	29
<b>TN</b>	True Negative . . . . .	29





## Chapter

# 1

## INTRODUCTION

What is a Malware? Sikorski & Honig (2012) use this term to describe pieces of software accordingly to their actions:

“Any software that does something that causes harm to a user, computer, or network can be considered malware [...]” (SIKORSKI; HONIG, 2012)

Mitnick & Simon’s (2003) definition is based on the software behavior during its execution and also its interactions with the user:

“Another kind of malware - short for malicious software - puts a program onto your computer that operates without your knowledge or consent, or perform a task without your awareness.” (MITNICK; SIMON, 2003)

These applications, purportedly built with intentions of reading, copying, or modifying information from computer systems - often without user consent - pose a high threat for modern information systems (LI et al., 2020; Microsoft Corporation, 2019; Symantec Corporation, 2019). The early detection of such malware is vital to minimize their effects on an organization, or even among regular users, and it is critical to maintain the **confidentiality, integrity** and **availability** of information.

Malware development can be seen as a large industry nowadays, due to its economical impact as both a revenue source for its developers and a mandatory cost for enterprises of different sizes willing to protect their data. This thriving business has a very flexible infrastructure, providing malware as services for several intents (Microsoft Corporation, 2021). It means that the defense mechanisms need to be updated at the same rate, in

a co-evolution process. Sophisticated attacks creates the demand for more sophisticated defense which requires even more creativity by resilient attackers.

Given this rapidly evolving nature of malware campaigns globally, long-established techniques for the assessment of samples can not always keep up with the amount of changes found in the wild. This is when machine-learning based methods can be useful - due to their power of generalization and scalability - to be integrated alongside existing methods in the task of identifying and mitigating possible damages caused by malware (SAXE; SANDERS, 2018).

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 (Microsoft 365 Defender Threat Intelligence Team, 2020; CHEN et al., 2020).

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 (ANDERSON et al., 2017). 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 (NATARAJ et al., 2011; GROSSE et al., 2016; ATHIWARATKUN; STOKES, 2017; YUE, 2017; CHEN, 2018; CHEN et al., 2020; RAFF et al., 2017; LE et al., 2018; SU et al., 2018; KHORMALI et al., 2019; BENKRAOUDA et al., 2021). We want to evaluate the vulnerability of these variants to the already known adversarial examples (GOODFELLOW; SHLENS; SZEGEDY, 2015), an approach with increasing popularity in the literature, especially in the context of malware (GROSSE et al., 2016; ANDERSON et al., 2017; AL-DUJAILI et al., 2018; KHORMALI et al., 2019; DEMETRIO et al., 2019, 2021; BENKRAOUDA et al., 2021; LUCAS et al., 2021). 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. Also, executable files have strict standards, which means byte ordering is relevant in some parts of the file. As mentioned earlier, we tried to conform our manipulations to the expected standards in order to preserve the functionality of the malware samples.

The rest of this dissertation is presented as follows: in Chapter 2 we provide high level definitions used throughout the text. In Chapter 3 we compare our methodologies to other present in the literature. In Chapter 4 we present how we generate and add data between sections of a Portable Executable (PE) file and also discuss the machine learning algorithms evaluated in this paper for malware classification. In Chapter 5 we discuss our evaluation strategies and their results, finishing with our conclusions in Chapter 6.

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 regards both of the location and also the amount of injected data.



## BACKGROUND

In this work we focus on the automatic detection and classification of malware samples with machine learning, and how data injection affects the performance of these algorithms. We are interested in identifying which family a given sample belongs to and if it is a derivative of other existing samples.

The accuracy of those machine learning algorithms has a strong correlation with the structure of the input data and the feature variation among different examples (GOOD-FELLOW; BENGIO; COURVILLE, 2016). It is important to understand the complexity of the used features since they are learned by the chosen algorithms.

Some important concepts are defined in this chapter before we dive into the methodologies explored in this work.

### 2.1 MALWARE ANALYSIS

Analyzing a given piece of software to determine its behavior on a computer system is a particularly challenging task. The analyst observes the sample, before, during and even after its execution to register its actions (SIKORSKI; HONIG, 2012). The entire process is divided into specific sets of procedures with different goals, that can be executed independently and in different orders.

Usually, the first step to be considered is the **static analysis**, the process of obtaining information about the sample without executing it. During this step, the goal is to obtain an identifier through hashing (useful for reverse searching in existing malware databases), analyze its set of instructions by disassembling it, checking its resources to be shown to the user (such as images or strings of messages), its imported libraries and dependencies.

Malware authors look for ways of hardening this process mainly through packing and obfuscation, techniques in which the original malware static data is hidden inside another layer which gets uncovered during its execution. This way the original behavior is kept intact while the appearance of the static file might look different.

Another step to be considered for the analysis is by actually executing the file, the **dynamic analysis**. This is usually performed in controlled environments, with the usage of sandboxes - virtual or physical machines, isolated in a dedicated network - to

analyze the entire lifecycle of the analyzed file. During this step some more information is collected, such as the endpoints the malware might try to connect through the network, the files it accesses, the configurations it applies to the operating system, the order and amount of system calls it performs. Just like in the static analysis, there are many countermeasures used by malware authors to make it harder to understand a given family of malwares. Some samples might stay dormant for several hours after initial infection to avoid being detected right away, activating after a certain amount of time or after a command comes through the network. It might also try to check information about the operating system to make sure it is a real system and not a sandbox (*e.g.* a virtual machine without internet connection or with a disabled antivirus software). Many of these countermeasures increases greatly the amount of effort required to fully understand the real behavior of the malware.

## 2.2 FILE FORMAT

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 distinct 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' PE32 format, as it is the only one included in publicly available malware datasets (NATARAJ et al., 2011; RONEN et al., 2018).

As shown in Figure 2.1, PE32 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 like ".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.

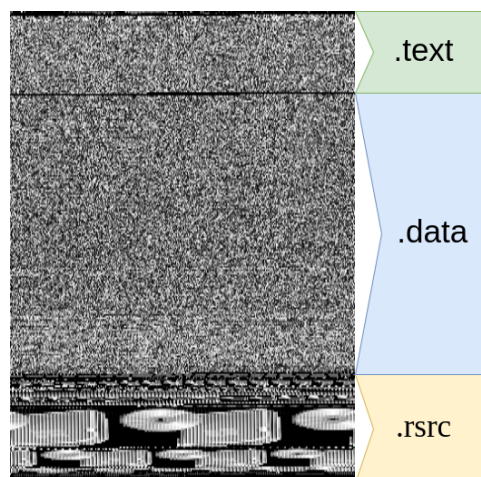


Figure 2.1: Illustration of the sections of a PE32 executable file.

## 2.3 MALWARE FEATURE EXTRACTION

As mentioned earlier, several types of information can be extracted from malware samples during static and dynamic analysis. If we are looking for automating this process, we need to somehow categorize this data in a way our machine learning algorithms can process and learn upon. Several characteristics have been used in the literature:

- **N-grams:** by analyzing an input file as a sequence of words, a N-gram can be seen as a sequence of N adjacent words, N being an integer number. For example, we can split a phrase such as "Blue sky with no clouds." in N-grams with N=2 we have the following groups: "Blue sky", "sky with", "with no", and "no clouds". This technique can be used to extract N-grams from the set of instructions of a sample (RAFF et al., 2018). This kind of feature can point to the frequency and the order that certain instructions appear in a sample's machine code.
- **Raw bytes:** the sequential bytes of the analyzed sample can be passed as a single array of features to a neural network (RAFF et al., 2017).
- **System calls:** The actions taken by a file during its execution can be stored and used as features for machine learning algorithms (KOLOSNAJI et al., 2016).
- **Histogram of byte's entropy:** The entropy - a measure of how much variation exists among bytes values - can be computed and used as a discriminative feature for malware samples (SAXE; BERLIN, 2015).
- **Imported libraries:** Libraries and functions used by a given sample are considered relevant to identify the semantics of the system calls performed by a malware family (SAXE; BERLIN, 2015).
- **File metadata:** static information such as compilation date, target architecture (*i.e* 32-bit or 64-bit) and number of sections can be used as characteristics of analyzed files (SAXE; BERLIN, 2015).

Those extracted features can then be used to group similar samples into malware families, given that they share similar traits up to a certain threshold. Despite not having the exact same sequence of bytes, there might be similarities among samples both in the aforementioned features and in their behavior. We are interested in exploring machine learning techniques to automatically perform this grouping.

In this work we have the intention of applying **representational learning**, *i.e* algorithms that can learn which characteristics are discriminative enough to classify a sample, and also understanding how these algorithms can be compared to feature-engineered ones. The techniques explored here can be thought of as part of the static analysis of a software sample, where we perform the classification without executing it.





## RELATED WORKS

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

### 3.1 MALWARE CLASSIFICATION

Nataraj et al. (2011) 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 Neighbours (KNN)) methods (AGARAP; PEPITO, 2018; LIU et al., 2019).

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 features from the file (*e.g.*, system calls, imported libraries, functions in use) (PASCANU et al., 2015; SAXE; BERLIN, 2015; ATHIWARATKUN; STOKES, 2017; ANDERSON; ROTH, 2018) or by using the raw bytes from the data as input (RAFF et al., 2017, 2018; HADDADPAJOUH et al., 2018). Some of them achieve high classification accuracy by training CNN from scratch (SU et al., 2018; LIU et al., 2019), or by using prior knowledge from a CNN pre-trained on a large dataset (YUE, 2017; CHEN, 2018; CHEN et al., 2020) like ImageNet (DENG et al., 2009). 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 (CHEN, 2018; RAFF et al., 2017). These networks, however, are vulnerable to adversarial attacks. In this work we explore various architectures - KNN+GIST as proposed by Nataraj et al. (2011), CNN, CNN-LSTM and CNN-BiLSTM as proposed by Le et al. (2018) and Mal-Conv as proposed by Raff (2017) (RAFF et al., 2017) - and how they behave against crafted adversarial 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(NATARAJ

et al., 2011), BIG 2015 (RONEN et al., 2018) and EMBER (ANDERSON; ROTH, 2018) are the most notable ones. Since our injection method requires reading the file header, BIG 2015 (RONEN et al., 2018) dataset is not possible because the samples have their headers stripped. EMBER (ANDERSON; ROTH, 2018), 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 3.1 we aggregate state of the art methods for malware detection and classification by their technique and the dataset it used.

Table 3.1: Summary of different malware classification techniques.

<b>AUTHOR</b>	<b>TECHNIQUE</b>	<b>DATASET</b>
Nataraj et al. (2011)	GIST + KNN	maling(NATARAJ et al., 2011)
Pascanu et al. (2015)	ESN + Logistic Regression	Private
Athiwaratkun e Stokes (2017)	LSTM + MLP	Private
Yue (2017)	CNN	maling(NATARAJ et al., 2011)
Raff et al. (2017)	Embedding + CNN	Private
Anderson e Roth (2018)	Embedding + CNN	Ember (ANDERSON; ROTH, 2018)
Su et al. (2018)	CNN	Private
HaddadPajouh et al. (2018)	LSTM	Private
Liu et al. (2019)	Multilayer SIFT	maling(NATARAJ et al., 2011), BIG 2015(RONEN et al., 2018)
Agarap e Pepito (2018)	GRU + SVM	maling(NATARAJ et al., 2011)
Le et al. (2018)	CNN-BiLSTM	BIG 2015(RONEN et al., 2018)
Chen (2018)	Inception-V1 (SZEGEDY et al., 2016)	maling(NATARAJ et al., 2011), BIG 2015(RONEN et al., 2018)
Chen et al. (2020)	Inception-V1 (SZEGEDY et al., 2016)	Private

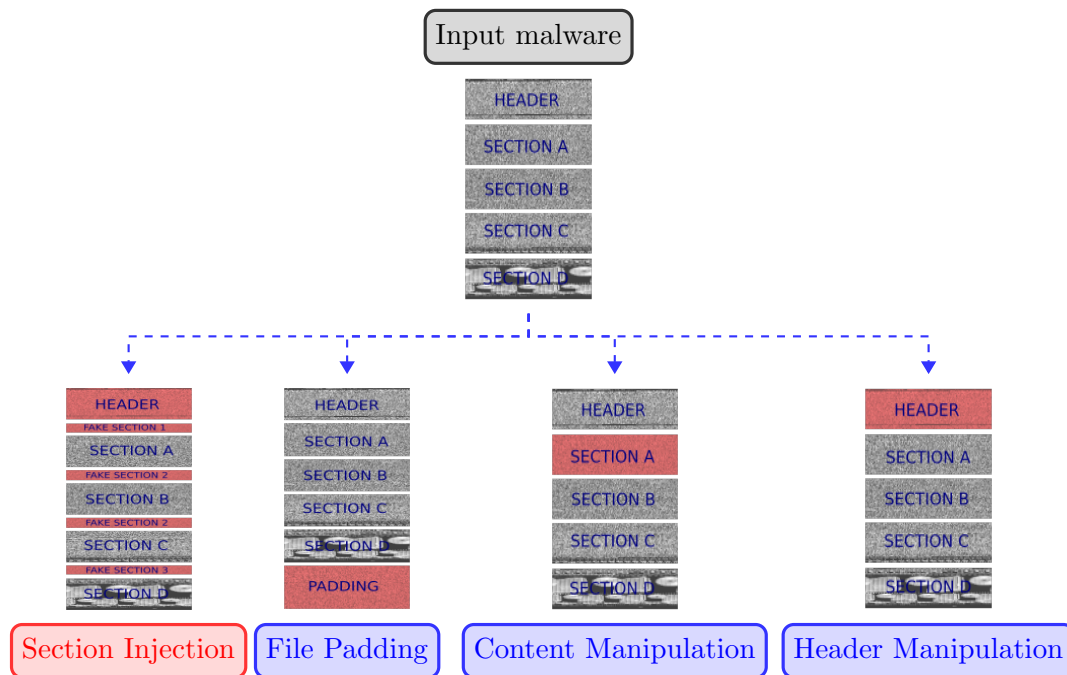


Figure 3.1: Illustration of the differences among attacks to image-based malware classifiers. The leftmost (red) square display our approach. Blue squares displays previous attacks.

### 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. But 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 literature some possible attacks that retain their functionalities. They are illustrated in Figure 3.1.

Different works exploited adversarial attacks in the malware domain. Grosse et al. (2016) and Al-Dujaili et al. (2018) extracted static features from malware files and used the Fast Gradient Sign Method (FGSM) (GOODFELLOW; SHLENS; SZEGEDY, 2015) 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. (2017) explore 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 “Section Injection” and “Content Manipulation” categories illustrated in Figure 3.1.

It achieves evasion rates up to 16% against a Gradient Boost Decision Tree (GBDT) model (ANDERSON; ROTH, 2018).

Khormali et al. (2019) focused on injecting bytes to the executable files’ end, an unreachable area during execution. It fits “File Padding” category illustrated in Figure 3.1. As the operating system will not execute it, 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. Besides, 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. (2019) propose a black-box attack called Genetic Adversarial Machine learning Malware Attack (GAMMA), 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(RAFF et al., 2017) - a shallow neural network - and GBDT (ANDERSON; ROTH, 2018). Its proposed methods fit all categories illustrated in Figure 3.1, despite not detailing how some of those are achieved.

Lucas et al. (2021) also employs functionality preserving techniques. They extend binary rewriting techniques such as In-place Randomization (IPR) (PAPPAS; POLYCHRONAKIS; KEROMYTIS, 2012) - where the binary is disassembled and some of its instructions are rewritten - and code displacement (Disp) (KOO; POLYCHRONAKIS, 2016) - where the disassembled version is also used, but with the intent of moving instructions between sections, fitting into “Content Manipulation” category illustrated in Figure 3.1. 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. (2021) propose a framework that mixes a mask generator to highlight the bytes that are possible to manipulate while retaining executability, adversarial example generation using Carlini-Wagner (CW) attack (CARLINI; WAGNER, 2017) 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 3.1. 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. (2021) 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 (RAFF et al., 2017), DNN with Linear (DNN-Lin) and ReLU (DNN-ReLU) (COULL; GARDNER, 2019) and GBDT (ANDERSON; ROTH, 2018), 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. (2017) and Demetrio et al. (2019), 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 *Extend* and *Shift* methods proposed by Demetrio et al. (2021) and the *padding* methods discussed by Khormali et al. (2019). The byte modifications presented by Lucas et al. (2021) 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 works listed here used some sort of private dataset, either by collecting samples from malware hosting services or expanding public ones - Benkraouda et al. (2021) merged maling(NATARAJ et al., 2011) and benign samples from Architecture Object Code Dataset (AOCD) (CLEMENS, 2015), Khormali et al. (2019) used BIG 2015(RONEN et al., 2018) and also formed a private IoT dataset. A summary of the functionality preserving attacks can be found in Table 3.2.

Table 3.2: Summary of functionality preserving attacks against PE32 malware classification.

<b>AUTHOR</b>	<b>METHODS</b>	<b>TARGETS</b>	<b>DATASET</b>
Anderson et al. (2017)	Set of Manipulations	<i>GBDT</i> (ANDERSON; ROTH, 2018)	Private
Khormali et al. (2019)	Padding	3-layer CNN	BIG 2015(RONEN et al., 2018) + Private IoT dataset
Demetrio et al. (2019)	Set of Manipulations	<i>MalConv</i> (RAFF et al., 2017), <i>GBDT</i> (ANDERSON; ROTH, 2018)	Private
Demetrio et al. (2021)	Partial DOS, Full DOS, Extend, Shift, FGSM, Padding	<i>MalConv</i> (RAFF et al., 2017), <i>DNN</i> (COULL; GARDNER, 2019), <i>GBDT</i> (ANDERSON; ROTH, 2018)	Private
Lucas et al. (2021)	IPR, Disp	<i>AvastNet</i> (KRČÁL et al., 2018), <i>MalConv</i> (RAFF et al., 2017), <i>GBDT</i> (ANDERSON; ROTH, 2018)	Private
Benkraouda et al. (2021)	Adversarial Generation + Optimization	CNN(KHORMALI et al., 2019; KOLOSNAJJI et al., 2016)	Private (combination of malimg(NATARAJ et al., 2011) and AOCD(CLEMENS, 2015))

## METHODOLOGY

Figure 4.1 illustrates the core principle of this work: tampering with input malware’s original bytes before feeding them into the classifier. As previously discussed in Chapter 3, our tampering is done by editing the malware’s header and adding data between already existing sections. This positional tampering is important due to the different nature of evaluated classifiers. Some of them might truncate the data to a specific input length, while others decide to reinterpret raw bytes as a grayscale image, as depicted by the second step in Figure 4.1. Our goal is to have a simple attack that works against a broad range of machine learning models. We explain how the proposed injection process works in Section 4.1, and we show how we built the malware classifiers used in our experiments in Section 4.2.

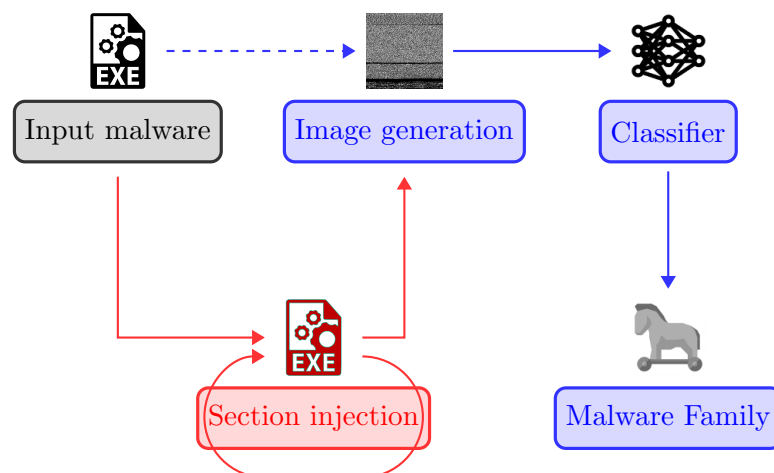


Figure 4.1: Flowchart of an image-based malware classification system (blue lines). Red lines replace the dashed blue line in our data injection scheme.



## 4.1 DATA INJECTION

To comply with a realistic usage scenario, we inject one or more sections filled with arbitrary bytes before any processing is done for classification purposes. This injection is performed sequentially, *i. e.* when multiple sections are to be included the result of an injection is used as input for the next one.

On a first look, the semantics of the injected bytes are not of particular interest. We rather focus on understanding how much structural change the classifiers can handle. Despite that, we also explore injecting adversarial bytes using the same routine. Those results are discussed further in Sections 5.3 and 5.4 respectively.

### 4.1.1 File header

The first step in our injection scheme is to obtain information about the input file by reading its header. Table 4.1 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  $[0, \textit{NumberOfSections}]$ . Sections 0 to  $k - 1$  remain in place, and sections  $k$  to  $\textit{NumberOfSections} - 1$  are shifted one position forward so that we can insert the new section in  $k$ -th place.

Table 4.1: Flags in the header of PE32 files.

FLAG NAME	DESCRIPTION
<i>NumberOfSections</i>	Number of sections in the file
<i>FileAlignment</i>	Section size in bytes is a multiple of this flag
<i>SectionAlignment</i>	Memory address of a section is a multiple of this flag
<i>SizeOfImage</i>	Memory size of all sections in bytes
<i>ImageBase</i>	Address of the first byte when the file is loaded to memory (default value is 0x00400000)

### 4.1.2 Section header

A section header is composed of 40 contiguous bytes. These bytes specify what the loader needs to handle this section. Table 4.2 shows the bytes that we fill when creating a new section. We refer to a flag of the  $i$ -th section as  $Flag_i$ .

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 4.1:

$$SizeOfRawData_k = \lceil \frac{N}{FileAlignment} \rceil \times FileAlignment \quad (4.1)$$

with  $N$  being the number of bytes we want to add. We always set  $N$  as a multiple of

Table 4.2: Flags in section headers of PE32 files.

FLAG NAME	SIZE	DESCRIPTION
<i>Name</i>	8 bytes	Section name
<i>VirtualSize</i>	4 bytes	Section size in bytes on memory
<i>VirtualAddress</i>	4 bytes	Section offset on memory relative to <i>ImageBase</i>
<i>SizeOfRawData</i>	4 bytes	Section size in bytes on disk
<i>PointerToRawData</i>	4 bytes	Section offset on disk relative to the beginning of the file
<i>Characteristics</i>	4 bytes	Section characteristics like usage and permissions

*FileAlignment* so that null padding is unnecessary. *FileAlignment* is usually 512 bytes, but it varies according to compilation options.

*PointerToRawData<sub>k</sub>* is set as in Equation 4.2, if the  $k$ -th section is the last one. Or else as in Equation 4.3, and we add *SizeOfRawData<sub>k</sub>* to *PointerToRawData<sub>i</sub>*,  $\forall i > k$ .

$$PointerToRawData_k = PointerToRawData_{k-1} + SizeOfRawData_{k-1} \quad (4.2)$$

$$PointerToRawData_k = PointerToRawData_{k+1} \quad (4.3)$$

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<sub>k</sub>* using Equation 4.4:

$$VirtualAddress_k = \lceil \frac{VirtualAddress_L + VirtualSize_L}{SectionAlignment} \rceil \times SectionAlignment \quad (4.4)$$

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

*VirtualSize<sub>k</sub>* 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<sub>k</sub>* as a read-only section with initialized data.

### 4.1.3 Injected data

In our work, injected data is 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 (KHORMALI et al., 2019). However, we do not do that because we assume we have no access to models and training data used by malware classifiers. Besides, our results show that our simple strategy is enough to affect the performance of a state-of-the-art malware classification approach substantially.

#### 4.1.4 Workarounds

We found some challenges when applying this method to an arbitrary PE32 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.

## 4.2 MALWARE CLASSIFICATION

As can be seen in Figure 4.1, this process is divided into two parts: image generation and classification. The former is described in Section 4.2.1. The latter is carried out with various approaches:

1. GIST+KNN (NATARAJ et al., 2011), which holds state-of-the-art performance for handcrafted methods. In this work we reproduced Nataraj *et al.*'s approach (NATARAJ et al., 2011) to the best of our abilities. To do so, we resize our images to  $64 \times 64$  pixels, extract 320-dimensional GIST descriptors, and then classify it using KNN with  $K = 3$ .
2. Le et al. (2018) presents three models:
  - (a) A simple model with three 1D-CNN layers before a fully connected layer, referred to as Le-CNN.
  - (b) A second model with a LSTM layer before the fully connected one, referred to as Le-CNN-LSTM.
  - (c) A third model with a bidirectional LSTM before the fully connected layer, referred to as Le-CNN-BiLSTM.

For all of them we employ the same input size of 10k 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).

3. Raff et al. (2017) presents 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* (DEMETRIO et al., 2019); input size of 1MB, training for a total of 10 epochs without early stopping, with a batch size of 16 due to memory constraints.

These approaches were chosen because, among those discussed earlier in Chapter 3, they provide reasonable performance  $\times$  resource usage tradeoffs. Besides that, the code used to construct the architecture is available publicly, which improves substantially the reproducibility of the results.

Different types of neural networks were explored to classify malware files (PASCANU et al., 2015; SAXE; BERLIN, 2015; ATHIWARATKUN; STOKES, 2017; RAFF et al., 2017, 2018; HADDADPAJOUH et al., 2018; SU et al., 2018; YUE, 2017; CHEN, 2018; LE et al., 2018). But to the best of our knowledge, those based on CNNs are the ones with the highest accuracy. Comparing them to the baseline method of GIST+KNN provides an insight into how discriminative the malware raw data can really be and how hard it is to learn these features automatically.

The core ideas behind the trained models are discussed more thoroughly in Sections 4.2.2, 4.2.3 and 4.2.4, respectively.

### 4.2.1 Image generation

We transform an executable into an image following Chen’s adaptation (CHEN, 2018) of Nataraj *et al.*’s specifications (NATARAJ et al., 2011). 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 4.3). We discard the last row if it is incomplete. The result is illustrated in Figure 2.1.

Table 4.3: Image width based on the executable size (NATARAJ et al., 2011).

NUMBER OF BYTES (kB)	IMAGE WIDTH (pixels)
< 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

### 4.2.2 GIST+KNN

As mentioned earlier, reinterpreting the malware bytes as a grayscale image is a relevant strategy for data preprocessing. In order to classify the image, we need to somehow describe its content in a way that the classification algorithm can use. This is exactly the attribution of the GIST descriptor.

Figure 4.2 illustrates how feature extraction is achieved using GIST. We start by resizing input images to  $64 \times 64$  pixels. Gabor filters are then applied to the resized images. Those filters are manipulated through their scale ( $\lambda$  parameter, “width” of the filter’s ellipsoids) and their orientation ( $\theta$  parameter, rotation angle of the filter’s ellipsoids). Three different scales are applied: eight orientations on the first and second

scales, four on the third one, for a total of 20 different filters. After that, resulting images are subdivided into  $4 \times 4$  grids and the average value of each subregion becomes a value for the final descriptor. Therefore, the resulting descriptor for this operation has 320 values (20 filters  $\times$  16 image subregions). For colored images, commonly structured with three-channels (RGB), these operations are done separately in each channel, producing a descriptor with 960 values (OLIVA; TORRALBA, 2001).

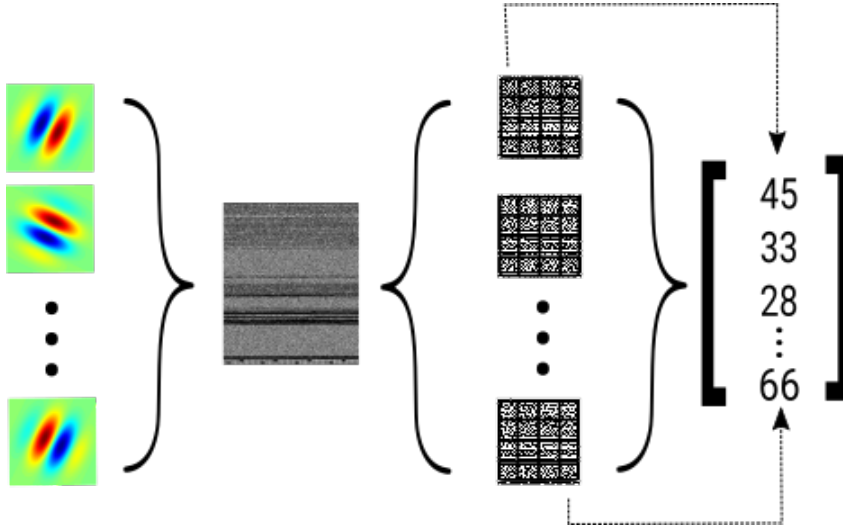


Figure 4.2: Image description using GIST feature extractor.

After obtaining these 1D vectors of features, we process them through K-Nearest Neighbors (KNN) algorithm to obtain the image classification. This is a *lazy* technique, which means that there is no fixed model being built and there are no internal parameter updates during training. KNN works simply by computing the distance between the input sample and the already known ones, with K being the number of nearest neighbors picked to choose the class of the analyzed sample. This process is illustrated by Figure 4.3.

A simple distance metric such as Euclidean distance, displayed by equation 4.5, can be used to compute the similarity between the vectors. In this context, N is the length of the feature vector, *e.g.* the number of features, that represent  $p$  and  $q$  observations.

$$d(p, q) = \sqrt{\sum_{i=1}^N (q_i - p_i)^2} \quad (4.5)$$

### 4.2.3 CNN and LSTM

All three models introduced by Le *et al* (LE et al., 2018) have a fixed input size to be processed by three 1D-CNN layers right at the start. They are used with a similar intent as mentioned earlier in Section 4.2.2: extracting relevant features to describe the contents of the input, using a fixed number of filters. The difference here is that these filters have their values updated during the training of the network, allowing for a deeper

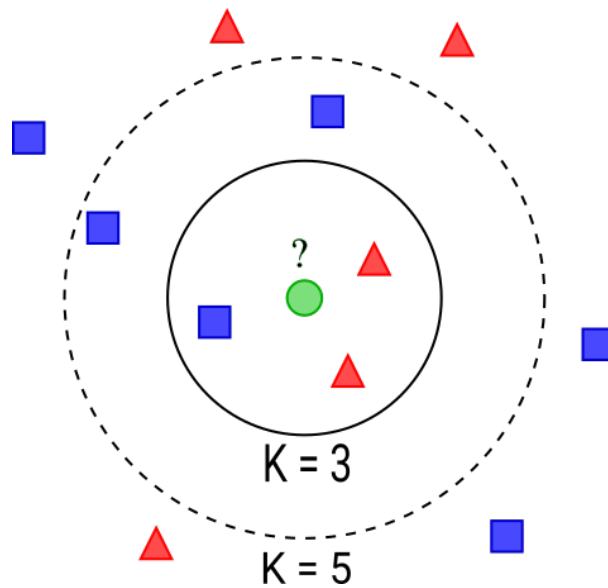


Figure 4.3: Classifying samples using K-Nearest Neighbors algorithm.

understanding of the extracted features. Figure 4.4 illustrates how the malware data is acquired and processed by these initial layers.

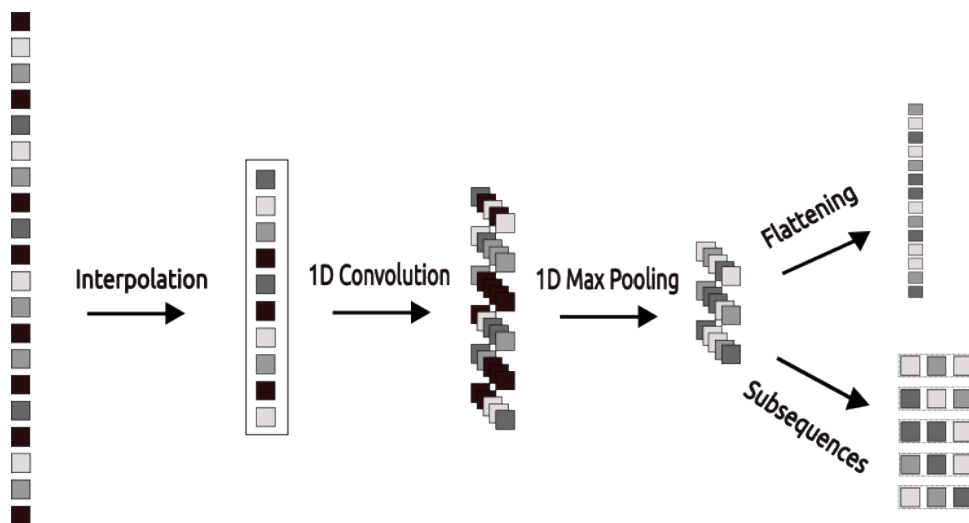


Figure 4.4: Illustration of the data flow in Le *et al* (LE et al., 2018) architectures. Transforming malware data to a fixed length. Feature extraction and dimensionality reduction through 1D CNN and 1D pooling layers. Since the input is scaled to a fixed size of 10K bytes, after these operations we have a matrix of size  $78 \times 90$ . We can either unroll to a 1D feature vector and classify it using fully connected layers or consider it as smaller subsequences and use recurrent layers.

First the input data is scaled to a vector with a fixed size of 10K bytes using bilinear interpolation. Since the average sample size on this dataset is 150kbytes, it means that

for Le2018 models we are downscaling the input size by resampling the data. No filter is applied before this transformation, which means aliasing might happen during this process. After that, the features are extracted through the 1D-CNN layers using 30, 50 and 90 filters respectively, and kernels of size 7. After each Convolutional layer a Max Pooling layer is used, 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 (GOODFELLOW; BENGIO; COURVILLE, 2016). After the last pooling layer, we have a  $(78 \times 90)$  matrix. This is the point where the models take different approaches:

1. reinterpreting this matrix as a vector of size 7020 and proceeding with two fully connected layers to obtain the final classification;
2. passing the data through a recurrent layer to extract positional features and then a fully connected layer to obtain the final classification.

Repeating a process many times using the outcome of one entry as information source to the next, *i.e.* recurrence, is the core idea for LSTMs, particularly useful when dealing with large data sequences (GOODFELLOW; BENGIO; COURVILLE, 2016). It happens due to their internal structures for state and memory control, as displayed by Figure 4.5. It requires splitting a long sequence into smaller subsequences of known size. In this case, given the output from the previous layer, we have 90 subsequences of size 78.

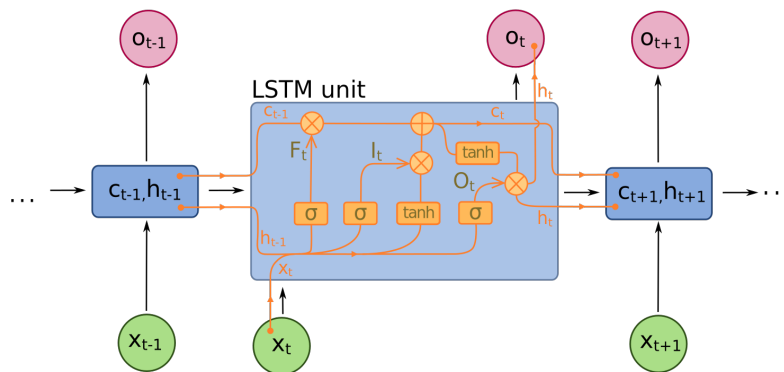


Figure 4.5: An example of a single Long Short-Term Memory cell.

As seen in Figure 4.5, the cell receives input  $x_t$ , current processed sequence, together with state  $c_{t-1}$  and output  $h_{t-1}$  both resulting from processing the previous sequence. Operating over  $x_t$  produces state  $c_t$  and output  $h_t$ , both used to process the next subsequence. These operations are shown at Equations 4.6 through 4.11.

$$F_t = \sigma(W_f \odot [h_{t-1}, x_t] + b_f) \quad (4.6)$$

$$I_t = \sigma(W_i \odot [h_{t-1}, x_t] + b_i) \quad (4.7)$$

$$\hat{C}_t = \tanh(W_{\hat{C}} \odot [h_{t-1}, x_t] + b_{\hat{C}}) \quad (4.8)$$

$$O_t = \sigma(W_o \odot [h_{t-1}, x_t] + b_o) \quad (4.9)$$

$$c_t = F_t \odot c_{t-1} + I_t \odot \hat{C}_t \quad (4.10)$$

$$h_t = O_t \odot \tanh(c_t) \quad (4.11)$$

In a LSTM cell, the data is handled through “gates”, substructures responsible for managing the input and updating the cell state. The first one of these is the “forget gate”. It decides what should be kept from previous output ( $h_{t-1}$ ) for the next steps.

The **number of units** is used to set the size of state vector ( $c_t$ ). Figure 4.6 illustrates the output  $h_{t-1}$  being concatenated to  $x_t$  and the resulting vector is multiplied by the weights  $W_f$ , added to bias  $b_f$  and handled by a sigmoid function (Equation 4.12), resulting in vector  $F_t$ . This function transforms its input values to  $[0, 1]$  range. This multiplication can be seen as deciding which values from previous output will be kept (multiplied by values close to 1) or *forgot* (multiplied by values close to 0).

$$y = \frac{1}{(1 + e^{-x})} \quad (4.12)$$

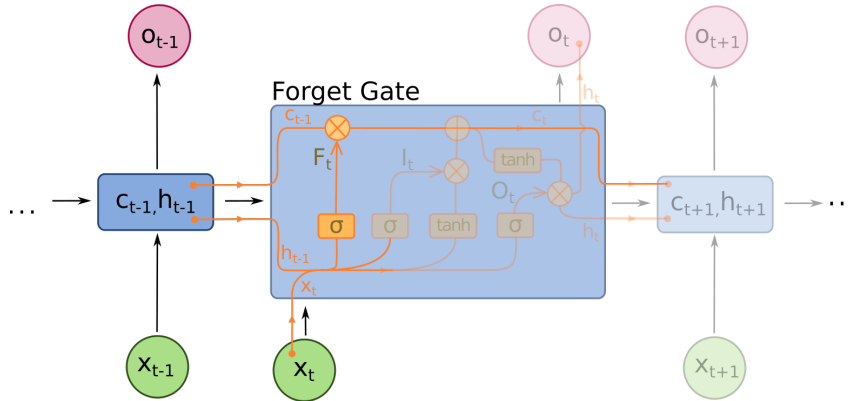


Figure 4.6: Operations performed at the “forget gate”.

The number of units chosen for these models was 128. It means that concatenating  $x_t$  and  $h_{t-1}$  results on a vector of size 206. Weights matrix  $W_f$  on this gate has  $206 \times 128$  values and 128 values for bias ( $b_f$ ). This gate is the application of Equation 4.6 and the first part of Equation 4.10, where  $F_t$  is multiplied by  $c_{t-1}$ .

At the “input gate” what will be kept from current input for the next iterations, by using Equations 4.7 and 4.8. As shown by Figures 4.7, the concatenation of  $x_t$  and  $h_{t-1}$  is multiplied by  $W_i$  and the result is added to  $b_i$ . This result is then passed to the sigmoid function, resulting in  $I_t$ .  $\hat{C}_t$  is defined by multiplying  $[x_t, h_{t-1}]$  by the weights  $W_{\hat{C}_t}$  and



adding the bias  $b_{\hat{C}_t}$ . The result of these operations is then processed by the hyperbolic tangent function (Equation 4.13). It transforms input values to  $[-1, 1]$  range. The main attribution of this function within the cell is to control the weights values, by avoiding their rapid growth or allowing it to decrease if necessary.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (4.13)$$

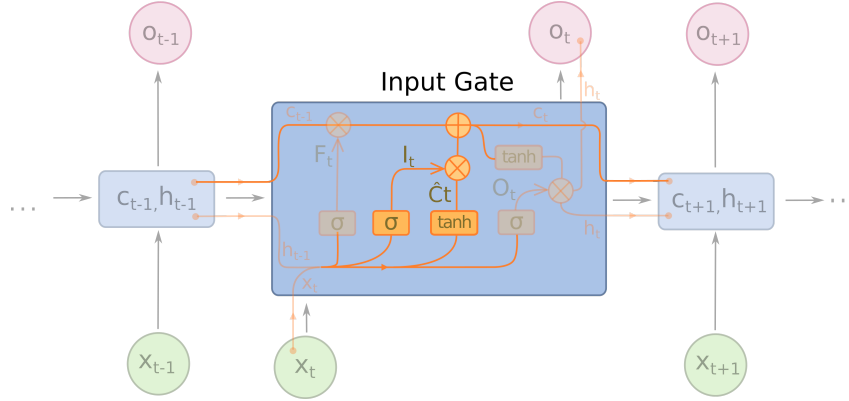


Figure 4.7: Operations performed at the “input gate”.

Matrices  $W_i$  and  $W_{\hat{C}_t}$  have  $206 \times 128$  as dimensions, and biases  $b_i$  and  $b_{\hat{C}_t}$  are of size 128. The second part of Equation 4.10 is executed here, multiplying  $\hat{C}_t$  by  $I_t$  and adding the result to the new value of  $c_{t+1}$  (updated at the forget gate).

The “output gate” decides what will be sent as state ( $c_t$ ) and as output ( $h_t$ ). As shown by Figure 4.8, the concatenation of  $x_t$  and  $h_{t-1}$  is once again passed through the sigmoid function (after being multiplied by  $W_o$  and added to  $b_o$ ), producing  $O_t$ .  $c_t$  (already updated at forget and input gates) is processed by the hyperbolic tangent function and then multiplied by  $O_t$ , resulting in  $h_t$  (Equation 4.11).

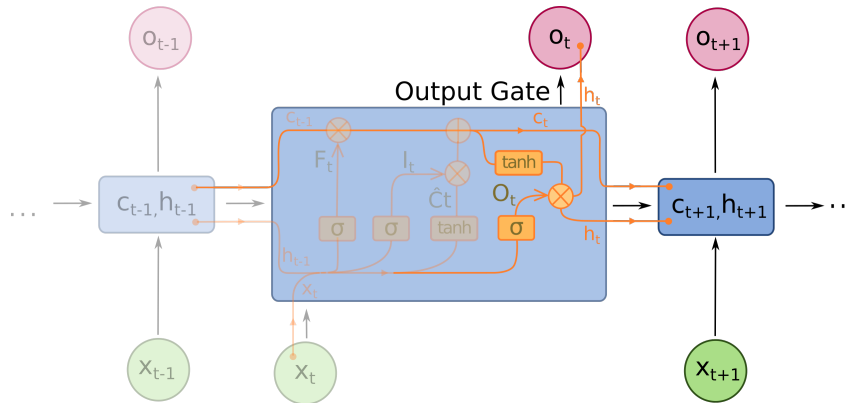
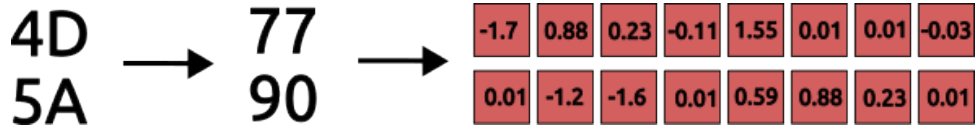


Figure 4.8: Operations performed at the “output gate”.

After passing through all steps in the LSTM cell, we have our output vector of size 128. In the case of the bidirectional LSTM, the data is also analyzed in its reversed form,



(a) Embedding vector transformation. Each input byte from the malware data becomes a vector with a fixed size.

Diagram (b) illustrates the gating mechanism. A 1x8 vector of values is multiplied by the sigmoid of a convolution operation applied to the embedding vector:

$$\begin{bmatrix} -1.7 & 0.88 & 0.23 & -0.11 & 1.55 & 0.01 & 0.01 & -0.03 \end{bmatrix} = \text{conv} \left( \begin{bmatrix} -1.7 & 0.88 & 0.23 & -0.11 & 1.55 & 0.01 & 0.01 & -0.03 \\ 0.01 & -1.2 & -1.6 & 0.01 & 0.59 & 0.88 & 0.23 & 0.01 \end{bmatrix} \right) \times \sigma \left( \text{conv} \left( \begin{bmatrix} -1.7 & 0.88 & 0.23 & -0.11 & 1.55 & 0.01 & 0.01 & -0.03 \\ 0.01 & -1.2 & -1.6 & 0.01 & 0.59 & 0.88 & 0.23 & 0.01 \end{bmatrix} \right) \right)$$

(b) Gating mechanism using embedding data. Same data goes through separate computations to produce the final feature vector.

Figure 4.9: Illustration of the embedding vector learning 4.9a and the transformation it goes through in the gating phase 4.9b.

resulting in a vector of size 256. We can then feed into the fully connected layer to obtain the classification.

#### 4.2.4 Gated CNN

The gating idea introduced with the LSTM cell above can be extended to be used without necessarily using a recurrent model. That is the core idea behind Gated Convolution Networks; controlling which kind of information is allowed to flow through the layers and the associated cost with each information unit.

The gating mechanism in this context was first introduced by Dauphin *et al* (DAUPHIN *et al.*, 2016), providing a non-recurrent model with a generalization power similar to recurrent architectures in language classification tasks. The idea was then extended by Raff *et al* (RAFF *et al.*, 2017) using MalConv model to classify malware samples, reinterpreting malware raw data as a language.

The first step in this architecture, as shown by Figure 4.9, is to transform the data through an Embedding layer - *i.e* using a lookup table to convert malware instructions into a set of integer values within a known range. In our context, it means mapping each input byte into an integer in the range  $[0, 257]$  (256 possible bytes plus an extra value for padding). The Embedding layer then learns during network training how to transform each value in this range into a vector of size 8 (the embedding vector). This mechanism is useful to learn the representation of all possible instructions using low dimensionality vectors.

Once acquired, the embedding vector then goes through two separate 1D-CNN layers. The output of one of these layers goes through the sigmoid function (Equation 4.12) and it is multiplied by the output of the other CNN layer. It produces a gating mechanism just like the one in the LSTM architecture mentioned earlier.

This architecture benefits from the control of the information, as in LSTM, but also from the location invariance brought by the convolutional layers. It is important in the malware domain because even though the language can be considered small (bytes 0-255), the context variance is remarkably high, and a given byte can appear in many locations

with different meanings.

## RESULTS

In this chapter we display our experimental results for different attack and defense scenarios. In Section 5.1 we provide an overview of the chosen dataset and its structure. In Section 5.2 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 5.3 and 5.4 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 5.5 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 5.6 we try to provide defense mechanisms against the attacks discussed previously, in three different fronts: using injected data in the training set in Section 5.6.1, creating a binary dataset by inflating the dataset with benign data in Section 5.6.2 and also finetuning larger models in Section 5.6.3.

### 5.1 DATASET

The forth coming experiments were made upon maling (NATARAJ et al., 2011) dataset, to evaluate malware classification before and after code injection. Table 5.1 displays the distribution of samples across all classes. It has 9,339 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 5.1 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 4.2.1. We can see that the class dissimilarities are mostly represented by lower density sections, *i.e.* image areas with higher number of

Table 5.1: Samples distribution and average size in maling 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
<b>TOTAL</b>	<b>25 families</b>	<b>9339 samples</b>	<b>176.29kB size</b>

black pixels. Texture can be seen in some samples, such as at the bottom of Figure 5.1c, usually caused by resource sections.

Another feature of this dataset is that most samples present a similar structure to the one illustrated in Figure 2.1, having *.text*, *.data* and *.rsrc* sections as the most prevalent

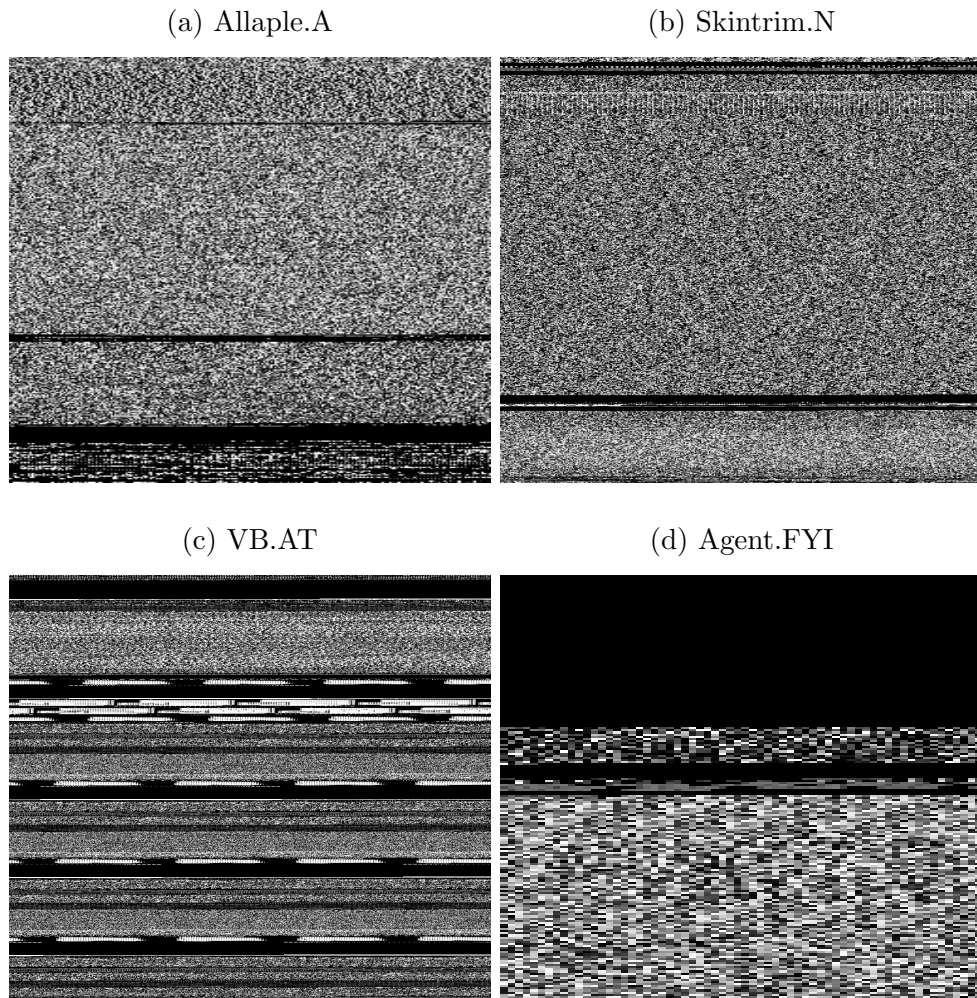


Figure 5.1: Illustration of samples from the class with most (5.1a) and the least (5.1b) number of samples. Also, class with higher (5.1c) and with smaller (5.1d) average size. Images were rescaled to use the same aspect ratio.

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.

## 5.2 METRICS

There are many different metrics and visualization techniques in the literature being used to evaluate 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 5.1, where True Positive (TP), True

Negative (TN), False Positive (FP) and False Negative (FN) are used.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (5.1)$$

- **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 5.1, 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 less 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 (DAVIS; GOADRICH, 2006; SAITO; REHMSMEIER, 2015). We also compute the Average Precision (AP) as described by Equation 5.2. 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.

$$AP = \sum_n (R_n - R_{n-1})P_n \quad (5.2)$$

Another relevant factor that might impact 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 (GOODFELLOW; BENGIO; COURVILLE, 2016).

### 5.3 INJECTION ATTACKS WITH RANDOM DATA

In this round of experiments 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 average results in Figure 5.2.

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

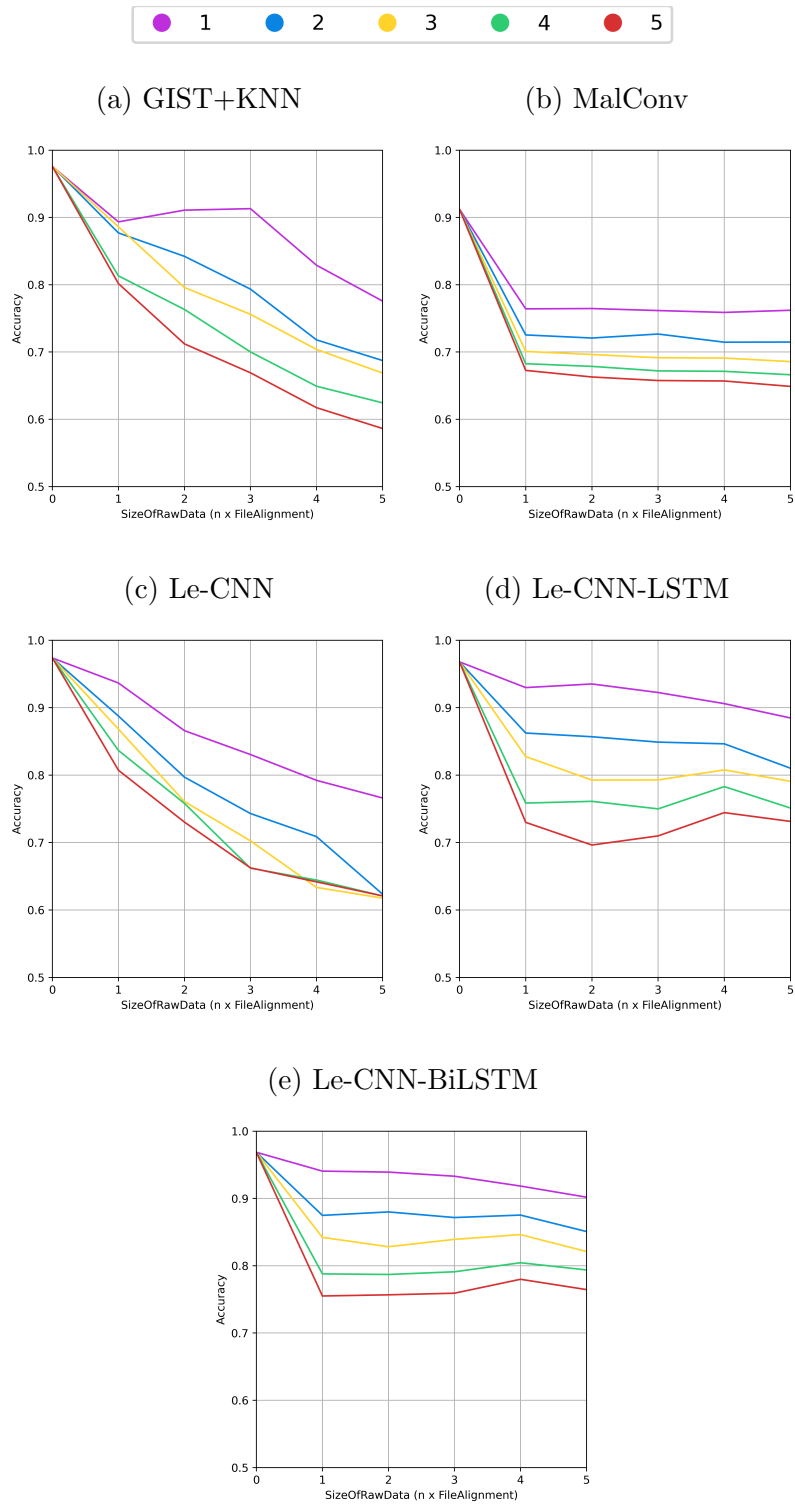


Figure 5.2: Average accuracy of malware classification under different injection scenarios. Distinct colors represent the amount of injected sections.



176kB, our injection approach accounts for an approximate 7% increase in file size and misclassification rates ranging between 25% and 40%. Figure 5.3 illustrates the misclassification differences between the test set with original samples and a set with injected samples.

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 (NATARAJ et al., 2011).

We can see that all models suffer to correctly classify these variants, even before data injection. Le-CNN-BiLSTM model, as seen in Figures 5.3c and 5.3d does not learn how to correctly identify samples from a packed family, *e.g.* “Autorun.K”. One behavior is clear in KNN and MalConv models: their tendencies to misclassify samples as “Autorun.K”, “C2LOP.P” and “C2LOP.gen!g”. Those families share samples with high average sizes, at 524.54kB, 386.92kB and 524.04kB, respectively. Since Le-CNN-BiLSTM resizes everything to 10k 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 5.4 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.

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.

## 5.4 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  $5 \times FileAlignment$ . Figure 5.4c displays the difference that injecting with adversarial data imposes.

Comparing with random injection results seen in Figure 5.4b, we can see that all models had its 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 highest average precision despite losing more accuracy than Le-CNN-BiLSTM (Figure 5.2).

## 5.5 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 (RONEN et al., 2018). Figure 5.5 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 5.5a. Only Le-CNN-BiLSTM increased its

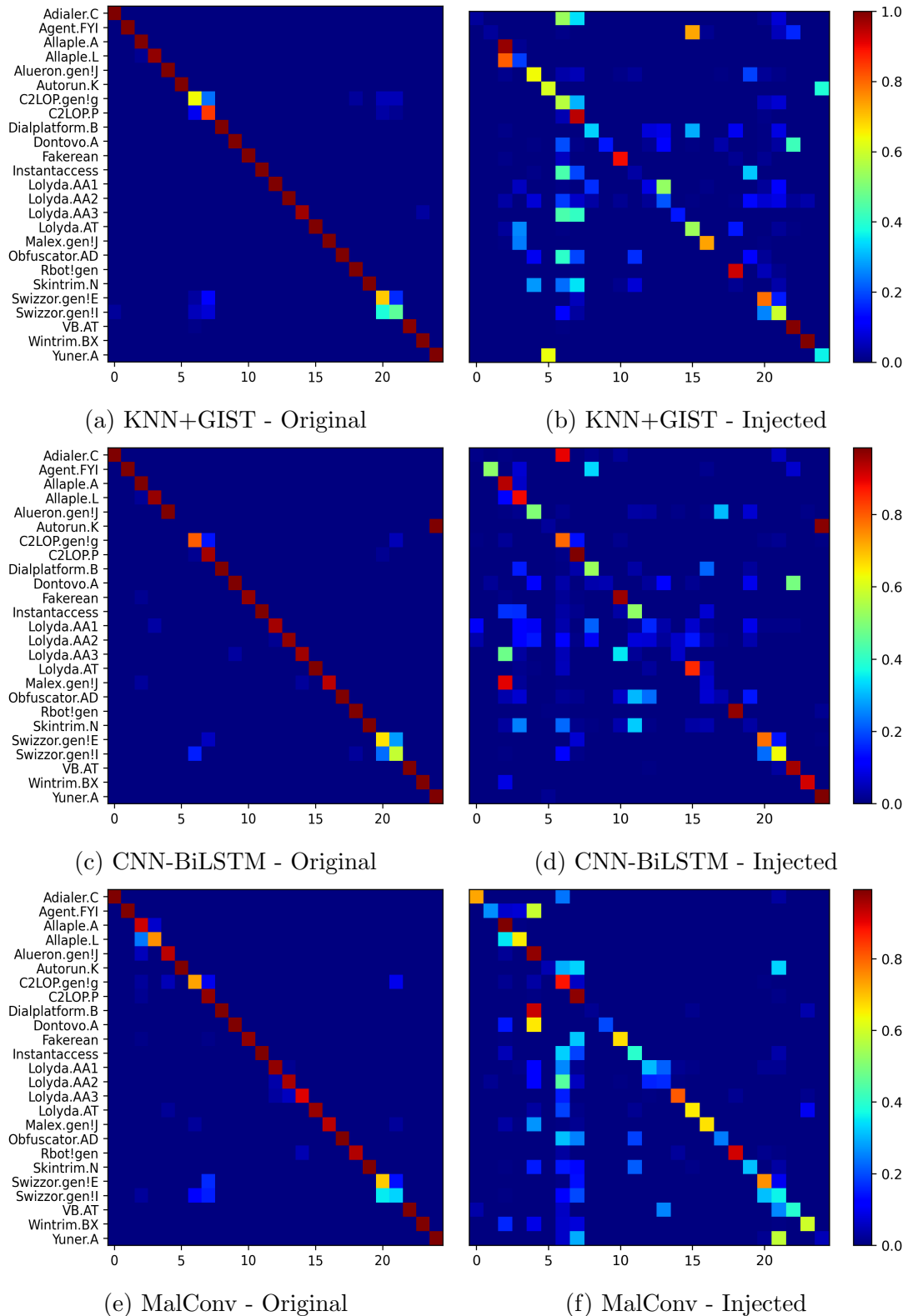


Figure 5.3: Confusion matrices for malware classification using KNN+GIST, Le-CNN-BiLSTM and MalConv in the original test set 5.3a,5.3c,5.3e and 5.3b,5.3d,5.3f when 5 sections of  $5 \times FileAlignment$  bytes are injected.

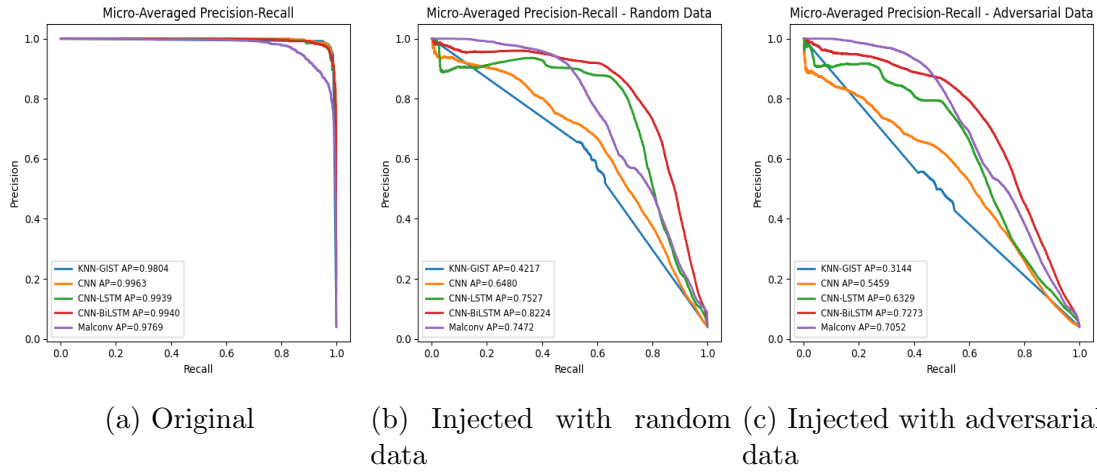


Figure 5.4: Precision Recall curves in the original test set 5.4a and when 5 sections of  $5 \times FileAlignment$  bytes are injected, 5.4b with random bytes and adversarial bytes in 5.4c. Each color represents a different model.

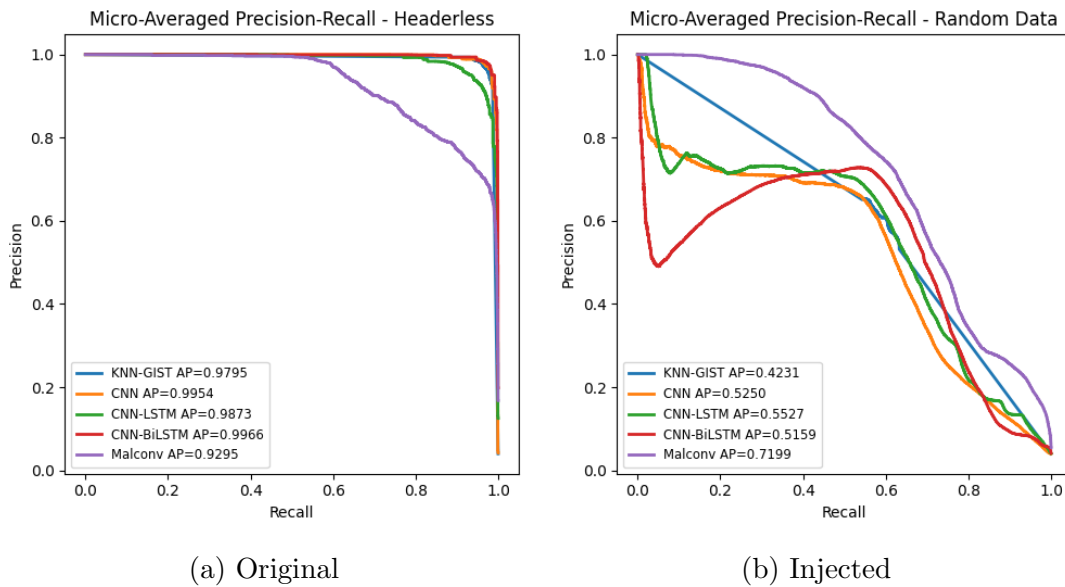


Figure 5.5: Precision Recall curves in the original test set 5.5a and 5.5b when 5 sections of  $5 \times FileAlignment$  bytes are injected, both versions without file header.

precision by 0.0026 in this scenario. All models became less robust to data injection, losing precision significantly when compared to complete executables in 5.4b. Despite that, MalConv is the only model with similar average precision to previous scenarios.

## 5.6 DEFENDING AGAINST DATA INJECTION

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

### 5.6.1 Augmentation

A solution proposed in the literature (CATAK et al., 2021; PEREZ; WANG, 2017; TAYLOR; NITSCHKE, 2018) is to augment the data used for training. Three strategies were initially evaluated:

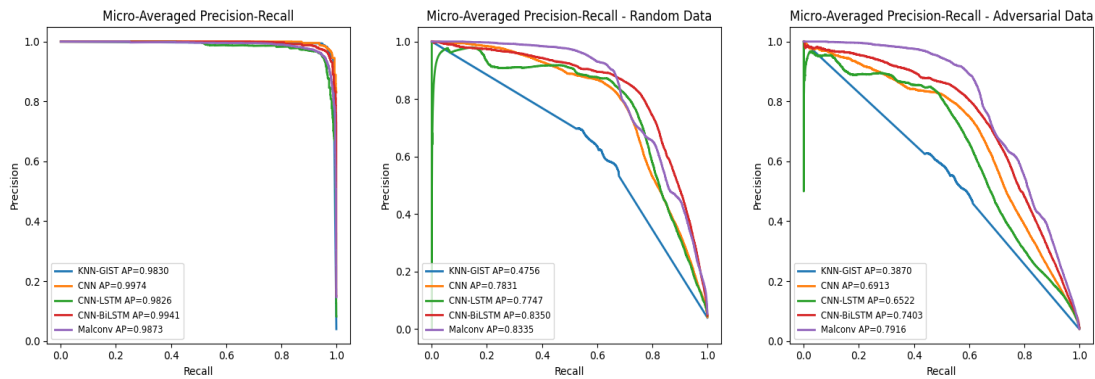
1. **Section reordering:** since our injection scheme adds new sections in a random position among the existing ones, the first augmentation idea was to reorder the sections on the training section. By doing this we wanted to check if the model could be more robust against data injection without seeing them during training. As shown by Figures 5.6a, 5.6b and 5.6c, this strategy increased a bit the performance of all models when compared to the vanilla results shown by Figure 5.4.
2. **Training with injected data:** since data is injected in the test set, a possibility was to include injected samples with random data in the training set as well. By comparing Figures 5.6d, 5.6e, 5.6f against Figure 5.4 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 Figures 5.6g, 5.6h, 5.6i, was also evaluated. Comparing with the original results in Figure 5.4 we can see that this may be a good defense strategy as well.

As shown by Figure 5.6, some models were improved by these augmentation strategies, even though they are still vulnerable.

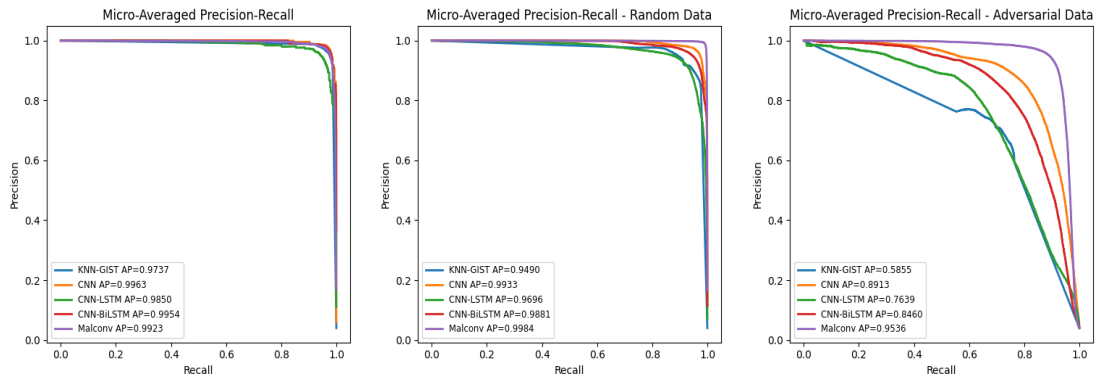
### 5.6.2 Binary Data

All experiments mentioned here were also performed in a binary dataset. We use Nataraj *et al.*'s entire dataset as a single malware class. We then collected 8,496 executables from a fresh Microsoft Windows 10 development environment (Microsoft Corporation, 2020) to form the benign class. We searched for all valid PE32 files and selected those with less than 5.5MB to approximate the malware class file characteristics. Benign files are used for training only, as our purpose is to investigate if an attacker can deceive a malware detector through data injection.

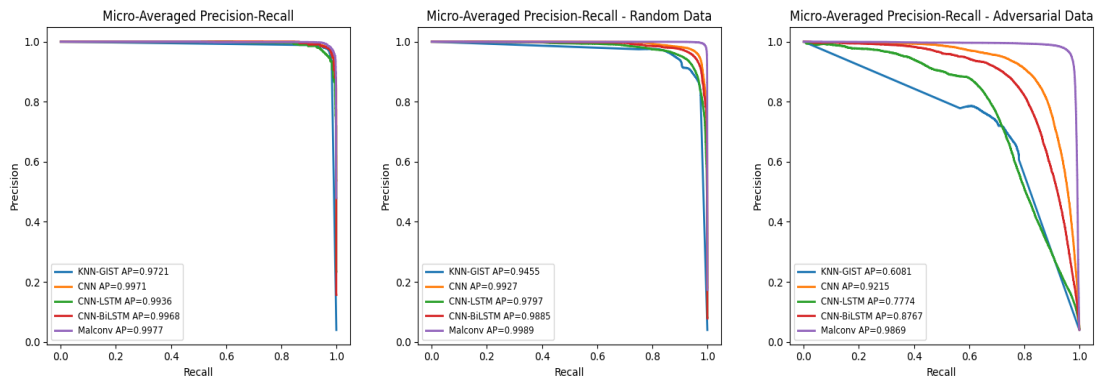
In this dataset the models were barely affected by data injection. We believe something similar to the mentioned by Raff *et al.* (RAFF et al., 2017) also happened in our



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

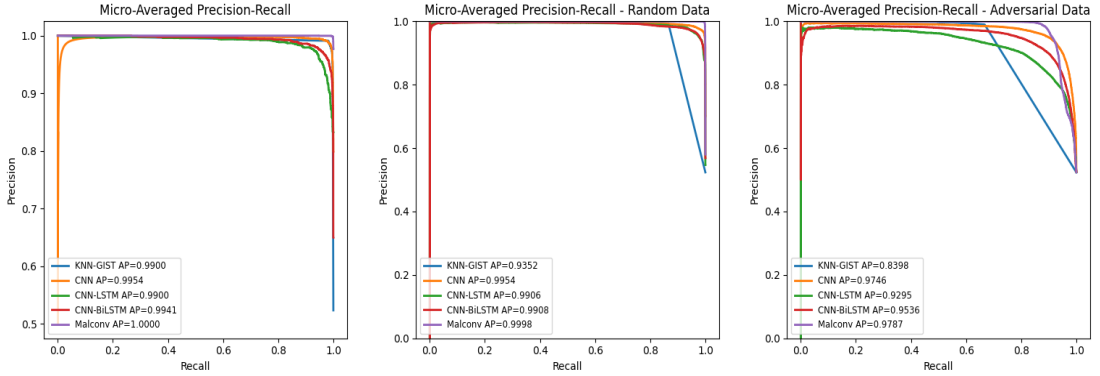


(d) Injection x Unmodified (e) Injection x Random (f) Injection x Adversarial



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

Figure 5.6: Precision Recall curves for tests with augmentation. 5.6a, 5.6d and 5.6g displays results on test sets with original samples. 5.6b, 5.6e and 5.6h displays results for datasets injected with random data. 5.6c, 5.6f and 5.6i display results for injection with adversarial data. 5 sections of  $5 \times FileAlignment$  bytes are injected in all cases. Each color represents a different model.



(a) Results on unmodified binary dataset (b) Results on random injected malware (c) Results on adversarial-injected malware

Figure 5.7: Precision Recall curves when 5 sections of  $5 \times FileAlignment$  bytes are injected in a binary dataset, 5.7b with random bytes and adversarial bytes in 5.7c. Each color represents a different model.

dataset: models were learning “Microsoft vs non-Microsoft” instead of “Benign vs Malign”, as shown by Figure 5.7. 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.

### 5.6.3 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 intra and extra families, packing and obfuscation of samples - but those are not the only concerns. Increasing architecture size does not necessarily lead to more robust models.

To verify that we follow Chen’s approach (CHEN, 2018; CHEN et al., 2020), which consists of fine-tuning a pre-trained CNN to classify malware families. We use the Inception-V3 architecture (SZEGEDY et al., 2016) pre-trained on the ImageNet dataset (DENG et al., 2009). For that, we resize our images to  $299 \times 299$  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, illustrated in Figure 5.8.

A straightforward observation in Figure 5.8a is that this model presents the same behavior as the previous one in Section 5.3: 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

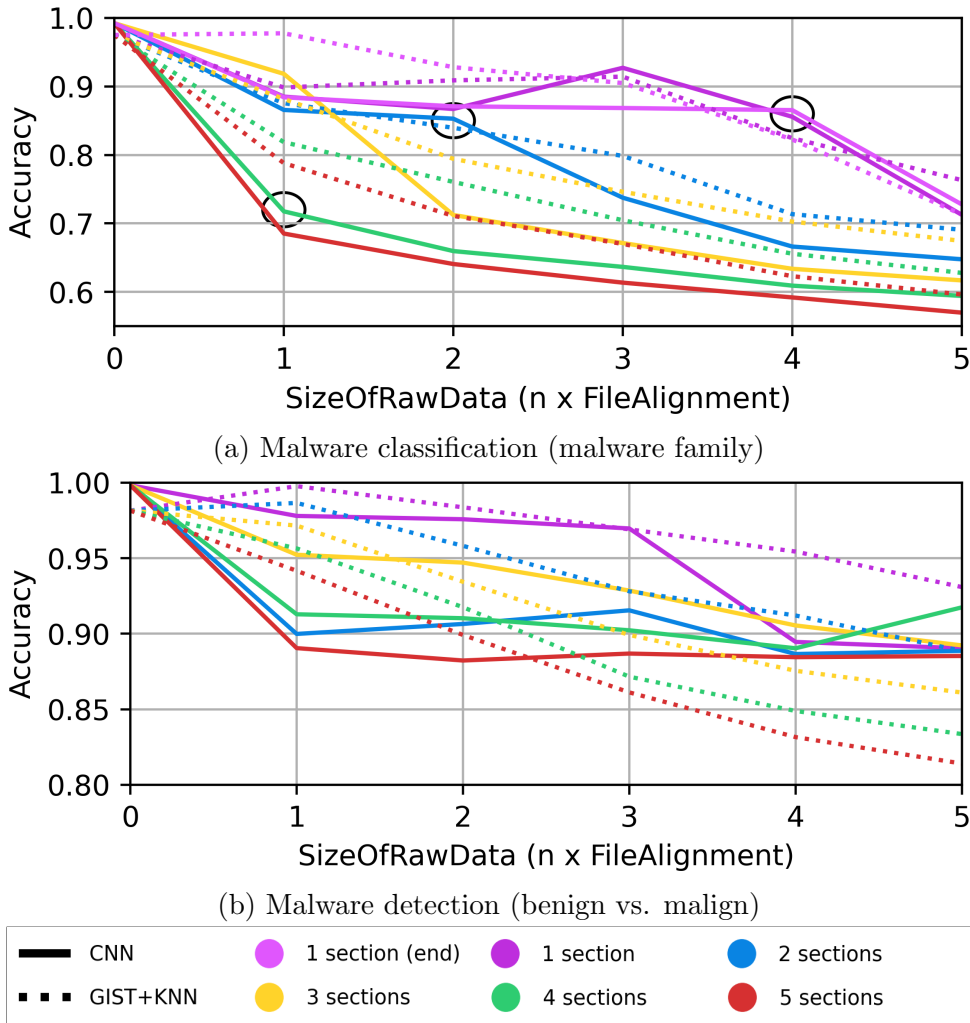


Figure 5.8: Average accuracy of 5.8a malware classification and 5.8b malware detection under different injection scenarios. Solid lines show results for Inception architecture, and dashed lines for GIST+KNN. Distinct colors represent the number of injected sections.

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 5.8a). 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 5.8b, we can also observe the same behavior described for the other malware detectors in Section 5.6.2. 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 (RAFF et al., 2017) with its Embedding layer and Le-CNN-BiLSTM (LE et al., 2018) 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.





## CONCLUSIONS AND FUTURE WORK

In this work we proposed a new method to inject data into malware files in order 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 (NATARAJ et al., 2011), MalConv (RAFF et al., 2017) and Le-CNN (LE et al., 2018) 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 literature (YUE, 2017; CHEN, 2018; SU et al., 2018; KHORMALI et al., 2019; CHEN et al., 2020; LE et al., 2018). This work shows a simple technique that is able to 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 (RAFF et al., 2017) or combining CNN with LSTM (LE et al., 2018) 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 were able to see 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) (KHORMALI et al., 2019), injecting FSGM generated sections in any position of the file.

- The classifiers in our experiments were heavily skewed towards using the header bytes as a feature. This area shares many relevant bytes across samples because of the PE file structure, making them a relevant and discriminative feature.
- Our attempt at crafting a reliable binary dataset was trickier than initially thought of. Despite still being affected by the injection mechanism, the classifiers lost way less performance than in the multiclass dataset. Further exploration using multiple sources for benign files is required to have better insights on which features are more discriminant for this scenario.

As mentioned in Section 5.6, 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 is kept. Current experiments point in the direction that instead of relying on a fixed preprocessing method - like truncating or interpolating - more dynamic approaches should be investigated, such as Attention based methods.

## BIBLIOGRAPHY

- 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. *CoRR*, abs/1801.00318, 2018. Disponível em: <http://arxiv.org/abs/1801.00318>.
- AL-DUJAILI, A. et al. Adversarial deep learning for robust detection of binary encoded malware. In: *2018 IEEE Security and Privacy Workshops (SPW)*. [S.l.: s.n.], 2018. p. 76–82.
- ANDERSON, H. S. et al. Evading machine learning malware detection. *black Hat*, v. 2017, 2017.
- ANDERSON, H. S.; ROTH, P. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.
- ATHIWARATKUN, B.; STOKES, J. W. Malware classification with lstm and gru language models and a character-level cnn. In: IEEE. *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. [S.l.], 2017. p. 2482–2486.
- BENKRAOUDA, H. et al. Attacks on visualization-based malware detection: Balancing effectiveness and executability. In: SPRINGER. *International Workshop on Deployable Machine Learning for Security Defense*. [S.l.], 2021. p. 107–131.
- CARLINI, N.; WAGNER, D. Towards evaluating the robustness of neural networks. In: *2017 IEEE Symposium on Security and Privacy (SP)*. [S.l.: s.n.], 2017. p. 39–57. ISSN 2375-1207.
- CATAK, F. O. et al. Data augmentation based malware detection using convolutional neural networks. *PeerJ Computer Science*, PeerJ Inc., v. 7, p. e346, 2021.
- CHEN, L. Deep transfer learning for static malware classification. *arXiv preprint arXiv:1812.07606*, 2018.
- CHEN, L. et al. Stamina: scalable deep learning approach for malware classification. *Intel White Paper*, v. 1, n. 2, p. 3, 2020.
- CLEMENS, J. Automatic classification of object code using machine learning. *Digital Investigation*, Elsevier BV, v. 14, p. S156–S162, aug 2015. Disponível em: <https://doi.org/10.1016/j.diin.2015.05.007>.

COULL, S. E.; GARDNER, C. Activation analysis of a byte-based deep neural network for malware classification. In: IEEE. *2019 IEEE Security and Privacy Workshops (SPW)*. [S.l.], 2019. p. 21–27.

DAUPHIN, Y. N. et al. *Language Modeling with Gated Convolutional Networks*. arXiv, 2016. Disponível em: [⟨https://arxiv.org/abs/1612.08083⟩](https://arxiv.org/abs/1612.08083).

DAVIS, J.; GOADRICH, M. The relationship between precision-recall and roc curves. In: *Proceedings of the 23rd international conference on Machine learning*. [S.l.: s.n.], 2006. p. 233–240.

DEMETRIO, L. et al. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583*, 2019.

DEMETRIO, L. et al. Adversarial examples: a survey and experimental evaluation of practical attacks on machine learning for windows malware detection. *ACM Transactions on Privacy and Security (TOPS)*, ACM New York, NY, USA, v. 24, n. 4, p. 1–31, 2021.

DENG, J. et al. ImageNet: A Large-Scale Hierarchical Image Database. In: *CVPR09*. [S.l.: s.n.], 2009.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016. [⟨http://www.deeplearningbook.org⟩](http://www.deeplearningbook.org).

GOODFELLOW, I. J.; SHLENS, J.; SZEGEDY, C. Explaining and harnessing adversarial examples. In: BENGIO, Y.; LECUN, Y. (Ed.). *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. [s.n.], 2015. Disponível em: [⟨http://arxiv.org/abs/1412.6572⟩](http://arxiv.org/abs/1412.6572).

GROSSE, K. et al. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

HADDADPAJOUH, H. et al. A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Generation Computer Systems*, Elsevier, v. 85, p. 88–96, 2018.

KHORMALI, A. et al. Copycat: practical adversarial attacks on visualization-based malware detection. *arXiv preprint arXiv:1909.09735*, 2019.

KOLOSNAJAJI, B. et al. Deep learning for classification of malware system call sequences. In: SPRINGER. *Australasian Joint Conference on Artificial Intelligence*. [S.l.], 2016. p. 137–149.

KOO, H.; POLYCHRONAKIS, M. Juggling the gadgets: Binary-level code randomization using instruction displacement. In: *Asia Conference on Computer and Communications Security (AsiaCCS)*. [S.l.: s.n.], 2016.

KRČÁL, M. et al. Deep convolutional malware classifiers can learn from raw executables and labels only. 2018.

LE, Q. et al. Deep learning at the shallow end: Malware classification for non-domain experts. *Digital Investigation*, v. 26, p. S118 – S126, 2018. ISSN 1742-2876. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1742287618302032>.

LI, Y. et al. Experimental study of machine learning based malware detection systems' practical utility. In: *HICSS SYMPOSIUM ON CYBERSECURITY BIG DATA ANALYTICS*. [S.l.: s.n.], 2020.

LIU, Y.-s. et al. A new learning approach to malware classification using discriminative feature extraction. *IEEE Access*, IEEE, v. 7, p. 13015–13023, 2019.

LUCAS, K. et al. Malware makeover: breaking ml-based static analysis by modifying executable bytes. In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. [S.l.: s.n.], 2021. p. 744–758.

Microsoft 365 Defender Threat Intelligence Team. *Microsoft researchers work with Intel Labs to explore new deep learning approaches for malware classification*. 2020. <https://www.microsoft.com/security/blog/2020/05/08/microsoft-researchers-work-with-intel-labs-to-explore-new-deep-learning-approaches-for-malware-classification/>, Last accessed on 2022-02-19.

Microsoft Corporation. *Microsoft Security Intelligence Report Volume 24*. [S.l.], 2019. Disponível em: <https://www.microsoft.com/security/blog/2019/02/28/microsoft-security-intelligence-report-volume-24-is-now-available/>.

Microsoft Corporation. *Windows 10 Development Environment*. 2020. <https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/>, Last accessed on 2020-06-19.

Microsoft Corporation. *Microsoft Digital Defense Report OCTOBER 2021*. [S.l.], 2021. Disponível em: <https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RWMMFi>.

MITNICK, K. D.; SIMON, W. L. *The Art of Deception: Controlling the Human Element of Security*. New York, NY, USA: John Wiley & Sons, Inc., 2003. ISBN 076454280X.

NATARAJ, L. et al. Malware images: Visualization and automatic classification. In: *Proceedings of the 8th International Symposium on Visualization for Cyber Security*. New York, NY, USA: ACM, 2011. (VizSec '11), p. 4:1–4:7. ISBN 978-1-4503-0679-9. Disponível em: <http://doi.acm.org/10.1145/2016904.2016908>.

OLIVA, A.; TORRALBA, A. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, Springer, v. 42, n. 3, p. 145–175, 2001.

PAPPAS, V.; POLYCHRONAKIS, M.; KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: *IEEE Symposium on Security and Privacy*. [S.l.: s.n.], 2012.

PASCANU, R. et al. Malware classification with recurrent networks. In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. [S.l.: s.n.], 2015. p. 1916–1920. ISSN 1520-6149.

PEREZ, L.; WANG, J. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

RAFF, E. et al. Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435*, 2017.

RAFF, E. et al. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, v. 14, n. 1, p. 1–20, Feb 2018. ISSN 2263-8733. Disponível em: <https://doi.org/10.1007/s11416-016-0283-1>.

RONEN, R. et al. *Microsoft Malware Classification Challenge*. 2018.

SAITO, T.; REHMSMEIER, M. The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets. *PLOS ONE*, Public Library of Science, v. 10, n. 3, p. 1–21, 03 2015. Disponível em: <https://doi.org/10.1371/journal.pone.0118432>.

SAXE, J.; BERLIN, K. Deep neural network based malware detection using two dimensional binary program features. In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. [S.l.: s.n.], 2015. p. 11–20.

SAXE, J.; SANDERS, H. *Malware Data Science*. 1st. ed. San Francisco, CA, USA: No Starch Press, 2018. ISBN 978-1-59327-859-5.

SIKORSKI, M.; HONIG, A. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st. ed. San Francisco, CA, USA: No Starch Press, 2012. ISBN 1593272901, 9781593272906.

SU, J. et al. Lightweight classification of iot malware based on image recognition. In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. [S.l.: s.n.], 2018. v. 02, p. 664–669. ISSN 0730-3157.

Symantec Corporation. *Internet Security Threat Report Volume 24*. [S.l.], 2019. Disponível em: <https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/istr-24-cyber-security-threat-landscape>.

SZEGEDY, C. et al. Rethinking the inception architecture for computer vision. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. [S.l.: s.n.], 2016. p. 2818–2826.

TAYLOR, L.; NITSCHKE, G. Improving deep learning with generic data augmentation. In: IEEE. *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*. [S.l.], 2018. p. 1542–1547.

YUE, S. Imbalanced malware images classification: a cnn based approach. *arXiv preprint arXiv:1708.08042*, 2017.