POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

DAMANI: Deep leArning Models for Antivirus mimicking and sigNature extractIon

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Nicole Filippi**

Student ID: 10719487
Advisor: Prof. Stefano Longari
Co-advisors: Michele Carminati, Gabriele Digregorio
Academic Year: 2024-2025

Abstract

Modern antivirus software heavily relies on static analysis, particularly signature-based detection, to efficiently identify malware. However, commercial antivirus engines typically operate as black-box systems, making their internal detection logic and specific signature patterns inaccessible for analysis. This thesis proposes DAMANI (Deep leArning Models for Antivirus mimicking and sigNature extractIon), a framework designed to approximate antivirus behavior and extract representative malicious byte patterns using deep learning techniques.

The framework operates in two stages. First, a supervised 2D-CNN is trained to mimic the decision boundaries of a target antivirus engine, achieving high accuracy in replicating both correct classifications and errors. Subsequently, a Convolutional Autoencoder is employed to generate sparse binary masks that isolate the specific byte sequences most responsible for triggering a malicious classification.

The framework is evaluated on three distinct antivirus engines: Microsoft Defender, ClamAV, and Avast Premium Security. Experimental results demonstrate that the extracted signatures are both effective, triggering detection when isolated from the original file, and necessary, as their removal significantly reduces the maliciousness score. A comparative analysis against ClamAV's open-source database confirms that the extracted patterns possess a high degree of similarity to real-world signatures used in production environments.

This framework provides deep insights into the behavior of antivirus engines, identifying the representative byte patterns that are essential for their detection logic. Ultimately, by characterizing the static detection patterns of black-box software, this research enables a comparative analysis of the decision logic used by different vendors.

Keywords: Malware Detection, Antivirus Mimicking, Deep Learning, Convolutional Neural Networks, Autoencoders, Signature Extraction

Abstract in lingua italiana

I software antivirus moderni si basano fortemente sull'analisi statica, in particolare sul rilevamento basato su firme, per identificare in modo efficiente i file malevoli. Tuttavia, gli antivirus commerciali operano tipicamente come sistemi black-box, rendendo inaccessibili all'analisi sia la logica interna di rilevamento sia gli specifici pattern di firma utilizzati. Questa tesi propone DAMANI (Deep leArning Models for Antivirus mimicking and sigNature extractIon), un framework progettato per replicare il comportamento degli antivirus ed estrarre pattern rappresentativi di file malevoli mediante tecniche di deep learning.

Il framework opera in due fasi: in primo luogo, una CNN bidimensionale supervisionata viene addestrata per imitare i criteri decisionali di un motore antivirus target, ottenendo un'accuratezza elevata nella replica sia delle classificazioni corrette sia degli errori e, successivamente, viene impiegato un autoencoder convoluzionale per generare maschere binarie sparse, che isolano le specifiche sequenze di byte maggiormente responsabili della classificazione malevola.

Il framework è stato valutato su tre distinti antivirus: Microsoft Defender, ClamAV e Avast Premium Security. I risultati sperimentali dimostrano che le firme estratte sono sia efficaci, poiché attivano il rilevamento anche quando isolate dal file originale, sia necessarie, in quanto la loro rimozione riduce significativamente il punteggio di malevolenza. Un'analisi comparativa con il database open-source di ClamAV conferma che i pattern estratti presentano un elevato grado di similarità con le firme utilizzate in ambienti reali.

Il framework fornisce una comprensione approfondita del comportamento degli antivirus, identificando i pattern di byte rappresentativi che risultano essenziali per la loro logica di rilevamento. Caratterizzando i pattern di rilevamento statico di software black-box, questa ricerca consente un'analisi comparativa della logica decisionale adottata da diversi vendor.

Parole chiave: Rilevamento di Malware, Emulazione di Antivirus, Deep Learning, Reti Neurali Convoluzionali, Autoencoder, Estrazione di Firme

Contents

1 | Introduction

Malware detection traditionally relies on signature-based mechanisms, where specific byte patterns or code fragments are used to identify known malicious software. Despite the evolution toward heuristic, behavior-based detection and the usage of machine learning, static signature-based analysis still plays a central role in many antivirus engines due to its efficiency and scalability. However, the exact nature of these signatures and the byte-level patterns effectively used in practice remain largely opaque, as commercial antivirus solutions do not expose their internal detection rules.

This thesis focuses on the problem of extracting representative static detection patterns from antivirus engines. The objective is to approximate and analyze the signature-like logic used by existing antivirus software. The core research question is whether it is possible to reconstruct, using deep-learning techniques, the byte sequences that most strongly influence an antivirus decision.

State-of-the-art machine learning and deep learning approaches for malware analysis primarily focus on classification performance [27, 50], robustness against adversarial attacks [74], or feature extraction techniques [12, 66]. These work demonstrate strong detection capabilities for certain architectures, such as Convolutional Neural Networks. However, these approaches aim at improving malware detection accuracy and do not extract and interpret the detection logic of real-world antivirus. Consequently, the problem of signature extraction from black-box antivirus engines remains mostly unexplored.

To address this gap, this thesis proposes DAMANI, a framework designed to approximate antivirus decision boundaries and extract meaningful byte-level detection patterns. The approach treats antivirus engines as black-box classifiers and leverages a deep neural network to mimic their behavior. Specifically, Portable Executable (PE) binaries are converted into fixed-size RGB images and then used to train a Two-Dimensional Convolutional Neural Network that reproduces the outputs of the antivirus. The objective of this stage is not to detect malware per se, but to replicate the decisions of the targeted engine. Once the mimicking model is trained, a Convolutional Autoencoder is employed to identify the minimal regions of the input representation that are sufficient to preserve the

malicious classification. By optimizing a mask under multiple constraints, the model isolates compact byte sequences that act as signature-like patterns. These extracted regions are then post-processed, aggregated, and analyzed across malware families and across different antivirus engines.

Experimental validation demonstrates that the extracted signatures are linked to the mimicked antivirus decisions. Malware samples containing only the extracted regions remain classified as malicious, while removing those regions significantly decreases the maliciousness score. Additional cross-engine analysis highlights both shared and engine-specific patterns, suggesting partial convergence in static detection strategies.

The main contributions of this thesis can be summarized as follows:

- DAMANI, a framework that offers:
  - An antivirus mimicking model capable of approximating antivirus decision logic using deep learning;
  - An autoencoder-based mask optimization strategy to isolate compact byte-level regions responsible for malicious classification.
- An analysis on information extracted by DAMANI on three well-known antivirus engines:
  - A cross-antivirus comparative methodology to analyze common static detection patterns;
  - An experimental validation demonstrating the relevance and reliability of the extracted signatures.

2 | Background

In this chapter, we provide an overview of some pivotal topics in the thesis. We introduce concepts such as malware, antivirus software, machine learning, deep learning, and deep learning algorithms, such as convolutional neural networks, and autoencoders.

2.1. Malware

Malware are pieces of software that are designed to damage systems, gain unauthorized access to computers, steal data, spy, and other malicious actions, when executed on a computer. Over the years, they evolved into complex and adaptive programs, and are able to evade basic security mechanisms [24].

Malware are divided in categories with respect to what they do on the host system [20, 24]. Among the most common malware families, the five that are used in this work are:

- **Viruses:** these software are "attached" to a non-malicious file, and their objective is to spread fast on various systems, in order to damage them.
- **Worms:** a more advanced version of viruses, with the key difference being that a worm is a standalone program. Their main characteristic is the high speed of diffusion among devices.
- **Trojans:** as the name suggests, they are programs that pretend to be something you may want to download, like a free game, a system update, or a "cracked" version of expensive software.
- **Backdoors:** these are malware that are installed usually by other malware, such as trojans or worms, in order to gain secretly access to the system.
- **Scarewares:** these are more of a "social attack", since they rely on social engineering techniques [24] to manipulate users through fear-inducing messages. They arrive on a computer, for example, as an urgent email or pop-up, presenting false or exaggerated security warnings, such as claims of system infections or critical vulnerabilities, in order to pressure the user into taking specific actions. For example, they

could trick people to delete some useful system files, thinking they are malicious, just to make them brake their own system, or they could pressure the user into downloading additional software or provide sensitive information.

Each family is characterized by different infection vectors and different malicious goals. Malware of the same family share common structural, behavioral and functional characteristics. These similarities arise from employment of the same attack technique, the frequent code reuse, and the automated malware generation, frequently used by hackers [19]. For this reason, malware detection systems usually rely on the identification of recurring patterns, such as signatures, that are known pieces of malicious code [18]. Since malware detection methods evolved, also malware authors needed to start using more complex techniques to evade detection [24, 52], including:

- Polymorphism: the malware modifies its code structure across different instances, preserving the functionality while producing distinct variants of the same malware;
- Packing: the malware payload is compressed or encrypted when not running, reducing the effectiveness of static analysis;
- Obfuscation: the code of the malware is transformed to make it more difficult to analyze or understand, for example altering control flow, renaming symbols, and inserting NOP (No Operation) instructions.

2.2. Portable Executables

Different operating systems rely on distinct executable formats, such as ELF on Unix systems, Mach-O on macOS, and Portable Executable (PE) on Microsoft Windows. Since Windows is the most used operating system on personal computers [3, 48], it remains a primary target for attackers [56]. For example, in a Surfshark Antivirus report of 2025, it is shown that of the 479K malware cases recorded, the 87% (419K) were on Windows [4]. For this reason, in this thesis we focus on the PE format, predominant in the Windows ecosystem. The PE format defines the structure of executable files, including headers, sections, and metadata required by the operating system loader. From a security perspective, the PE format provides a rich source of features for static analysis: detectors can inspect file structure, metadata, and not just byte-level patterns [56]. The PE format, however, also introduces a large attack surface: malware authors can manipulate headers, reorder sections, inject padding bytes, or modify non-essential fields without affecting program execution [48]. This dual nature makes PE files a central object of study in malware detection research [48].

2.3. Antivirus

An antivirus is a software designed to detect and prevent malware infections on different kind of devices, such as personal computers. There are two main types of analysis that can be done by an antivirus: static and dynamic [24, 58].

Static analysis is done without executing the program, and includes techniques such as signature-based detection, heuristic analysis, and code reachability. This kind of analysis is faster, but weaker in detecting malicious code that has been obfuscated or new types of malware (zero-days) [24].

Dynamic analysis' most used techniques are behavioral monitoring and sandboxing, much slower and resource-consuming than static ones, because the program needs to be executed, but this kind of analysis is more precise [24].

Traditionally, antivirus operate primally through static analysis. However, modern and advanced engines combine both static and dynamic detection techniques, becoming hybrid systems [24]. To balance accuracy and performance, usually antivirus use dynamic analysis only in certain situations, for example when static analysis' result is uncertain, or the user requests explicitly the analysis of a specific file [18].

In this thesis, an important concept is the signature-based detection. This kind of detection is a foundational element of antivirus systems, due to its efficiency [18]. A signature is a pattern associated with a known malware, and can be a sequence of malicious bytes, a structural characteristic of a file, a rule that describes known properties of malicious code (such as the YARA rules [9]). The signature-based detection has low computational cost and high precision for known threats, since every signature can be searched rapidly across large volume of files [11, 24]. However, this approach has some well-known limitations: signatures are sensitive to syntactic changes, even a small modification to a malware sample may invalidate the contained signature, making it undetectable. In malware detection, robustness is particularly important due to the adversarial nature of the domain, since malware samples can often be syntactically altered without affecting their malicious behavior [26]. As a result, a detection system may achieve high accuracy on a certain dataset, while remaining fragile to minor input changes.

To improve the generality of our analysis, this thesis considers three antivirus engines: Microsoft Defender [55], ClamAV [22], and Avast Premium Security [14]. These systems were selected to represent different deployment contexts within the antivirus ecosystem.

- **Microsoft Defender** is a widely deployed antivirus solution, since it is integrated

into the widespread Windows operating system, for free. Is a black-box antivirus, therefore its internal detection mechanisms are not accessible for direct inspection, the analysis is limited to observable inputs and outputs. It represents a modern protection platform that combines not only static and dynamic analysis, but also machine learning and cloud-based intelligence [54].

- **ClamAV** is an open-source antivirus engine based mainly on signature matching [18]. It was developed for Unix operating systems, and [23] shows how Unix-based systems are the majority among those regularly tested and supported. Due to its transparency, it provides a useful reference point for studying static signature-based detection mechanisms.

- **Avast Premium Security** is a commercial antivirus solution that employs an hybrid detection approach, combining both static and dynamic analysis, and machine learning [13]. It was selected as a representative commercial antivirus because its engine is deployed across multiple other products (such as Norton and AVG [18], all owned by the Gen Digital Inc. company [28, 29]). As a result, observations made on Avast are likely to reflect properties of a wider class of commercial antivirus engines rather than a single product.

By considering multiple antivirus engines, in this thesis we reduce vendor-specific bias and focus on general properties of detection systems rather than the behavior of individual products.

2.4. Machine Learning

Machine learning is a subfield of artificial intelligence centered on the development of mathematical models and algorithms that allow systems to learn patterns from data and consequentially make autonomous decisions. These models optimize their internal parameters (denoted as *weights*) to improve performance on a given metric by iteratively processing training data [32]. This capability allows for the processing of high-dimensional data (where each dimension is denoted as *feature*) where traditional algorithmic approaches would be computationally or logically infeasible. There are different learning paradigms, depending on the kind of samples used for training and the goal of the model [32, 35]:

- **Supervised learning:** relies on labeled datasets to learn how to produce the correct expected outcome. This family of models is employed in problems such as:
  - **Classification:** assign a label (class) to each sample in the dataset;

  - **Regression:** assign a value, or a set of values, to each sample in the dataset.

- **Unsupervised learning:** learns from unlabeled data to find regularities in samples with similar characteristics;
- **Reinforcement learning:** focuses on learning optimal decision-making strategies through interaction with an environment.

In this thesis, we used both classification and unsupervised learning.

To train and evaluate machine learning models, datasets are typically divided into distinct subsets with specific roles [32]:

- **Training set:** this set is used to learn the model parameters, by optimizing the chosen objective function;
- **Validation set:** this set is employed during development to tune hyperparameters and to monitor model behavior, in order to improve generalization;
- **Test set:** this set is reserved for the final evaluation and provides an unbiased estimate of the model's performance on previously unseen data.

When a set of samples is provided, it needs to be divided within these three sets. Usually 70% of the samples are used in the training set, 15% in the validation set, and 15% in the test set. This separation ensures a reliable evaluation of the model's ability to generalize to unseen real-world data.

2.5. Deep Learning

Deep learning is a subfield of machine learning based on artificial neural networks [32]. Deep learning models are characterized by the presence of multiple layers of nonlinear transformations, which allow the network to progressively extract higher-level and more abstract features from the input data [45]. Lower layers typically capture simple, local patterns, while deeper layers combine these patterns into more complex representations. Unlike machine learning models, deep learning ones autonomously extract features directly from data.

2.5.1. Artificial Neural Networks

Artificial Neural Networks (ANNs) are a class of machine learning models inspired by the structure of the human brain. They are composed of interconnected computational units, called neurons, that are organized into multiple layers. Each neuron computes a weighted

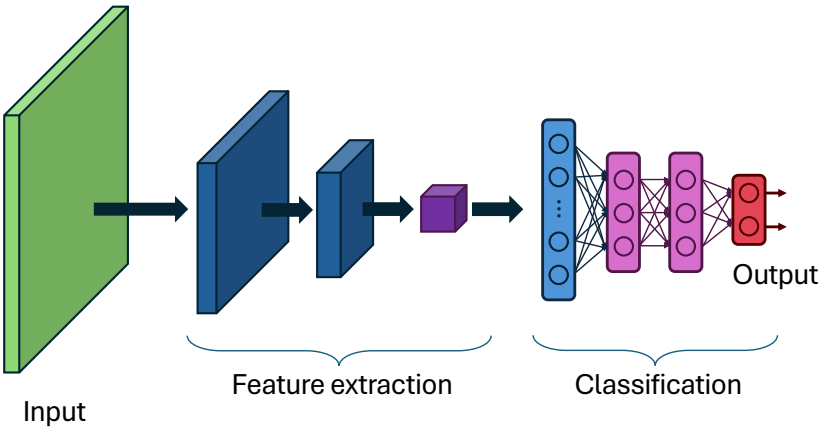


Figure 2.1: 2D-CNN typical structure

sum of its inputs and produces an output by applying a nonlinear activation function. The output of each layer is propagated as input to the subsequent layer, allowing the network to learn increasingly complex representations of the data. Neural networks are designed to approximate a target function by adjusting their weights during the training process [36].

Training consists of minimizing a loss function that quantifies the discrepancy between the predicted outputs and the corresponding target values (called *labels*). This loss serves as a measure of how well the network fits the training data. The optimization of the loss function is typically performed using gradient-based methods, such as gradient descent, which iteratively update the network parameters by moving in the direction that minimizes the loss function [61].

A major drawback of neural networks is their limited explainability. In many cases, the internal decision-making process of a trained model is not directly interpretable: given an output prediction, it is often difficult to determine how specific input features contributed to that result. This lack of transparency makes neural networks commonly referred to as black-box models [33].

To mitigate this limitation, some explainability techniques have been proposed. Among them, Gradient-weighted Class Activation Mapping (Grad-CAM) [63] provides visual explanations by highlighting the regions of the input that most strongly influence a model's prediction.

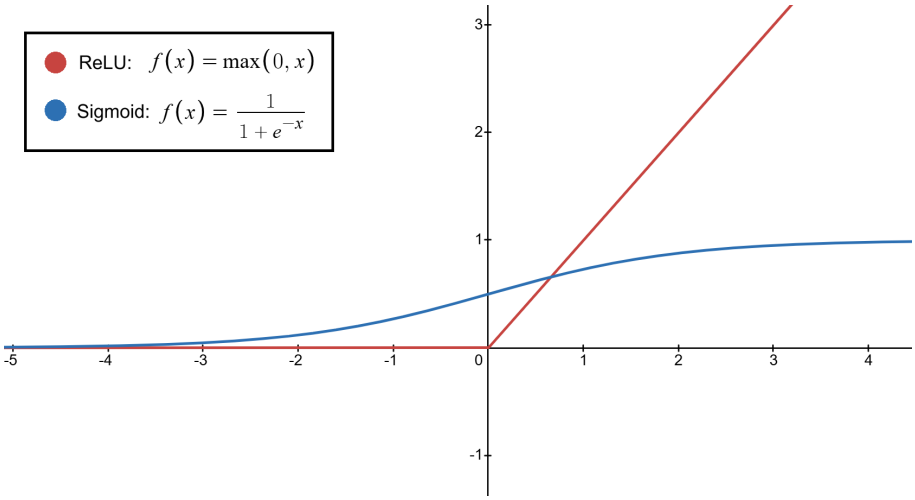


Figure 2.2: Common activation functions

Two-Dimensional Convolutional Neural Networks

Two-Dimensional Convolutional Neural Networks (2D-CNNs) are supervised deep learning models designed to process structured grid data, such as images, by exploiting spatial relationships between input elements [44]. For this reason, they have become the standard architecture for image-based learning tasks. Unlike fully connected neural networks, 2D-CNNs preserve spatial information through convolutional layers. These layers apply learnable filters that slide across the input, generating feature maps that capture local patterns while sharing weights across spatial locations [43]. This design significantly reduces the number of trainable parameters and improves computational efficiency. A visual representation of a 2D-CNN is provided in Figure 2.1.

Feature extraction in CNNs follows a hierarchical process: early layers learn low-level patterns, while deeper layers progressively combine them into higher-level representations. Pooling layers are commonly interleaved with convolutional layers to reduce spatial dimensionality while retaining the most informative features [75]. For example, max-pooling aggregates local regions by selecting the maximum activation value. After the convolutional and pooling stages, the resulting feature representations are processed by fully connected layers, which perform the final classification task.

Non-linear activation functions are essential to model expressiveness. Some commonly used activation functions are ReLU (Rectified Linear Unit) [31] and Sigmoid, that can be seen in Figure 2.2. ReLU is widely used in hidden layers due to its simplicity and effectiveness, while Sigmoid is commonly employed in the final layer of binary classification tasks to produce outputs in $[0, 1]$.

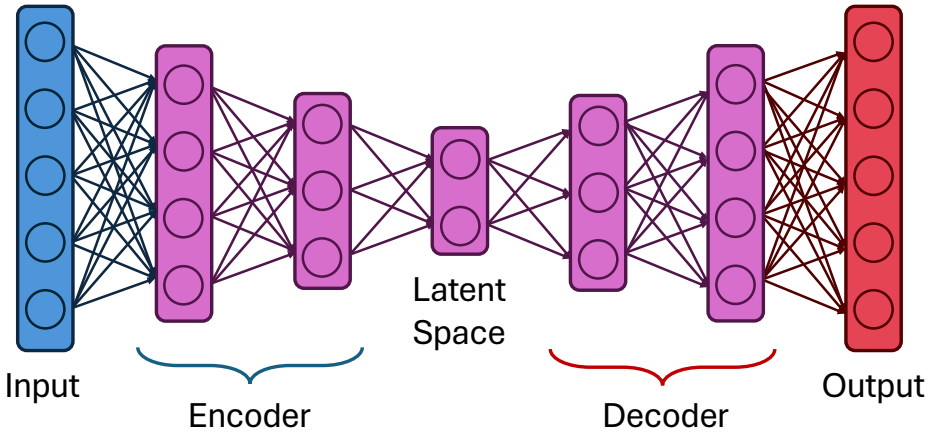


Figure 2.3: Standard autoencoder structure

CNN performance can be further improved through transfer learning, where models that have been pre-trained on large datasets are adapted to specific tasks [57]. This approach uses the generality of learned representations and reduces training requirements.

Autoencoders

An autoencoder is a specific class of neural network designed to learn efficient representations of data in an unsupervised way [38]. A visual representation of a 2D-CNN is provided in Figure 2.3. The architecture of an autoencoder is characterized by its symmetrical structure, that consists of two components:

- **Encoder:** this module maps the high-dimensional input data into a lower-dimensional representation: the latent space. During this phase, the network is forced to prioritize the most relevant features of the data, while discarding redundant information or noise;
- **Decoder:** this module tries to map the latent representation back into the original input space. The objective is to produce a reconstruction that is as close as possible to the original input.

The performance of an autoencoder is measured by a reconstruction loss function, which calculates the difference between the original input and the reconstructed output. Since the system uses the input itself as the target for learning, it does not require external labels.

A widely used variant is the Convolutional Autoencoder (CAE), which replaces fully connected layers with convolutional and deconvolutional layers [51]. Convolutional autoencoders are particularly effective when the input data has a spatial structure, such as images or two-dimensional representations of binary data. By using convolutional operations and weight sharing, CAEs preserve spatial locality and learn more robust and meaningful feature representations. In this thesis, we used a Convolutional autoencoder that is also Cross-Domain (or Domain Translator) [40]. A Cross-Domain autoencoder learns to extract the underlying features from one domain and "re-renders" them in the format of another domain. In this setup, the latent space behaves as a bridge between the two domains: the encoder is trained to map the data from *Domain A* to the latent space, and the decoder is trained to generate an output in *Domain B* from the latent representation.

3 | Motivation

This chapter motivates the need for a systematic analysis of antivirus detection behavior. It introduces the problem addressed in this thesis, reviews related research in this field, and outlines the goals and challenges of the proposed approach.

3.1. Problem Statement

Modern antivirus software relies on a combination of static and dynamic analysis techniques to detect malicious software [15]. Among these, static detection mechanisms often employ signature-based approaches to efficiently identify known malware pattern [21]. While effective, such mechanisms may be sensitive to syntactic variations in the binaries.

In this thesis we investigate whether it is possible to approximate or extract representative malicious sequences of bytes (called *signatures*) used by different antivirus software for detection. First, we leverage a deep learning model to mimic the antivirus behavior. Then, we employ another deep learning model to extract the patterns of bytes seen as malicious instructions. Finally, we perform studies on the extracted bytes, analyzing whether there are common patterns used by different antivirus engines.

Understanding these patterns is important for several reasons: first of all, it allows researchers to assess the robustness of static detection mechanisms against minor syntactic transformations, and secondly, it provides insights about different antivirus engines that rely on similar detection logic. If detection depends on fragile patterns and the author of the malware has the ability to extract these patterns even partially, small modifications to malicious software can be applied to disrupt such patterns, potentially leading to false negatives and posing security risks [11]. For this reason, studying the characteristics of antivirus detection patterns is essential to evaluate the resilience of current malware detection approaches. Moreover, a comparative analysis across multiple antivirus engines enables a deeper understanding of similarities among different detection techniques. Lastly, this work aims to support the development of more robust antivirus systems.

3.2. State of the Art

The application of machine learning and deep learning techniques to malware detection has been widely explored in recent years. Existing research can be categorized according to the type of learning models adopted.

3.2.1. Machine Learning and Deep Learning approaches

Early studies such as *Static Malware Detection & Subterfuge* [27] and *Analyzing Machine Learning Algorithms for Antivirus Applications* [50] investigate the effectiveness of machine learning models in comparison to traditional antivirus engines. These works show that learning-based classifiers can outperform signature-based antivirus solutions, particularly in adversarial settings. However, they primarily focus on detection accuracy and do not provide insights into the internal decision patterns learned by the models or their relationship with antivirus detection mechanisms.

Several works rely on handcrafted static features extracted from malware samples. For instance, *An Android Malware Detection Approach Based on Static Feature Analysis* [65] proposes the use of API call sequences combined with classical classifiers such as Support Vector Machines (SVM), k-Nearest Neighbors (KNN), and Naive Bayes (NB). These approaches highlight the feasibility of static analysis without executing the sample, but they heavily depend on domain-specific feature engineering and often struggle to generalize across platforms or malware families. Similarly, *Machine Learning With Feature Selection Using Principal Component Analysis for Malware Detection: A Case Study* [73] explores the use of metadata-based features combined with Principal Component Analysis (PCA) and Artificial Neural Networks (ANNs). While dimensionality reduction improves computational efficiency and classification performance, the approach remains constrained by the quality and completeness of manually selected features.

More recent studies use deep learning architectures to overcome the limitations of feature engineering. *Malware Analysis and Detection Using Machine Learning Algorithms* [10] demonstrates that among various learning models, Convolutional Neural Networks (CNNs) achieve superior performance by automatically learning useful features from data. CNN-based approaches have been successfully applied to binary representations, opcode sequences, and image-like encodings of executables, showing strong detection capabilities. For example, *Malware Detection by Eating a Whole EXE* [59] operates directly on executable binaries and demonstrates that neural networks can be trained on raw byte sequences achieving competitive performance while preserving complete binary infor-

mation. Other approaches based on One-Dimensional Convolutional Neural Networks (1D-CNNs) further confirm that binary-level analysis can yield highly accurate and efficient classifiers for distinguishing malicious and benign samples [64, 69]. Another prominent research direction involves transforming malware binaries into image representations. By mapping binary values into two-dimensional pixel matrices, malware samples can be treated as images and analyzed using Two-Dimensional Convolutional Neural Networks (2D-CNNs) [34]. Survey studies highlight that this representation enables the application of transfer learning, achieving classification accuracies exceeding 98% on benchmark datasets [17]. Works such as *Malware Detection with Malware Images Using Deep Learning Techniques* [37] show also that CNN-based models are more robust than recurrent architectures against adversarial manipulations.

Other deep learning architectures have also been explored in the literature. For instance, *Malware Classification on Malimg Using MobileNet and LSTM for Efficient Detection* [16] integrates the computational efficiency of MobileNet with Long Short-Term Memory (LSTM) [39] networks. Sequence-based methods have also been applied in specialized contexts such as ransomware detection, where LSTM networks are combined with file entropy analysis [42].

Additional studies investigate the use of ANNs and Transformer-based models coupled with binary data analysis [62], or autoencoder-based approaches [71], used to model benign and malicious distributions by minimizing reconstruction error. By using grayscale image representations and dimensionality reduction, autoencoders have demonstrated promising results.

In the end, works such as *Machine Learning Aided Malware Detection for Secure and Smart Manufacturing* [60] and *Comparison of Malware Detection Techniques Using Machine Learning Algorithms* [67] confirm the effectiveness of deep learning models across different application domains.

It is noteworthy that all these studies are primarily focused on detection performance rather than interpretability or cross-system analysis.

3.2.2. Adversarial approaches

Another different branch of research investigates adversarial techniques to evade both machine learning-based detectors and antivirus systems. *Practical Attacks on Machine Learning: A Case Study on Adversarial Windows Malware* [25] demonstrates that Portable Executable (PE) files can be adversarially manipulated while preserving their malicious functionality. *Tarallo: Evading Behavioral Malware Detectors in the Problem Space* [26]

shows that adversarial modifications can be performed without alterations to the malware logic, highlighting that even comprehensive detection systems can be evaded through carefully designed transformations.

Several studies focus specifically on generating adversarial samples against deep learning-based detectors: *GAMBD: Generating Adversarial Malware against MalConv* [47] and *MalPatch: Evading DNN-Based Malware Detection with Adversarial Patches* [72] target convolutional models by introducing perturbations that exploit architectural weaknesses.

In parallel, research has explored robustness-oriented defences: *Enhanced DNNs for Malware Classification with GAN-Based Adversarial Training* [74] proposes the use of Generative Adversarial Networks (GANs) to model adversarial perturbations as noise added to raw bit sequences. By filtering such perturbations during training, the proposed approach improves classifier resilience against adversarial attacks.

3.2.3. Static characteristics approaches

Beyond machine learning and deep learning methods, several works explore malware detection through information-theoretic and structural properties of executable files. *Comparing Files Using Structural Entropy* [66] proposes the use of entropy profiles to characterize the internal organization of binaries, enabling similarity comparisons without relying on explicit signatures. By modeling how information is distributed across file sections, this approach captures global structural patterns. Similarly, in *Detecting Malware with Information Complexity* [12] malware's information complexity is measured to distinguish malicious from benign software. Rather than focusing on specific instructions or API calls, this line of research assumes that malware exhibits distinctive complexity patterns due to packing, obfuscation, or encryption, making these metrics effective indicators of malicious behaviour.

While these approaches capture high-level structural properties, they rely on handcrafted metrics and cannot extract detection patterns.

3.2.4. Learning-based antivirus mimicking

Mimicking Anti-Viruses with Machine Learning and Entropy Profiles [53] explicitly investigates the feasibility of modeling antivirus detection behaviour using machine learning. By learning entropy-based representations extracted using antivirus decisions, this work demonstrates that it is possible to approximate antivirus classification boundaries without accessing internal signatures. This approach is more aligned with the objectives of this

thesis, as it shifts the focus from malware detection alone to the analysis and imitation of antivirus detection mechanisms. However, the mimicked model is employed to search for misclassified malware, without extracting useful information from the antivirus.

The work that mostly inspired our thesis is presented in the thesis by Di Gloria [30], which investigates the feasibility of mimicking antivirus detection behaviour using a 2D-CNN. In this approach, a the CNN is trained to mimic an antivirus engine treated as a black box. An autoencoder is then employed to generate synthetic samples that are classified as false positives, and in the end these samples are used to iteratively retrain the classifier as in a GAN. While this work demonstrates the effectiveness of combining supervised and unsupervised learning to improve detection robustness, its primary focus lies in attacking the antivirus with false positives and then enhancing classification performance of the model. In contrast, our thesis still uses a model for antivirus mimicking, but with a different objective: rather than generating synthetic samples to refine the classifier, the autoencoder is used to identify and isolate the most informative or malicious regions of the input representation. These extracted patterns are then analyzed across multiple antivirus engines to study similarities and differences in their detection behaviour.

3.2.5. Limitations of existing approaches

Despite the substantial progress in machine learning-based malware detection, most existing works share common limitations. First, the majority of studies aim at maximizing classification accuracy, treating antivirus engines only as oracles, and not as subjects of analysis. Second, detection models are typically trained and evaluated against a single labelling source, without considering differences across multiple antivirus engines. Finally, limited attention has been given to understanding or extracting detection patterns, especially in a comparative cross-engine setting.

Similarly, since deep learning-based malware detection systems are still vulnerable to adversarial attacks, research is still focused on robust learning strategies. Already existing deep learning approaches work well at malware classification, but they primarily focus on detection accuracy, while rarely addressing the interpretability of learned features or their relationship with antivirus detection mechanisms. Even in adversarial research, the primary focus is on attack generation and improving classifier robustness, rather than on analyzing model outputs.

This research gap further motivates the need for approaches that aim to model and interpret antivirus detection behaviour, rather than only optimizing evasion or classification performance.

3.3. Goals and Challenges

3.3.1. Goals

The primary goal of this thesis is to fill a gap in existing research by investigating whether black-box antivirus software can be characterized by representative static detection patterns, using deep learning techniques. Specifically, we aim to:

- Model antivirus detection behavior using a 2D-CNN;
- Approximate detection patterns using an autoencoder;
- Do the same for more antivirus engines;
- Identify common or recurring patterns;

3.3.2. Challenges

Addressing these goals is inherently challenging due to the opaque nature of modern antivirus software. The majority of antivirus engines are closed-source systems that combine multiple detection techniques, including signatures, heuristics, machine learning models, code reachability analysis, and dynamic analysis. As a result, the exact contribution of each component to a detection decision is unknown.

Furthermore, antivirus products operate as black-box systems, exposing only input-output behavior. This requires extensive experiments to validate whether the extracted patterns can meaningfully reflect the detection mechanisms.

Finally, the results evaluation across multiple antivirus engines ensures the validity and generality of the framework. This is challenging due to the heterogeneous behavior of different antivirus products, which required distinct procedures for data labeling, testing, and result collection. In practice, interacting with multiple antivirus engines involved implementing separate workflows and evaluation pipelines tailored to each system, significantly increasing the complexity of the experimental setup.

4 | Approach

This chapter describes the methodological approach adopted in this thesis to extract, analyze and compare antivirus detection patterns. Starting from the problem formulation introduced in the previous chapter, we present the overall pipeline used to approximate antivirus behavior and to extract representative malicious patterns from PE files. Here, we focus on the design choice and the logic behind each component, while the implementation is discussed in the next chapter.

4.1. Approach Overview

In this thesis we proposes DAMANI, a framework to extract static detection patterns used by an antivirus. The core idea is to treat antivirus engines as black-box classifiers and approximate their behavior using a supervised deep-learning model, and then use an unsupervised deep-learning model to extract from the mimicking model the most malicious byte sequences. A Two-Dimensional Convolutional Neural Network (2D-CNN) is trained on antivirus outputs to mimic its detection decisions, using both malicious and benign PE binaries converted into RGB images.

Given the trained model, an autoencoder is then employed to identify and extract the byte-level regions (denoted as *signatures* for brevity) that most strongly influence the malicious classification of the 2D-CNN, and consequently of the mimicked antivirus. The autoencoder is trained on malware belonging to a single family, to better overfit the results on the characteristics of that specific malware family.

We apply this framework on three antivirus, and for each one of them we extract malicious byte sequences of five different malware families. After collecting all the extracted malicious signatures, we analyze and compare them to investigate commonalities in the static detection logic of different antivirus software and extract some statistics.

The proposed approach is structured into the following logical steps:

- **Step 1 - Antivirus Selection & Dataset Collection**
  We carefully select three different antivirus engines to represent different detection

technologies and design philosophies, ensuring a broad evaluation of current industry standards. We collect malware software from two online public repositories: MalwareBazaar and VirusShare, while goodware software were collected from Softonic, a popular software distribution platform. We select from these collections only 32-bit PE files and removed any duplicates.

- **Step 2 - Labels Extraction**
  The filtered 32-bit PE files are subjected to scanning by each selected antivirus engine, to obtain ground-truth detection labels. These labels represent the specific classification logic of each antivirus, and are essential for supervising the training of our mimicking model, allowing it to learn the classification criterion of each antivirus software.

- **Step 3 - Malware Family Clustering**
  Using the labels provided by the public repositories, we categorize the samples into distinct malware families.

- **Step 4 - DAMANI: Binary-to-Image Conversion**
  The analyzed binaries are converted into RGB images of a fixed-dimension, compatible with the input dimension of the 2D-CNN model. This representation still preserves local byte- level patterns.

- **Step 5 - DAMANI: Dataset Splitting**
  The images are divided in three datasets: one for training, one for evaluation and one for testing. To do this, malware binaries are divided in families, to best balance these datasets.

- **Step 6 - DAMANI: Antivirus Mimicking**
  A 2D-CNN is trained to approximate the decision boundaries of the antivirus engine based on the transformed representations and the labels provided by the antivirus. The network learns to emulate the internal detection logic of the antivirus without requiring access to source code or signature database.

- **Step 7 - DAMANI: Signature Extraction**
  An autoencoder is trained on a specific malware family. After the training, it is used to identify the byte sequences of the input sample that mostly contribute to malicious classifications. This step is repeated for the five most present malware families.

- **Step 8 - DAMANI: Signature Post-Processing**
  All extracted sequences are post-processed, aggregating similar patterns to obtain representative signature-like byte sequences.
- **Step 9 - Repeat for Other AVs**
  Steps 2-8 are repeated for the other two antivirus engines.
- **Step 10 - Signature Analysis**
  The extracted signatures are analyzed and compared to collect some statistics.

In Figure 4.1 we provide a high-level diagram that illustrates the overall pipeline and the input/output relationships of each step.

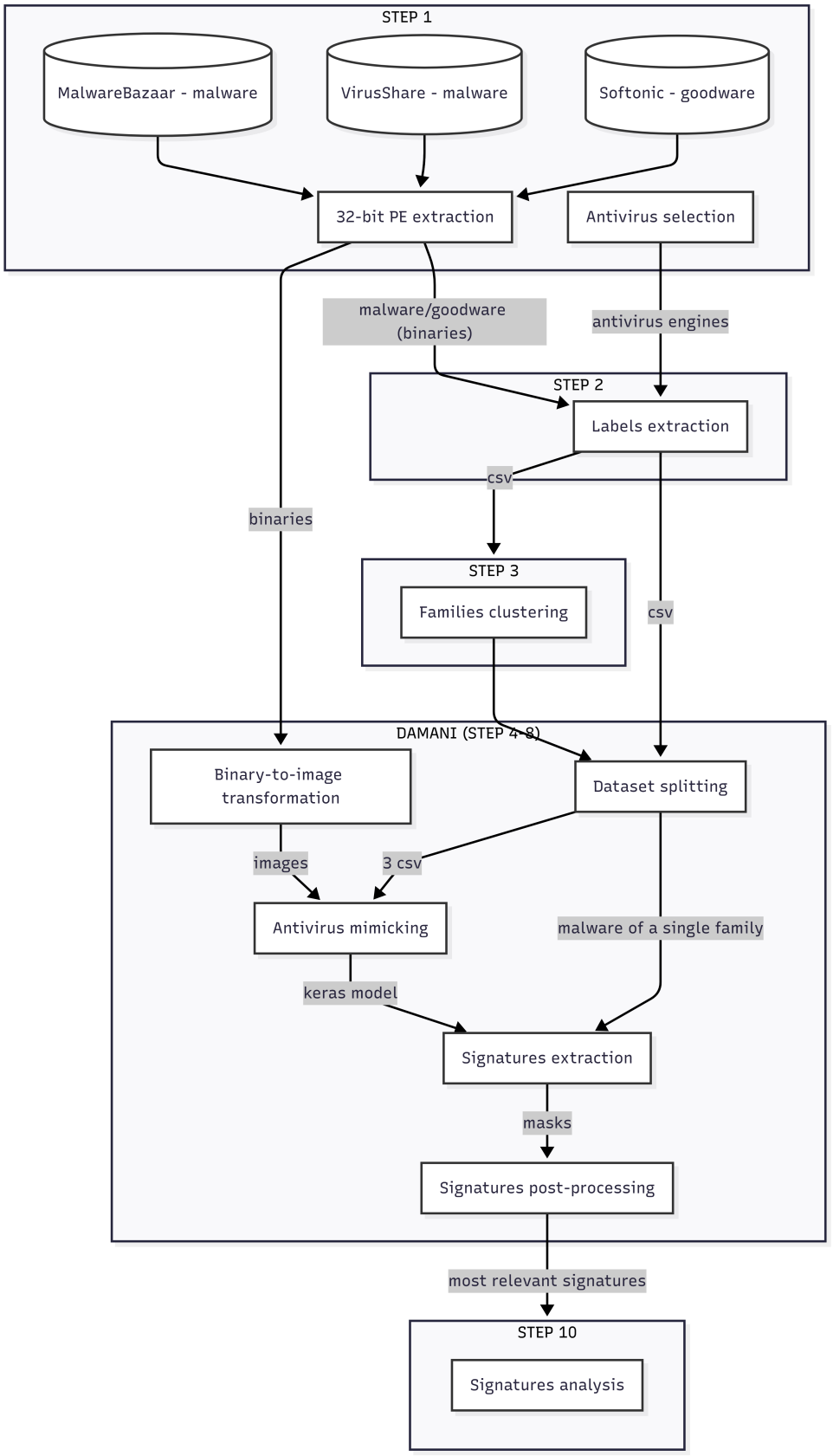


Figure 4.1: Block Diagram

4.2. Approach Details

In this section we expand the logical steps introduced in the previous section, clarifying the role of each phase in the overall pipeline and providing a detailed description of the DAMANI framework.

4.2.1. Antivirus Selection and Dataset Collection

The first phase of our work consisted in selecting a set of antivirus engines and building a dataset of both malicious and benign Portable Executable (PE) files. We selected three different antivirus products in order to capture heterogeneous detection strategies and design philosophies, and to reduce vendor-specific bias in the extracted signatures. Using multiple engines allows us not only to extract signatures from a single detector with our framework, but also to compare the extracted patterns across different vendors.

The three antivirus we selected are Windows Defender [55], ClamaAV [22], and Avast Premium Security [14]. We choose Windows Defender as a representation of a free antivirus. ClamAV was instead selected due to its transparency: since the signatures database is accessible, we used it to prove the goodness of extracted signatures. Lastly, we used Avast Premium Security as a representation of commercial antivirus: it is the most representative software of this family due to the usage of its detection engine in other antivirus from the same company (such as Norton and AVG).

Malware samples were collected from two public repositories, MalwareBazaar [49] and VirusShare [70], which provide a wide range of real-world malicious binaries captured in the wild. Benign software (goodware) was instead collected from Softonic [6], a popular and well-known software distribution platform. We downloaded some archives and then post-processed the obtained files: we retained only 32-bit PE files to guarantee architectural consistency, and we removed eventual duplicates using hash-based comparison, to avoid biasing the models with identical samples. This filtering process ensures a consistent and homogeneous dataset suitable for the pipeline. In the end, our dataset contained 288,197 binary files, out of which 170,926 are malware and 117,271 are goodware.

4.2.2. Labels Extraction

Once the dataset is built, each binary is scanned by the selected antivirus engine in order to obtain detection labels. To do this, we had to use a different interaction mechanisms for every antivirus (such as daemons, user interface, and command line interface). The antivirus is treated as a black-box classifier: for every input file, it outputs a decision

(malicious or not). These labels are used to train the mimicking model. By learning from the antivirus outputs rather than from correct manually assigned labels, the framework aims at replicating the actual decision boundaries adopted by each engine in its detection process: if the antivirus makes a wrong evaluation of a software, the model needs to perform the same misclassification. These labels are saved in a CSV file that will be used to divide the dataset in three smaller datasets. This step is repeated independently for each antivirus, producing three labeled datasets that reflect the behavior of the corresponding antivirus engine.

4.2.3. Malware Family Clustering

We divide the malware in ten malware families: worm, virus, scareware, trojan, email, ransomware, rootkit, botnet, backdoor, and hacktool. This is useful for one of the next steps: the dataset splitting, where is fundamental to balance all kind of malware in the three datasets. Among these families, we select the five families that contain the largest number of samples to train the signature extraction model. These families are: virus, trojan, backdoor, scareware and worm.

4.2.4. Binary-to-Image Conversion

To exploit convolutional architectures, PE binaries are transformed into RGB images, saved in PNG format to avoid lossy compression. Each 3-byte sequence of a binary file is mapped into a pixel, preserving local spatial relationships between neighboring bytes. Consecutive bytes are arranged row-wise to form a two-dimensional matrix, and then grouped into three channels to obtain an RGB representation. Files are padded and resized to match the input dimension required by the 2D-CNN model, that is $224 \times 224 \times 3$. Although the representation changes the data structure, it still preserves local byte-level patterns that can be efficiently processed by convolutional filters. This transformation allows the network to learn from the images the malicious patterns of the binary.

4.2.5. Dataset Splitting

For the mimicking model, the labelled dataset is divided into three subsets: training set, validation set, and test set. To reduce bias and improve generalization, malware are splitted taking into account their family, to ensure that samples from the same family are uniformly divided among every subset. We used 70% of the binaries for training, 15% for validation and other 15% for testing. The training set is used to learn the model parameters, the validation set to tune hyperparameters and prevent overfitting with the

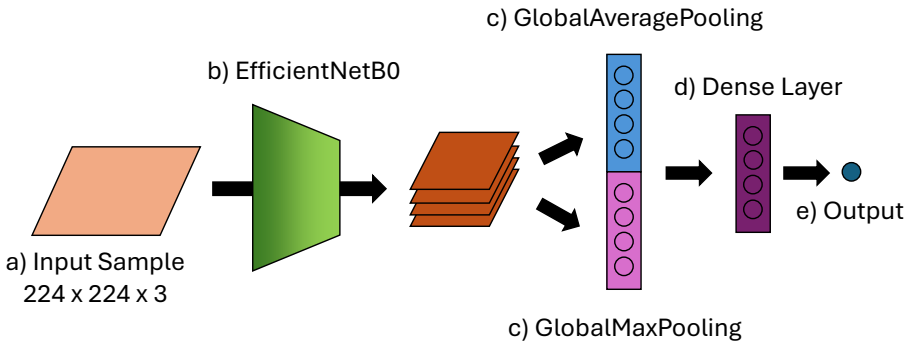


Figure 4.2: Convolutional Neural Network architecture

early stopping, and the test set to evaluate the performance of the mimicking model on previously unseen samples, better representing real-world scenarios. In this step the CSV file provided in previous step is divided into three different CSV files, one for every subset. For the signature extraction model, we isolated samples correctly identified as malware, and we created a triplet of training, validation, and test sets for each family, in order to analyze specific binary family separately. In the test sets, we included also false negatives and true positives, in order to compute relevant statistics that can be seen in Chapter 6.

4.2.6. Antivirus Mimicking

A Two-Dimensional Convolutional Neural Network is trained to mimic the behavior of each antivirus engine. As a starting architecture, we adopted EfficientNet-B0 [68] initialized with ImageNet pre-trained weights, which provides an effective trade-off between accuracy and computational cost. This choice was also motivated by previous work on antivirus mimicking [30], where the same architecture demonstrated strong performance in approximating antivirus decision boundaries. Using an already validated design allowed us to build upon a stable and well-performing baseline.

The architecture of the 2D-CNN is shown in Figure 4.2. The input of the network consists of the RGB image representation of binaries, while the target labels are the corresponding outputs provided by the antivirus, given to the model in three CSV files. Given an input image (a), the output feature maps of EfficientNet-B0 (b) are aggregated using both Global Average Pooling and Global Max Pooling (c). This representation is then normalized regularized using Batch Normalization and Dropout respectively. Follows a fully connected layer (d) with ReLU activation and L2 weight regularization, to penalize large weights and improve generalization. The final classification layer consists of a single

neuron with sigmoid activation (e), producing a maliciousness score in the range $[0, 1]$. A threshold of 0.5 is adopted: values above the threshold are classified as malicious, while lower values are classified as benign. The model is trained using the Binary Cross-Entropy loss function. Optimization is performed using the Adam optimizer, which efficiently adapts learning rates during gradient descent. Training is performed for 30 epochs, with early stopping based on validation loss to prevent overfitting.

The objective of this phase is not to directly detect malware, but to reproduce as accurately as possible the decisions of the antivirus, effectively creating a mimicking model of the engine capable of predicting the antivirus output. This mimicking model becomes the basis for the signature extraction step.

4.2.7. Signature Extraction

To extract representative malicious byte-level patterns, we employ a Convolutional Autoencoder (CAE) integrated with the previously trained 2D-CNN. Unlike a standard autoencoder used only for reconstruction, this model is optimized to generate sparse masks that preserve the malicious semantics learned by the antivirus mimicking model.

Every autoencoder is trained in an unsupervised manner on malware samples belonging to a single malware family at a time, allowing it to focus on structural properties and patterns that characterize that specific family.

The encoder receives as input batches of RGB images, that represent True Positive malware samples, and progressively compresses them through a sequence of convolutional layers, performing spatial downsampling. Each block consists of a convolutional layer followed by a LeakyReLU activation and Batch Normalization. The number of filters increases across layers, allowing the encoder to capture increasingly abstract spatial features. The resulting feature maps are flattened and projected into a compact latent space of fixed dimension, which represents a compressed encoding of the malware structure.

The decoder mirrors the encoder architecture. Starting from the latent representation, a fully connected layer reshapes the vector back into a spatial volume, which is then progressively upsampled using transposed convolutions. Each upsampling block applies Conv2DTranspose, LeakyReLU, and Batch Normalization. A final convolution with sigmoid activation reconstructs a mask with values in the range $[0, 1]$, representing the relevance of each byte. A visual representation of the autoencoder structure can be seen in Figure 4.3.

The output of the autoencoder is discretized using a Straight-Through Estimator (Round-

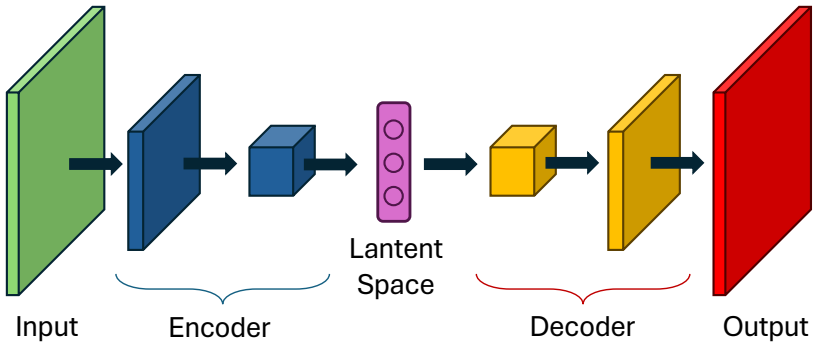


Figure 4.3: Convolutional Autoencoder architecture

STE) to obtain a binary mask. The RoundSTE layer applies a rounding function to the output of the autoencoder, forcing each output value to be either 0 or 1. During stochastic gradient descent, the round function is replaced with a similar, but differentiable function, in order to improve training convergence. The resulting binary mask is applied to the original malware, using the AND operator, to isolate the most malicious bytes. The masked malware is then reprocessed into RGB format and fed into the mimicking model.

The 2D-CNN output provides a feedback that guides the autoencoder to maximize the *maliciousness score*, while minimizing both the number of ones in the mask and the residual malicious content outside the extracted signatures. In this way, the autoencoder is not just reconstructing the input, but is learning to extract minimal sequences of bytes that are sufficient to trigger a malicious classification. After 15 epochs, the autoencoder is able to highlight which bytes sequences contribute the most to the malicious classification produced by the 2D-CNN that mimics the antivirus. These sequences are candidate signatures fragments used by the original antivirus engine. For each antivirus, this process is repeated for the five malware families selected in step 3.

4.2.8. Signature Post-Processing

The highlighted regions obtained from the autoencoder are post-processed to extract contiguous byte sequences. Since the obtained bytes sequences can be short and fragmented, we used a smoothing procedure to obtain more representative and consistent signature-like sequences.

4.2.9. Repeat for Other AVs

The entire process of labeling, 2D-CNN training, and autoencoder training is repeated independently for each selected antivirus engine. This design allows the framework to isolate the detection logic of each engine and later compare the extracted signatures across vendors. By keeping the pipeline identical for all antivirus products, differences in extracted patterns can be attributed to the engines' detection strategies rather than to variations in pre-processing or model configuration.

4.2.10. Signature Analysis

The resulting signatures are analyzed statistically to investigate commonalities among antivirus engines. This analysis provides insight into how different vendors implement static detection and whether shared byte-level patterns exist across engines. In addition, in our work we also perform a semantic analysis of the extracted signatures: the most frequent and representative byte sequences shared across antivirus engines are decompiled in order to inspect the underlying assembly instructions. This step allows a higher-level understanding of the code semantics captured by antivirus detection logic, providing further insight into what kind of operations and structures are targeted by detection mechanisms.

5 | Implementation

This chapter describes the practical implementation of the approach introduced in the previous chapter. Here, we focus on how each component has been concretely realized.

The goal of this chapter is to provide enough technical detail to ensure reproducibility and to clarify the engineering choices behind the proposed pipeline. In particular, we describe the software environment, the construction of the dataset, the DAMANI framework, and the final analysis.

5.1. System Details

The proposed framework was implemented in Python (3.13.2), leveraging several widely adopted machine learning and data processing libraries. Deep learning models were developed using TensorFlow (2.20.0) and Keras (3.10.0), while NumPy (2.1.3) was employed for numerical operations and image manipulation. A virtual environment was used to enable consistent experimental conditions across runs. All random seeds were fixed to ensure reproducibility of the experiments, and trained models were saved using Keras checkpoints to allow later reuse during the autoencoder optimization phase. All experiments were conducted in an isolated virtualized environment to ensure both safety and reproducibility. In particular, two virtual machines were employed: a Windows 10 Virtual Machine (VM) on VirtualBox [8] was used to execute and interact with the Windows Defender and Avast Premium Security antivirus software, since they cannot be used on a Linux OS. This environment was responsible for malware scanning, label extraction, but also data preprocessing (such as binary-to-image conversion, and dataset splitting). Secondly, a Linux Virtual Machine was used for interacting with ClamAV software, training autoencoders, and signature post-processing and analysis. Virtualization ensured isolation from the host system, preventing accidental execution of malware outside the controlled sandbox. For the most computationally intensive phase, that is the training of the 2D-CNN mimicking models, Google Colab [1] was employed. Colab provides access to GPU-accelerated hardware, significantly reducing training time compared to local execution. CNN models were trained in this environment using NVIDIA GPU (T4 GPU), enabling faster convergence

and efficient experimentation with different hyperparameters.

5.2. Dataset Collection

We collected raw binaries from MalwareBazaar [49], VirusShare [70], and Softonic [6], downloading different ZIP archives. After decompression, every file was checked by a Python program that, using the `pefile` library, retains only 32-bit Portable Executable files. After this, the program calculates the SHA-256 hash, using the `hashlib` Python library, and saves it in a file, if not already present, otherwise it is removed. We choose to retain only 32-bit PE files in order to maintain a homogeneous representation across the dataset, and to not bias the 2D-CNN model with the 32 and 64-bit differences. Every file was renamed with the SHA-256 value and, for malware, the family label already given by MalwareBazaar and VirusShare. To do this, we used the Windows 10 VM on VirtualBox, to prevent infections that could happen with the malware software. We had to deactivate Windows Defender in order to prevent unwanted file deletion that is automatically provided by the software.

```
def is_pe_file(filepath):
    try:
        pe=pefile.PE(filepath)
        pe.close()
        return True
    except pefile.PEFormatError:
        return False

def calculate_sha256(path):
    sha256 = hashlib.sha256()
    with open(path, "rb") as f:
        for chunk in iter(lambda: f.read(8192), b""):
            sha256.update(chunk)
    return sha256.hexdigest()

directory="malware"
for file in os.listdir(directory):
    src_path = os.path.join(directory, file)
    dst_path = os.path.join("not_valid", file)
    if not is_pe_file(file):
        shutil.move(src_path, dst_path)
    else:
```

```
hash_hex = calculate_sha256(src_path)
# existent_hash is an array of hashes extracted from a TXT file
if hash_hex not in existent_hash:
    existent_hash.add(hash_hex)
else:
    shutil.move(src_path, dst_path)
```

5.3. Labels Extraction

Once the dataset was built, we scanned each binary using every antivirus engine, in order to obtain detection labels. For each AV software, a CSV file is created (called "`labels.csv`"). This file contains an entry for every file (both goodware and malware), the entry is formatted as "`filename,label`", with label that is equal to "Goodware" or "Malware".

- **Defender**: we implemented a Python program that scans one binary file at a time (calling `MpCmdRun.exe` with `subprocess` Python library) and analyzes the output of that command: if the string **`"threats found"`** is present, then the file is malicious for Defender, otherwise, if the string **`"no threats"`** is present, the file was not revealed as malicious. For each file, the program adds a line in the CSV file;
- **ClamAV**: we wrote a Python program that uses the library `pyclamd` to interface with the daemon of ClamAV (`clamd.ctl`). We used the command `contscan_file()` that scans all binary files and returns an array with the names of malicious files. After parsing the output, the program creates the CSV file;
- **Avast Premium Security**: we activated the free-trial of 30 days and used the provided Graphic User Interface to analyze the folder where all binaries where contained, flagging the option to write all malicious files' names on a TXT file. Given this file, we create the complete CSV, adding an entry for non-malicious files too.

In the end, each entry in the dataset is therefore associated with the corresponding antivirus decision, which is later used for the mimicking model. This abstraction layer allows the procedure to remain independent from the specific antivirus implementation, making it possible to extend the pipeline to additional engines with minimal modifications.

5.4. Malware Family Clustering

To enable both family-specific signature extraction and three balanced datasets for training, validation, and testing, malware samples are grouped into families. Malware's family

Family	Malware
Worm	8356
Virus	15669
Scareware	12960
Trojan	112709
Email	3033
Ransomware	2346
Rootkit	245
Botnet	665
Backdoor	12216
Hacktool	2727

Table 5.1: Distribution of malware in families

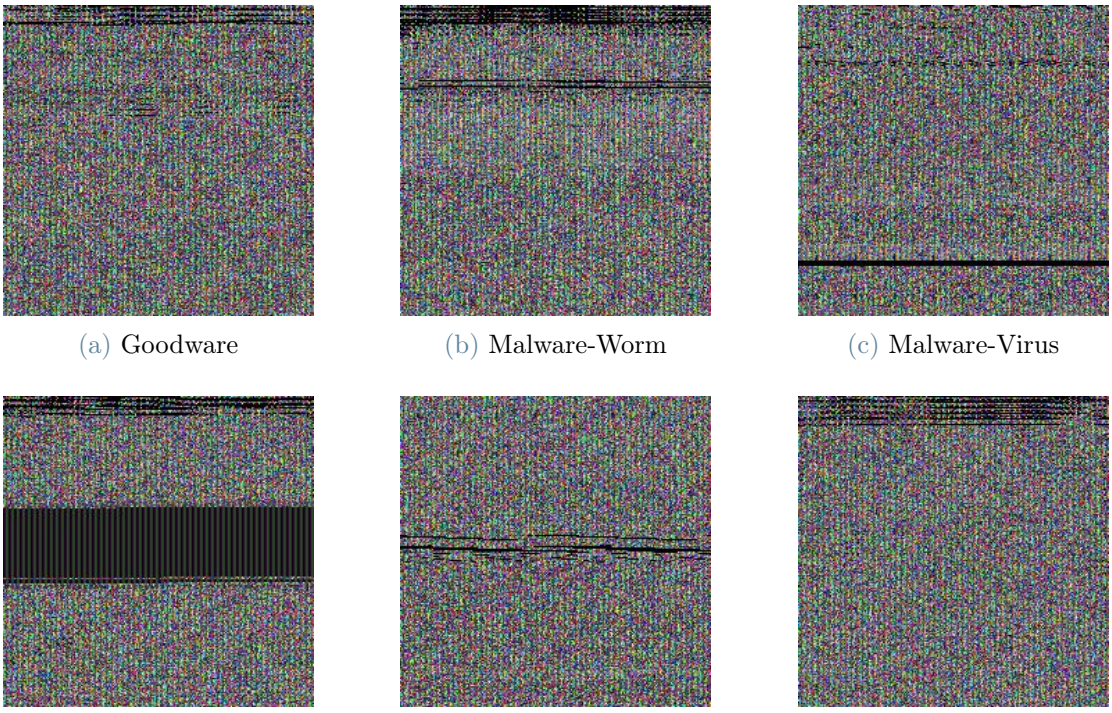
information were derived from the labels provided by the public repositories. Only families with a sufficient number of samples were retained to ensure stable training, these ten families are: worm, virus, scareware, trojan, email, ransomware, rootkit, botnet, backdoor, and hacktool. In Table 5.1 we show the division of the 170926 malware into these ten families.

After identifying the families, we use a Python program that, given the "`labels.csv`" file, parses the filenames of all True Positive malware and creates another labels' CSV file, "`parsed.csv`", where the labels of malware (both True Positive and False Negative) also include the name of the family (e.g. "Malware-Trojan" or "Goodware-Trojan"). We use this file for the dataset splitting.

5.5. Binary-to-Image Conversion

We use a Python program that reads each binary file and converts it into a tensor. Then, the dimension of the file is used to calculate the minimum width needed to obtain a squared image of 3 layers, without truncating the original file. For this reason a padding of zeroes is usually needed. The byte stream is then reshaped into a two-dimensional matrix and mapped to RGB channels. The resulting image is resized, using bilinear interpolation, to a fixed resolution compatible with EfficientNet-B0 input format (224×224×3). Finally, the image is saved in PNG format to preserve information without introducing additional loss.

```
def RGBLayer(tensor):
    pixel_array = tf.cast(tensor, tf.float32)
    width = get_width(tf.shape(pixel_array)[0])
```



(a) Goodware (b) Malware-Worm (c) Malware-Virus

(d) Malware-Scareware (e) Malware-Trojan (f) Malware-Backdoor

Figure 5.1: Output of the Binary-to-Image Conversion for a goodware and 5 malware, selected from the largest malware families.

```
height = width
pad_dim = width * height * 3 - tf.shape(pixel_array)[0]
pixel_array = tf.pad(pixel_array, [[0, pad_dim]])
pixel_matrix = tf.reshape(pixel_array, (width, height, 3))
image = tf.image.resize(pixel_matrix, [224, 224])
return image
```

We provide some output examples of the process, both for a goodware and for malware from the largest families in Figure 5.1.

Since the training of the mimicking CNN model is done on Google Colab, we had to divide all images into folders of maximum 9999 files (because folders with a larger number of files are difficult to manage on Google Drive). Then, we zipped all these folders and saved them on Google Drive.

5.6. Dataset Splitting

We wrote a Python program that, given the file "`parsed.csv`", divides all malware families and goodware in three datasets: training, validation and testing, containing 70%, 15% and

15% of all files respectively. These datasets are used for the antivirus mimicking phase. In the division, we balanced True Positives, False Negatives, True Negatives, and False Positives; and we paid attention to balance each malware family among True Positives and False Negatives across the datasets. For the signature extraction phase, instead, we only employed malware samples from the largest five families, in order to have enough data for each malware family throughout the whole training pipeline. We selected the samples correctly identified as malicious by the antivirus, but filtering out files that exceeded the input dimensions of the autoencoder's input (300B $\times$ 300B = 90KB). For every family, we created the training, validation and test datasets, with the same percentages as those for the mimicking model.

5.7. Antivirus Mimicking

The goal of this phase is not to build a generic malware detector, but to approximate as closely as possible the decision logic of each antivirus engine by treating it as a black-box classifier. For this reason, a Two-Dimensional Convolutional Neural Network (2D-CNN) is trained to mimic the outputs produced by the antivirus. The input of the network consists of the RGB image representation of PE binaries, while the target labels are the ones in the CSV file obtained directly from the antivirus scans, "`labels.csv`". Each antivirus engine therefore produces its own labeled dataset and its own mimicking model. The adopted architecture is based on EfficientNet-B0, selected as a backbone CNN due to its strong performance-efficiency tradeoff and its ability to extract hierarchical spatial features from image-like representations. The structure of the mimicking CNN model is composed by the following layers.

- Input layer with dimensions $224 \times 224 \times 3$ (image representation of the samples);
- EfficientNetB0 layer, with `trainable` option set to true in order to perform both transfer learning and fine tuning. The output activation map has dimensions $7 \times 7 \times 1280$;
- GlobalAveragePooling2D *and* GlobalMaxPooling2D layers, with their output concatenated, yielding $1280 + 1280 = 2560$ neurons;
- BatchNormalization and 25% Dropout layers to reduce overfitting and improve generalization;
- Dense layer with 512 neurons, ReLU activation and L2 kernel regularizer with weight 10^{-3}. The kernel regularizer introduces a term in the final loss proportional to the sum of the squares of each neuron's weights, reducing overfitting;

- A second set of BatchNormalization and 25% Dropout layers;
- Output layer with a single neuron and sigmoid activation.

As loss function to be minimized, we employed Binary Cross-Entropy (BCE) with label smoothing $\epsilon = 0.1$:

$$\text{BCE}_{\text{smooth}}(y_{\text{true}}, y_{\text{pred}}) = -\left[(y_{\text{true}}(1-\epsilon) + \epsilon(1-y_{\text{true}}))\log(y_{\text{pred}})) + \right.$$

$$\left. + (1 - y_{\text{true}}(1-\epsilon) + \epsilon(1-y_{\text{true}}))\log(1-y_{\text{pred}}))\right]$$

As training optimizer, we employed Adam [41] with learning rate $\alpha = 10^{-3}$. As training callbacks, we employed EarlyStopping and ReduceLROnPlateau to improve training convegence.

We report the Python script employed to build and compile the mimicking CNN model.

```
def build_model(input_shape):
    inputs = tf.keras.Input(shape=input_shape)
    base_model = tf.keras.applications.efficientnet.EfficientNetB0(
        input_tensor=inputs, include_top=False, weights="imagenet"
    )
    base_model.trainable = True
    avg_pool = keras.layers.GlobalAveragePooling2D()(base_model.output)
    max_pool = keras.layers.GlobalMaxPooling2D()(base_model.output)
    x = tf.keras.layers.Concatenate()([avg_pool, max_pool])
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Dropout(0.25)(x)
    x = keras.layers.Dense(512, activation="relu",
        kernel_regularizer=tf.keras.regularizers.l2(0.001)
    )(x)
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Dropout(0.25)(x)
    out = tf.keras.layers.Dense(1, activation=tf.keras.activations.sigmoid)(x)
    model = tf.keras.Model(inputs, out)
    model.compile(
        loss=tf.keras.losses.BinaryCrossentropy(label_smoothing=0.1),
        optimizer=tfk.optimizers.Adam(1e-3), metrics=["accuracy"]
    )
    return model
```

During training, the model is periodically saved at the end of each epoch using a check-

pointing strategy. This approach prevents the loss of progress in case of unexpected interruptions, such as system disconnections, session timeouts, or exhaustion of computational resources. By storing intermediate versions of the network, training can be safely resumed without restarting from scratch. This mechanism is particularly important when training models that require long execution times. The model is saved in a `.keras` file instead of the legacy `.h5` format. The `.keras` format is the modern serialization standard of Keras and is designed to provide better forward compatibility, improved security, and more reliable model restoration across different framework versions [7]. It preserves not only the network architecture and weights, but also the optimizer state, loss functions, and training configuration in a more robust way.

5.8. Signature Extraction

Having established a CNN model as a reliable baseline for antivirus mimicking, we now leverage it to isolate specific malicious regions within the binary. We describe our signature extraction model in two stages. The first stage introduces a Convolutional Autoencoder designed to perform dimensionality reduction and localizing malicious byte patterns. Taking the malware as input, the network compresses the data into a low-dimensional latent space. The decoder then reconstructs this representation into a "soft" mask with continuous entries in the interval $[0, 1]$. This mask represents a probability map, indicating the likelihood of specific pixel regions contributing to the malicious nature of the file.

The second stage establishes the training loop that refines the autoencoder. Here, the generated soft mask is rounded to enforce decision boundaries. This binary mask is applied to the original image via element-wise multiplication, filtering out non-essential data and isolating the selected signatures. The resulting masked image is then forwarded to the antivirus mimicking model, that assigns a score based on the malware detection criterion learned from the specific antivirus. This downstream model provides the gradient feedback necessary to train the autoencoder.

5.8.1. Convolutional Autoencoder

In this stage, we build the mask-generating autoencoder, which processes the initial malware. By compressing the input into a low-dimensional latent space, the network forces an abstraction of the binary's structure. The decoder then reconstructs this latent vector into a continuous soft mask with values in the interval $[0, 1]$.

The autoencoder is composed by the following layers.

- **Input:** Raw binary file reshaped on a 2D grid;
- **Encoder:** A sequence of three blocks, each consisting of:
  - 3×3 Conv2D layer with `strides` $= 2$ and increasing filter counts to reduce spatial dimensions;
  - LeakyReLU activation function;
  - BatchNormalization to improve model generalization;

  These blocks are followed by a Flatten layer to produce the latent feature vector.
- **Decoder:** A Dense layer and a Reshape layer to recover an initial spatial structure, followed by three blocks containing:
  - 3×3 Conv2DTranspose with `strides` $= 2$ and decreasing filter counts to recover the original spatial dimensions;
  - LeakyReLU activation function;
  - BatchNormalization layer;

  The decoder concludes with a sigmoid activation layer to normalize the output entries to the range $[0, 1]$.
- **Autoencoder:** the encoder and decoder are sequentially connected to form the complete architecture.

We report the Python script employed to build the autoencoder.

```
def build_autoencoder(width, height, depth, filters=(8, 16, 32), latentDim=32):
    # Encoder
    input_layer = tf.keras.layers.Input(shape=(width, height, depth))
    x=input_layer
    for f in filters:
        x = tf.keras.layers.Conv2D(f, (3, 3), strides=2, padding="same")(x)
        x = tf.keras.layers.LeakyReLU(alpha=0.2)(x)
        x = tf.keras.layers.BatchNormalization(axis=-1)(x)
    volumeSize = tf.keras.backend.int_shape(x)
    x = tf.keras.layers.Flatten()(x)
    latent = tf.keras.layers.Dense(latentDim)(x)
    encoder = tf.keras.Model(input_layer, latent)

    # Decoder
    latentInputs = tf.keras.layers.Input(shape=(latentDim,))
```

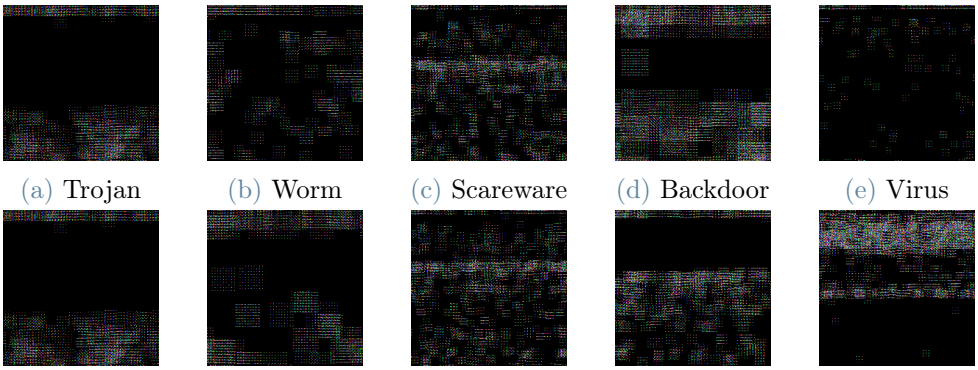
```
    x = tf.keras.layers.Dense(np.prod(volumeSize[1:]))(latentInputs)
    x = tf.keras.layers.Reshape(
        (volumeSize[1], volumeSize[2], volumeSize[3])
    )(x)
    for f in filters[::-1]:
        x = tf.keras.layers.Conv2DTranspose(
            f, (3, 3), strides=2, padding="same"
        )(x)
        x = tf.keras.layers.LeakyReLU(alpha=0.2)(x)
        x = tf.keras.layers.BatchNormalization(axis=-1)(x)
    x = tf.keras.layers.Conv2DTranspose(depth, (3, 3), padding="same")(x)
    outputs = tf.keras.layers.Conv2D(
        depth, (3, 3), activation="sigmoid", padding="same
    )(x)
    decoder = tf.keras.Model(latentInputs, outputs)

    # Autoencoder
    autoencoder = Model(input_layer, decoder(encoder(input_layer)))
    return autoencoder
```

5.8.2. Training Pipeline

We now describe the training pipeline, where the generated soft mask is rounded and applied element-wise to the original malware. The masked malware is forwarded to the pre-trained antivirus mimicking model, which acts as a static oracle to validate the maliciousness of the retained bytes. Through this feedback loop, the autoencoder is optimized to isolate malicious signatures.

The pipeline starts by rounding the autoencoder's output to perform signature extraction. While the autoencoder generates a "soft" mask with continuous values between $[0, 1]$, a signature is strictly binary: a byte is either part of the malicious code (1) or it is not (0). To fix this, a custom `RoundSTE` layer forces the continuous values into strict integers of 0 or 1. Since a standard rounding operation blocks the learning process (because its derivative is zero, preventing backpropagation), this layer uses a technique called Straight-Through Estimator during training. This approximates the gradient using a sharp sigmoid function, allowing the network to update its weights even though the output is being rounded.

```
@tf.custom_gradient
def round_ste(x):
    y = tf.round(x)
```



(a) Trojan (b) Worm (c) Scareware (d) Backdoor (e) Virus

(f) Trojan (g) Worm (h) Scareware (i) Backdoor (j) Virus

Figure 5.2: Images of masked malware extracted from the autoencoder.

```
    def grad(dy):
        sharpness=20.0
        sigmoid = tf.sigmoid(sharpness*(x-0.50))
        grad_sigmoid = sharpness * sigmoid * (1.0 - sigmoid)
        return dy * grad_sigmoid
    return y, grad

class RoundSTE(tf.keras.layers.Layer):
    def call(self, input_val):
        return round_ste(input_val)
```

Next, a `preprocLayer` handles the actual filtering of the data. This layer takes the binary mask and the original malware bytes, and performs an element-wise multiplication: any byte aligned with a mask value of 0 is turned into a 0, while bytes aligned with a mask value of 1 are kept. This step hides the parts of the file that the autoencoder considers irrelevant, leaving only the selected signature bytes visible.

The filtered binary data is then converted into an image format using a custom `RGBLayer`. This layer works in the exact same way of the program used for the Binary-to-Image conversion, to maintain coherence for the input data that are fed into the mimicking model.

Finally, we integrate these individual components into a unified training framework. After the malware is processed by the autoencoder and rounded by the `RoundSTE` layer, the architecture uses two parallel evaluation branches to guide the learning process. In the primary branch, the generated mask is applied to the malware, converted to an image, and forwarded to the CNN mimicking model. Since this model remains frozen during training, it functions as a static oracle, providing a probability score that quantifies the

maliciousness of the signatures. Simultaneously, a secondary "inverted" branch applies the logical complement of the mask ($1 - \texttt{mask}$) to the input. This mechanism evaluates the content excluded by the autoencoder, verifying that the discarded bytes do not contribute to the malicious classification.

The output of the model is composed by the rounded binary mask returned by the autoencoder, and the mimicking model's scores for both the masked malware (that is, the selected signatures) and its inverse (that is, the bytes *not* selected by the binary mask).

```
def build_model(input_shape):
    # Inputs
    malware = tf.keras.Input(shape=input_shape)
    lengths = tf.keras.layers.Input(shape=(), dtype=tf.int32)

    # Autoencoder
    output_autoencoder = build_autoencoder(300,300,1)(malware)

    # Mask and inverted mask
    mask = RoundSTE(name = "round_layer")(output_autoencoder)
    masked_malware = preprocLayer()([mask,malware])
    masked_inverse_malware = preprocLayer()([1.0 - mask,malware])

    # Conversion to RGB image
    image_rgb = RGBLayer()((masked_malware,lengths))
    inverted_image_rgb = RGBLayer()((masked_inverse_malware,lengths))

    # Antivirus mimicking oracle
    mimicking_model = tf.keras.models.load_model("mimicking_model.keras")
    for l in mimicking_model.layers:
        l.trainable = False
    preproc_input =
    ↪  tf.keras.applications.efficientnet.preprocess_input(image_rgb)
    preproc_inverted_input =
    ↪  tf.keras.applications.efficientnet.preprocess_input(inverted_image_rgb)
    prediction_cnn = mimicking_model(
        preproc_input, training=False, name="prediction_cnn"
    )
    prediction_cnn_inverted = mimicking_model(
        preproc_inverted_input, training=False, name="prediction_cnn_inverted"
    )
```

```
# Final model
model = tf.keras.Model(
    [malware,lengths],
    [mask, prediction_cnn, prediction_cnn_inverted]
)
return model
```

5.8.3. Loss Function Design

We now move to the design of the loss function. Since this is a multi-objective optimization problem, we must balance three goals: we need to minimize the number of the extracted bytes, while ensuring that the signatures retain a high maliciousness score and that the removed bytes (selected by the inverted mask) result in a low maliciousness score.

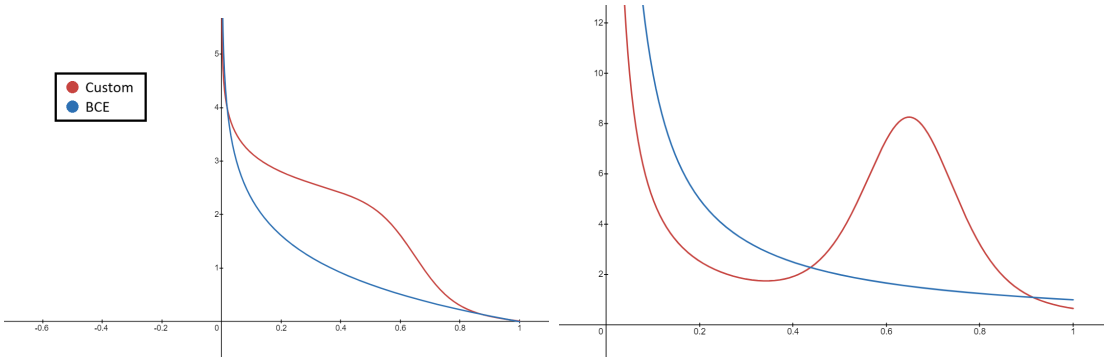
To achieve the first objective (signature sparsity) we employ a Mean Squared Error (MSE) loss calculated between the generated binary mask and a target tensor of all zeros. By minimizing this term, we penalize the model for every byte it chooses to retain, pressuring the autoencoder to discard any feature that is not strictly necessary for the classification:

$$\mathcal{L}_{sparsity}(mask) = \frac{1}{N}\sum_{i=0}^{N} mask_i^2 \tag{5.1}$$

For the remaining two objectives, we found that standard loss functions like Binary Cross-Entropy (BCE) are insufficient for forcing the maliciousness score of the selected signautres to be sufficiently high. Therefore, we design a custom loss function, $\mathcal{L}_{mimic}$, which replaces standard BCE. The rationale behind this design is to create a non-linear objective that heavily penalizes uncertainty. It combines a standard logarithmic term with a shifted sigmoid function centered at 0.65 with a high sharpness factor (15.0). This steep slope incentivizes the model to push its predictions above 0.65 when the target is malicious (signature branch) and below 0.35 when the target is benign (inverted branch), preventing the model from converging to low-confidence regions.

Considering y_{pred} to be the output of the mimicking oracle, $y_{\text{true}} = 1$ for the selected signatures, $y_{\text{true}} = 0$ for the excluded bytes, and letting $\hat{p} = y_{\text{true}}y_{\text{pred}} + (1-y_{\text{true}})(1-y_{\text{pred}})$, the custom loss $\mathcal{L}_{mimic}$ is:

$$\mathcal{L}_{mimic}(y_{\text{true}}, y_{\text{pred}}) = -\frac{1}{2}\left(\ln(\hat{p}) - 4\left(\frac{1}{1+e^{-15(\hat{p}-0.65)}} - 1\right)\right) \tag{5.2}$$



(a) Function (b) Negative first order derivative

Figure 5.3: Comparison of function values and negative first order derivative between our custom loss $\mathcal{L}_{mimic}$ and Binary Cross-Entropy when $y_{\text{true}} = 1$.

Figure 5.3 compares $\mathcal{L}_{mimic}$ against Binary Cross-Entropy when $y_{\text{true}} = 1$ (meaning that the prediction should be maximized). As it can be seen, our custom loss provides a high gradient for values larger than 0.5, forcing the model to take strong decisions. In this way, we force the autoencoder at the beginning of the training pipeline to only provide significant signatures, and we avoid cases where the autoencoder proposes suboptimal signatures and the maliciousness score is slightly above 50%.

The final loss function is a linear combination of these terms, applied to the respective outputs of the model:

$$\mathcal{L}_{total} = \lambda_1 \mathcal{L}_{sparsity}(mask) + \lambda_2 \mathcal{L}_{mimic}(1, \hat{y}_{sig}) + \lambda_3 \mathcal{L}_{mimic}(0, \hat{y}_{inv}) \tag{5.3}$$

Here, $\hat{y}_{sig}$ represents the prediction for the isolated signature (target 1), and $\hat{y}_{inv}$ represents the prediction for the removed bytes (target 0). The optimal values for the weights $\lambda_1, \lambda_2,$ and λ_3 are selected through an exhaustive hyperparameter tuning search to ensure the best trade-off between signature sparsity and detection capability.

Some examples of images of masked malware are given in Figure 5.2.

5.9. Signature Post-Processing

After the autoencoder produces a binary mask highlighting the regions of the input that mostly contribute to malicious classification, an additional post-processing phase is required to transform these raw outputs into meaningful, stable, and analyzable signature-like byte sequences. This phase is implemented through a dedicated pipeline that refines the masks, extracts contiguous byte patterns and validates the resulting signatures, as we

present in next chapter. The autoencoder outputs a mask where each position corresponds to a byte in the original malware sample and takes value 1 if the byte is considered relevant, and 0 otherwise. However, raw masks may contain fragmented regions. To mitigate this issue, a sliding window smoothing procedure is applied. For each mask, overlapping windows of increasing size are scanned and converted into continuous regions when the majority of values inside the window are active. In particular, for each window of size w, if at least half of the elements are equal to 1 and both boundary elements are active, the whole window is set to 1. This sliding window mechanism to enforce spatial consistency and reduce fragmentation in the extracted regions. A critical parameter of this process is the window size, which has a trade-off: if the window is too small, the resulting signatures tend to be noisy and fragmented; if it is too large, the mask becomes overly coarse, including irrelevant bytes and reducing interpretability. For this reason, the window size was not chosen arbitrarily, but determined through a heuristic optimization procedure. Multiple candidate window sizes were evaluated, and for each configuration three quantities were measured:

- Malware preservation score (`means_mw`): the average output of the CNN mimicking model when the extracted masks are applied to malware samples. This value estimates how much maliciousness is retained after masking;
- Goodware suppression score (`means_gw`): the average output of the CNN the negated new mask is applied to the malware, so only bytes not contained in the extracted signatures are kept. A low value indicates that the excluded bytes do not trigger malicious classification;
- Average signature length (`len_sig`): the mean size of the extracted byte sequences, used to control the compactness of the resulting patterns.

For each window size w, these quantities are combined into a single heuristic score: $\texttt{score}(w) = \texttt{means\_mw}(w) + (1 - \texttt{means\_gw}(w)) + \lambda \texttt{len\_sig}(w)$ where λ is a weighting factor that balances discriminative power against signature length. We chose its value empirically.

The first term encourages the window to preserve malicious classification, the second penalizes configurations that leave out malicious bytes, and the third promotes signatures that are sufficiently compact. The window size that maximizes this heuristic score is selected as the optimal configuration for post-processing. This procedure ensures that the chosen window size reflects a compromise between faithfulness to the autoencoder's output and interpretability of the extracted signatures.

Once the masks are refined, the next step consists of extracting the actual byte sequences from the original malware binaries. For each sample, the mask is aligned with the original byte stream to obtain the `masked malware`, and all signatures are saved on a file. Not all extracted sequences are meaningful, therefore a filter is applied: sequences shorter than 4 bytes and trivial patterns (e.g., sequences composed entirely of zero bytes) are discarded. We choose 4 byte as the minimum signature length because even antivirus engines do not use shorter signatures, as we understood by analyzing the signatures extracted from ClamAV database. Each segment that is kept is written in a file, forming a set of signatures for each malware sample.

5.10. Signature Analysis

The processed signatures can now be compared across different samples of the same family and for the same antivirus to make statistics. A duplicate detection procedure is applied across all extracted signature files, and each unique signature is mapped to the set of malware samples in which it appears. Signatures are sorted by frequency and stored together with the list of files in which they occur. This allows the identification of most recurrent byte patterns across multiple malware instances: these are the potentially candidate signatures of that antivirus engine. After extracting most recurrent signatures for every antivirus and for every family, we do some analysis by searching signatures extracted simultaneously from different engines.

The extracted signatures are further analyzed through decompilation in order to inspect the underlying executable code. This phase employs the `pefile` module for handling PE files and the `capstone` Python library for binary disassembly. The process begins by reading the original PE file to locate the extracted signature within the file. Then, both the exact signature and the context are disassembled. Evaluating the context is fundamental to determine whether the signature's bytes belong to executable code or ASCII data (strings). This analysis allows us to visualize the semantic information present in extracted signatures.

6 | Experimental Validation

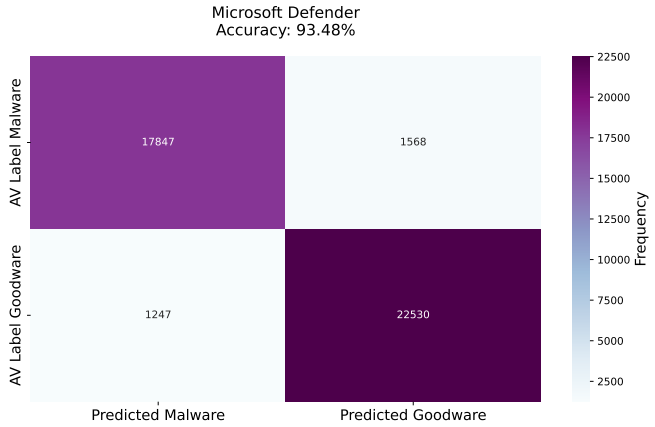
This chapter presents the experimental evaluation of the proposed DAMANI framework. The objective of the experiments is not just to measure classification accuracy of CNN mimicking models, but to assess whether the extracted byte-level signatures are related to the antivirus decisions approximated by the mimicking models. The evaluation aims to verify the functional role of extracted patterns in triggering malicious classifications. To this end, we designed a set of experiments to test the effect of signatures when isolated, removed, or injected into different binaries. After this validation phase, we present an analysis of the signatures extracted by DAMANI. We compare signatures extracted from different antivirus engines and present some examples of disassembled malicious byte patterns.

6.1. Evaluation Goals

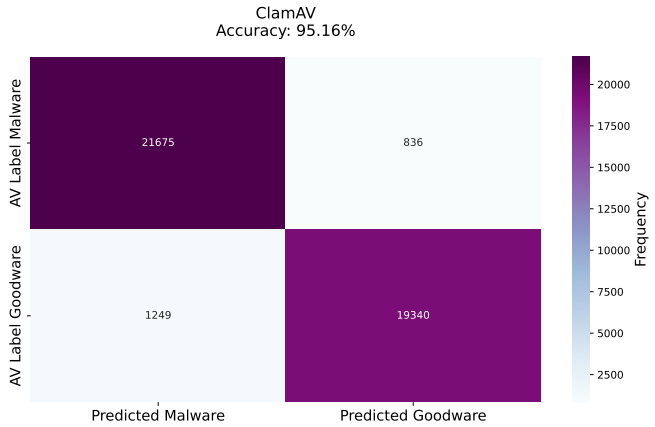
The experimental evaluation is designed to address the following main goals:

- **Mimicking Accuracy:** verify that the mimicking model accurately replicates the behavior of each antivirus software, not only performing malware classification but also applying the same classification criterion employed by the mimicked antivirus;
- **Signature Effectiveness:** verify that the extracted signatures are sufficient to preserve malicious classification when isolated from the rest of the malware sample;
- **Signature Necessity:** verify that removing the extracted signatures from malware significantly reduces the malicious score, showing that these regions are necessary for the mimicked antivirus decision;
- **Real-World Consistency:** verify that the extracted signatures reflect the behavior of real antivirus engines and their false positives and false negatives, demonstrating that the framework performs antivirus mimicking rather than generic malware detection.

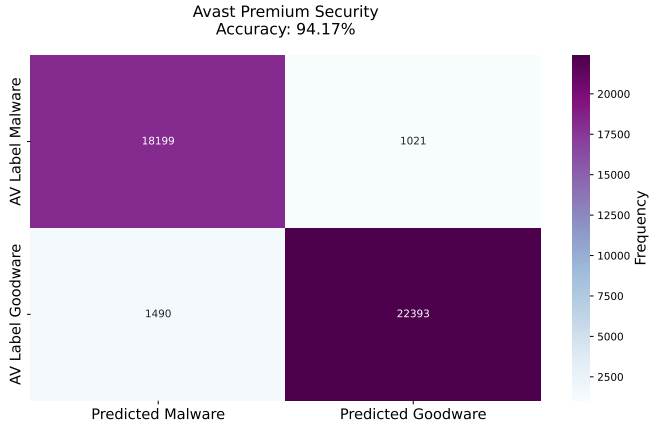
Together, these goals assess the reliability and interpretability of the extracted patterns.



(a) Microsoft Defender - Confusion Matrix



(b) ClamAV - Confusion Matrix



(c) Avast Premium Security - Confusion Matrix

Figure 6.1: Confusion matrices and accuracy values of CNNs of all antivirus engines on the entire test set.

6.2. Mimicking Accuracy

The first experiment evaluates the ability of the mimicking model of replicating the decisions of each antivirus software.

Test 1 - Accuracy on whole test set. Figure 6.1 reports the confusion matrices and accuracy values obtained for each mimicking model across the considered antivirus engines, when applied to the whole test set. As it can be seen, the accuracy is high across all the datasets, between 93% and 95%, with both malware and goodware classifications matching the labels assigned by the antivirus in the vast majority of cases.

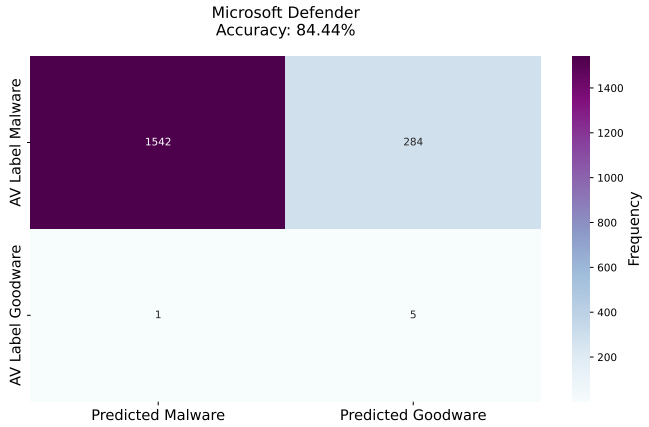
Test 2 - Accuracy on misclassified samples. To verify that the model effectively performs antivirus mimicking and not only generic malware classification, we performed an evaluation of the final model using the antivirus' False Positives (FP) and False Negatives (FN). False Positives include the goodware samples that are misclassified by the *antivirus* as malware, while False Negatives include the malware samples misclassified as goodware. These samples are fed to the CNN to assess whether the network reproduces the same misclassification behavior of the antivirus. The results of this test are shown in Figure 6.2: the high accuracy on FP and FN subsets, between 84% and 89%, indicates that the CNN has learned the antivirus decision boundaries, instead of simply distinguishing malware from benign files.

Outcome interpretation. These results quantify how closely the CNN approximates the behavior of the original antivirus classifiers, reflecting the classification criterion of each antivirus.

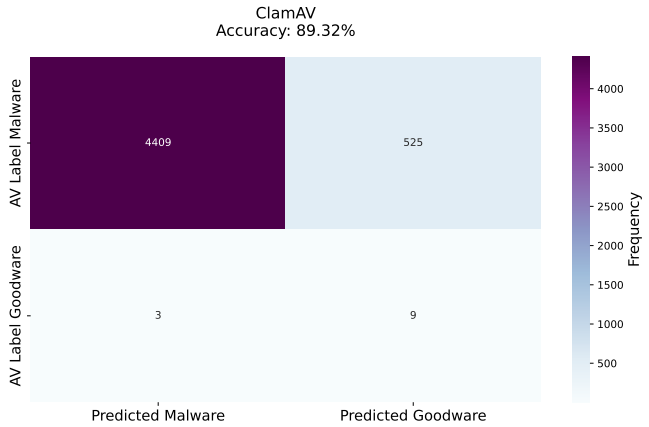
6.3. Signature Effectiveness

The second experiment evaluates whether the extracted signatures are sufficient to trigger malicious classification. The validation process has been performed starting from a CNN mimicking model because, as it can be experimentally verified, any input obtained by applying small modifications to binary files leads to undefined behavior in the actual antivirus. We discuss this phenomenon in Appendix A.

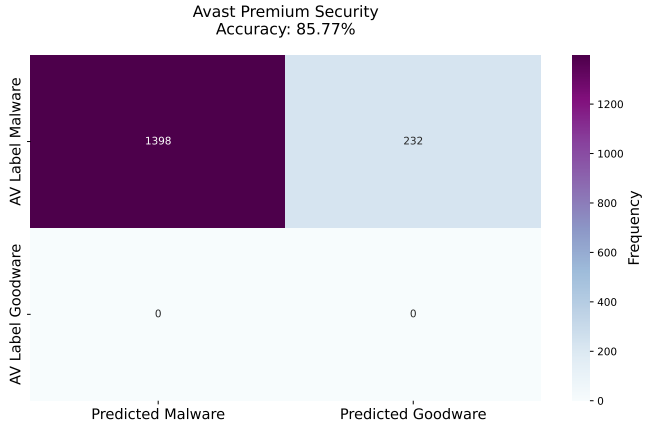
Test 1 - Malware with only signatures. For each malware sample, all bytes outside the extracted signature regions are set to zero, while signature bytes are preserved. The modified binaries are converted into RGB images and fed to the trained CNN mimicking



(a) Microsoft Defender - Confusion Matrix



(b) ClamAV - Confusion Matrix



(c) Avast Premium Security - Confusion Matrix

Figure 6.2: Confusion matrices and accuracy values of CNNs of all antivirus engines on samples misclassified by the antivirus.

Table 6.1: Percentage of masked malware (extracted signatures) classified as malware.

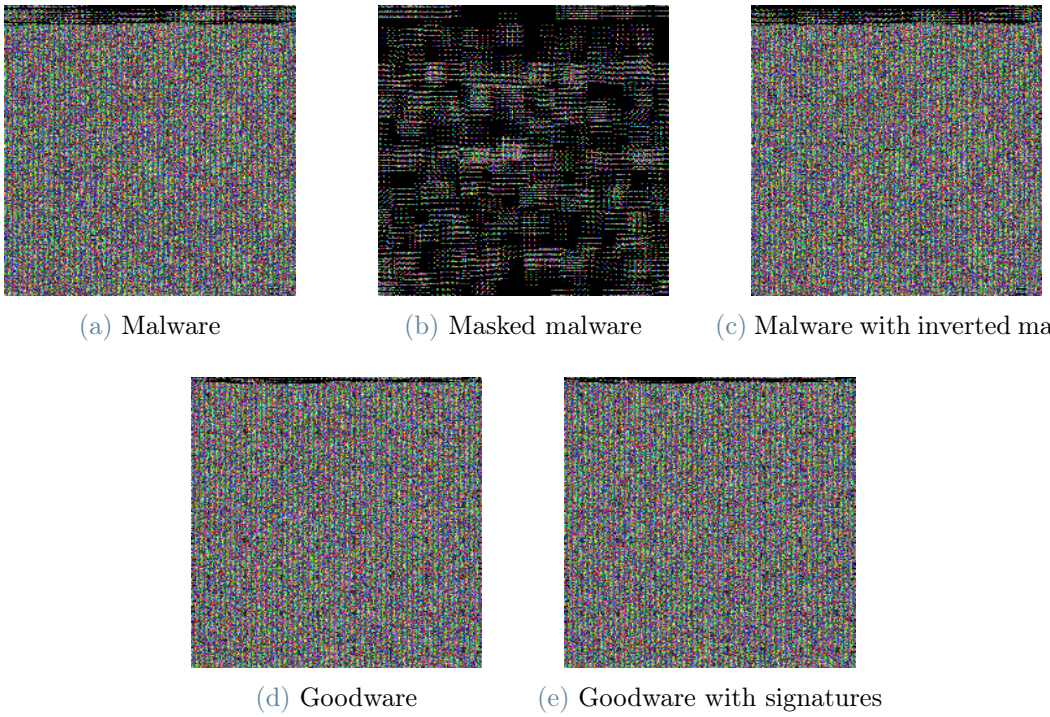
	Defender	ClamAV	Avast
Backdoor	90.83%	100.0%	93.10%
Worm	86.62%	100.0%	99.19%
Scareware	100.0%	100.0%	100.0%
Trojan	98.33%	98.85%	100.0%
Virus	91.86%	83.27%	91.96%

Table 6.2: Percentage of goodware with injected signatures classified as malware.

	Defender	ClamAV	Avast
Backdoor	74.16%	74.48%	72.37%
Worm	69.88%	96.26%	90.75%
Scareware	87.50%	100.0%	98.55%
Trojan	68.51%	92.47%	96.66%
Virus	78.27%	92.47%	88.39%

model. Figure 6.3a shows an example of a malware converted to image, while Figure 6.3b shows the malware with the binary mask applied, representing the extracted signatures. If the extracted signatures capture the core detection logic, the CNN should still classify these reduced samples as malicious. Table 6.1 shows the result of this test. Between 83% and 100% of extracted binary patterns are still classified as malicious from the CNN. In particular, it can be seen that Scarewares' signatures are malicious in the 100% of the tests. This could mean that extracting signatures from this family is easier than for the other families.

Test 2 - Signature injection into goodware. To further validate the quality of the extracted signatures, we provide another test. For each malware, the extracted signatures are injected in benign file: the signatures are substituted to the bytes in the same position as the original malware, while the rest of the goodware bytes are kept unchanged. These modified goodware samples are then converted into PNG (as can be seen in Figure 6.3e) and then is evaluated by the CNN mimicking model. If the signatures truly encode malicious patterns, the bytes substitution should significantly increase the maliciousness score, compared to the original goodware (whose PNG version can be seen in Figure 6.3d).



(a) Malware (b) Masked malware (c) Malware with inverted mask

(d) Goodware (e) Goodware with signatures

Figure 6.3: Visual examples of malware, masked malware, goodware and goodware with signatures employed in the tests.

This experiment demonstrates that maliciousness is in the extracted signatures, rather than by the presence of zero-padding or unrelated artifacts. For every malware, a random goodware sample is selected, to reduce bias that could rise from the usage of the same goodware. Table 6.2 shows the results of this test. While we have percentages that are lower with respect to the previous test, it is important to say that when the goodware with the injected signatures is not seen as malicious by the CNN, the output score is always higher that the output score of the original goodware.

Outcome interpretation. Together, these tests demonstrate that the extracted signatures actively encode malicious semantics learned by the mimicked antivirus.

6.4. Signature Necessity

The next experiments evaluate whether the extracted signatures are necessary for malicious classification.

Test 1 - Malware without signatures. For each malware sample, the extracted signatures are removed by setting them to zero, while the rest of the file is preserved. The resulting binaries are evaluated by the CNN mimicking model. An example of malware

Table 6.3: Percentage of malware with removed signatures (that is, masked with $1-\texttt{mask}$) classified as malware.

	Defender	ClamAV	Avast
Backdoor	17.50%	3.85%	15.52%
Worm	15.84%	0.74%	10.87%
Scareware	1.79%	0.00%	0.55%
Trojan	10.84%	1.65%	3.24%
Virus	21.73%	9.53%	16.08%

Table 6.4: Percentage of malware with random bytes removed still classified as malware.

	Defender	ClamAV	Avast
Backdoor	75.83%	69.60%	74.13%
Worm	96.69%	81.14%	88.42%
Scareware	69.36%	65.85%	71.67%
Trojan	76.47%	72.50%	68.71%
Virus	65.74%	69.87%	75.89%

with the applied inverted mask is provided in Figure 6.3c. If the signatures capture the main detection logic, the malicious score should drop significantly, potentially below the classification threshold. Table 6.3 presents the result of this test, showing that without the extracted bytes patterns the malware no longer triggers the mimicked antivirus decision. We have percentages of maliciousness that go from 0% to 17.5%. In these results, it can be seen how ClamAV has lower percentages: this could be related to the fact that ClamAV is a signature-based antivirus, and for this reason it is very sensible to malicious bytes removal.

Test 2 - Random bytes removal. To exclude the possibility that the effect is caused simply by removing some bytes, a control experiment is performed. First of all, for every family and for every antivirus, we computed the average fraction of bytes that are contained in extracted signatures. After that, we randomly removed from each sample the same number of bytes but in random positions, ensuring that the removed bytes do not overlap with the extracted signatures bytes. These modified samples are converted in PNG images and evaluated by the CNN. If they remain malicious, it indicates that the drop

	Similarity Score
Backdoor	83.66%
Worm	82.31%
Scareware	87.48%
Trojan	85.15%
Virus	89.41%

Table 6.5: Average similarity score between extracted signatures and best match in ClamAV database.

observed in the previous test is not due to random malware corruption, but specifically due to the removal of meaningful signature bytes. The results of this experiments, shown in Table 6.4. Percentages here range between 65% and 96%, a much higher values with respect to previous test.

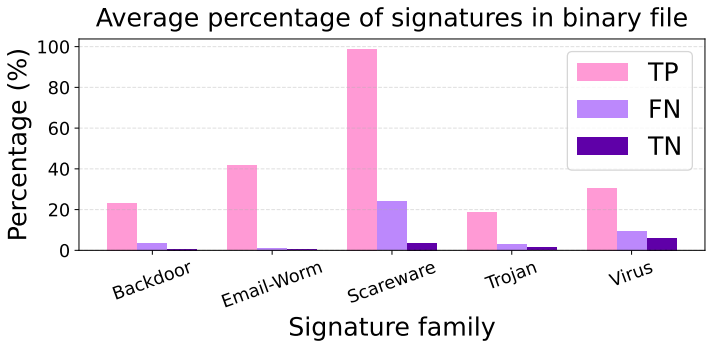
Outcome interpretation. These experiments demonstrate that the extracted signatures are not arbitrary: removing them specifically breaks the malicious classification, while random perturbations do not.

6.5. Real-World Consistency

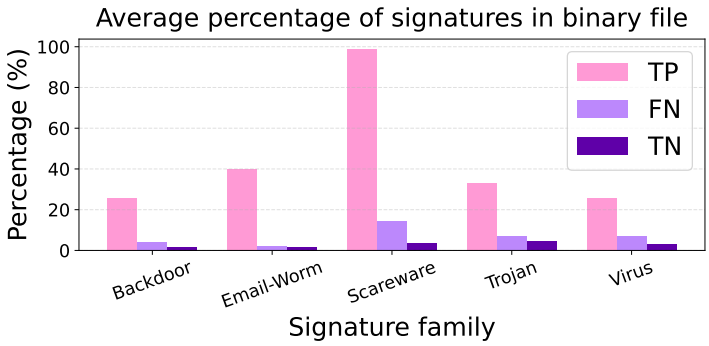
This set of experiments evaluates whether the extracted signatures reflect real antivirus behavior beyond the mimicking model.

Test 1 - Coverage against ClamAV signatures. For ClamAV, which exposes its signature database, we compared the extracted signatures against the real-world signatures used by the antivirus. The goal is to measure how much ClamAV's signatures are similar to the signatures extracted by DAMANI using the CNN that mimics ClamAV. To do this, for each one of the five selected malware families, we took all extracted signatures. Then, for every extracted signature we searched for the database signature with the lower distance. To calculate the distances between signatures, we used the Levenshtein distance [46] on optimal substrings (called `partial-ratio` distance, provided by `rapidfuzz` Python library [5]). Unlike traditional Levenshtein distance, which compares entire sequences and compute the minimum number of insertions, deletions, or substitutions needed to transform one sequence into another, the partial ratio focuses on the most similar subsequence. This distance returns the lowest Levenshtein distance on substrings

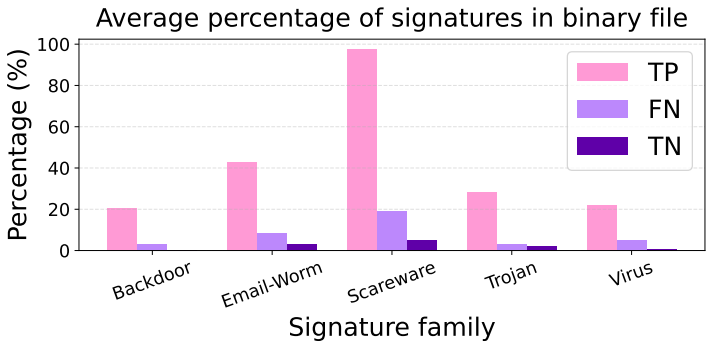
within a longer string. This makes it particularly effective when comparing strings of different lengths. By identifying the substring that maximizes similarity with the target string, the partial ratio provides a normalized *similarity score*, ranging from 0% to 100%.



(a) Windows Defender



(b) ClamAV



(c) Avast Premium Security

Figure 6.4: Average percentage of extracted signatures present in each binary file, considering True Positives and False Negatives of each malware family, and all True Negatives.

For all the signatures, we searched a match in the ClamAV database by computing the similarity scores, and retaining the highest score. The results of this test are shown in Table 6.5. On average, the signatures extracted by DAMANI had an almost exact match in ClamAV signature database.

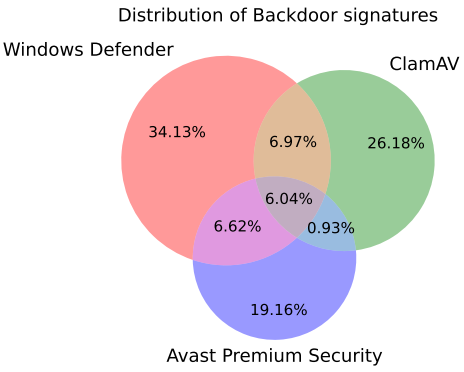
This experiment acts as a sanity check, providing external validation that the extracted signatures are similar to the only white-box detection logic, and are not artifacts of the CNN alone. We did not evaluate the coverage of ClamAV's signature database against our extracted signatures, since the latter are extracted from only five malware families, whereas ClamAV database contains signatures for a much broader set of families. As a result, the opposite test would not provide a meaningful assessment of our extraction capability.

Test 2 - Signature presence across TP, FN, and TN. Finally, we searched extracted signatures inside different antivirus classification groups:

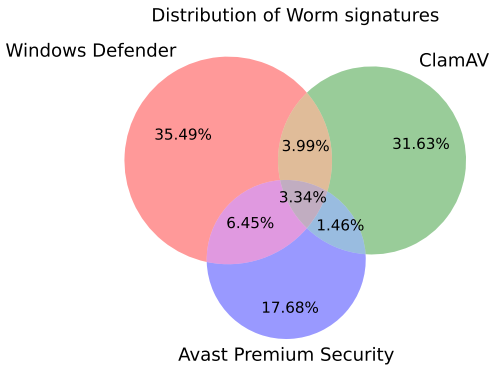
- True Positives (TP), are malware samples identified by the antivirus as malware;
- False Negatives (FN), are malware samples not identified as malicious by the antivirus;
- True Negatives (TN), are goodware samples not identified as malicious by the antivirus.

If the framework truly mimics antivirus behavior, signatures should appear more frequently in TP samples, less frequently in FN samples and in TN samples. In fact, a malware that does not contain one of antivirus' signatures, cannot be identified as malicious by the antivirus, while a malware that contains one or more of antivirus' signatures is automatically identified as malicious. We subdivided TP and FN in families, according to the classification employed in the previous sections, and we searched the signatures extracted by DAMANI in each binary file. The results are shown in Figure 6.4. This distribution shows that the framework is capturing antivirus-specific decision logic rather than simply learning intrinsic malware properties. In particular, the low presence of signatures in FN cases helps illustrate that the framework models the antivirus mistakes as well. The high number of matches in Scareware binaries shows that code reutilization is frequent in this malware family. Likewise, the low number of matches in Backdoor and Virus binaries suggests that there is a high variance between the files.

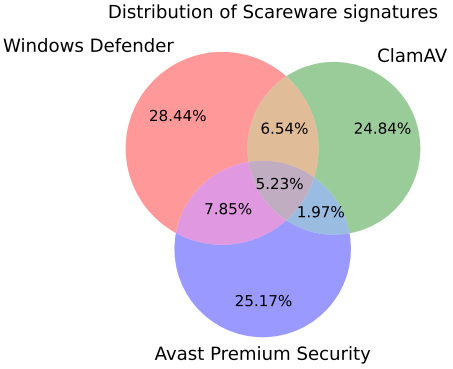
Outcome interpretation. This experiment validates that DAMANI extracts patterns consistent with real antivirus engines and their misclassification behavior, reinforcing the credibility of the previous experimental results.



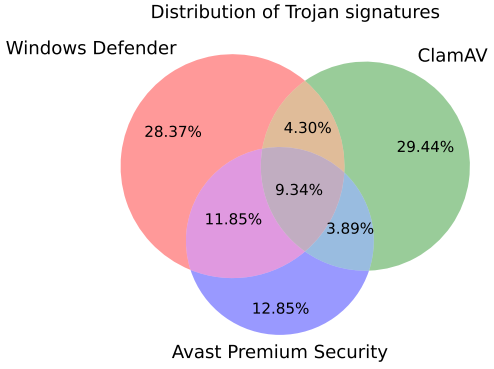
(a) Backdoor



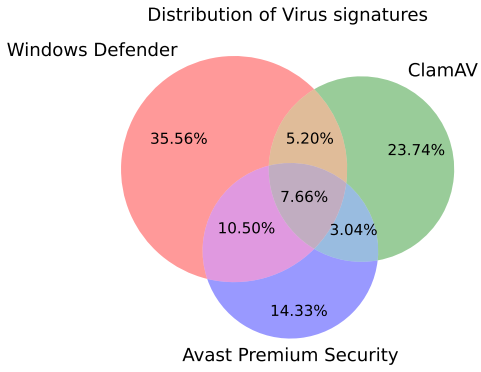
(b) Worm



(c) Scareware



(d) Trojan



(e) Virus

Figure 6.5: Distribution of the extracted signatures for the three antivirus software Windows Defender, ClamAV, and Avast Premium Security.

6.6. Extracted Signatures Analysis

In this section, we provide considerations about the signatures extracted by DAMANI.

6.6.1. Common Signatures Analysis

To verify the existence of malicious byte patterns in common between different antivirus software, we first selected the 20% of extracted signatures for each antivirus appearing in the largest number of malware files, allowing us to only consider the most significant signatures, while maintaining a coherent proportion of each dataset. Figure 6.5 shows the distribution of the extracted signatures across Windows Defender, ClamAV, and Avast Premium Security. The first note is on the absolute number of extracted signatures, where we can see that DAMANI extracted a large number of signatures from Windows Defender. This phenomenon is due to the complex classification criterion of Windows Defender [54], requiring a larger number of extracted signatures in order to reliably capture its decision boundaries. Another fact worth noticing is that the intersection between Windows Defender and Avast Premium Security is the largest among the three antivirus. This aligns with the target environments of the three software, with the ones of Windows Defender and Avast Premium Security being more similar to each other with respect to ClamAV. Both Defender and Avast are modern commercial antivirus systems optimized for broad, real-time protection against prevalent Windows malware, employing both static and dynamic analysis. By contrast, ClamAV is an open-source antivirus that relies primarily on a static, community-maintained signature database, without the same analysis depth found in commercial products.

6.6.2. Signatures Disassembling

After identifying the most relevant signatures extracted by DAMANI, we employed a disassemblation routine, wrote in Python using the `capstone` library, to analyze the semantic functionality of the extracted byte patterns. It is important to note that, while these observations provide additional context, they do not constitute a comprehensive analysis. These insights offer a glimpse into the types of program fragments that antivirus engines may target, but do not represent a full semantic evaluation of the code. Further in-depth analysis would be necessary to fully understand the implications and functionality of these code snippets.

We now provide three examples of disassembled signatures.

Example 1: String extracted from Worms

```
0x73 0x73 0x61 0x67 0x65 0x20 0x77 0x61 0x73 0x20 0x69 0x6e 0x63 0x6c 0x75 0x64
0x65 0x64 0x20 0x61 0x73 0x20 0x61 0x74 0x74 0x61 0x63 0x68 0x6d 0x65 0x6e 0x74
0x00 0x54 0x68 0x69 0x73 0x20 0x4d 0x65 0x73 0x73 0x61 0x67 0x65 0x20 0x77 0x61
0x73 0x20 0x75 0x6e 0x64 0x65 0x6c 0x69 0x76 0x65 0x72 0x61 0x62 0x6c 0x65 0x20
0x64 0x75 0x65 0x20 0x74 0x6f 0x20 0x74 0x68 0x65 0x20 0x66 0x6f 0x6c 0x6c 0x6f
0x77 0x69 0x6e 0x67 0x20 0x72 0x65 0x61 0x73 0x6f 0x6e 0x3a 0x0d 0x0a

"ssage was included as attachment. This Message was undeliverable due to the
↪  following reason:\r\n"
```

This was a string extracted by Worms. Indeed, Email-Worms usually hardcode this kind of messages for phishing attacks.

Example 2: Evasion Routine

```
0x83 0xec 0x28 0xe8 0x85 0x00 0x00 0x00 0xc7 0x44 0x24 0x14 0x00 0x00 0x00 0x00
0xc7 0x44 0x24 0x10 0x00 0x00 0x00 0x00 0xc7 0x44 0x24 0x0c 0x00 0x00 0x00 0x00
0xc7 0x44 0x24 0x08 0xc2 0x16 0x40 0x00 0xc7 0x44 0x24 0x04 0x00 0x00 0x00 0x00
0xc7 0x04 0x24 0x00 0x00 0x00 0x00 0xe8 0x91 0x10 0x00 0x00 0x83 0xec 0x18 0xc7
0x04 0x24 0x60 0xea 0x00 0x00 0xe8 0x92 0x10 0x00 0x00 0x83 0xec 0x04 0xe8 0xda
0x0f 0x00 0x00 0x89 0xc1 0xb8 0xe9 0xa2 0x8b 0x2e 0xf7 0xe9 0xd1 0xfa 0x89 0xc8
0xc1 0xf8 0x1f 0x29 0xc2 0x89 0xd0 0x89 0x45 0xfc 0x8b 0x55 0xfc 0x89 0xd0 0xc1
0xe0 0x02 0x01 0xd0 0x01 0xc0 0x01 0xd0 0x29 0xc1 0x89 0xc8 0x89 0x45 0xfc 0x8b
0x45 0xfc 0x89 0x04 0x24 0xe8 0xf9 0x04 0x00 0x00 0xeb 0xb3 0x90 0x55 0x89 0xe5
0x83 0xec 0x28 0xc7 0x45 0xf8 0x00 0x00 0x00 0x00 0xc7 0x04 0x24 0x00 0x00 0x00
0x00 0xe8 0x47 0x10 0x00 0x00 0x83 0xec 0x04 0xc7 0x44 0x24 0x08 0x04 0x01 0x00
0x00 0xc7 0x44 0x24 0x04 0x30 0x50 0x40 0x00 0x89 0x04 0x24 0xe8 0x3c 0x10 0x00
0x00 0x83 0xec 0x0c 0xc7 0x44 0x24 0x04 0xbe 0x40 0x40 0x00 0xc7 0x04 0x24 0x30
0x50 0x40 0x00 0xe8 0x65 0x0f 0x00 0x00 0xa3 0x14 0x50 0x40 0x00 0x83 0x3d 0x14
0x50 0x40 0x00 0x00 0x75 0x0c 0xc7 0x04 0x24 0x00 0x00 0x00 0x00 0xe8 0x2b 0x0f
0x00 0x00 0xc7 0x44 0x24
```

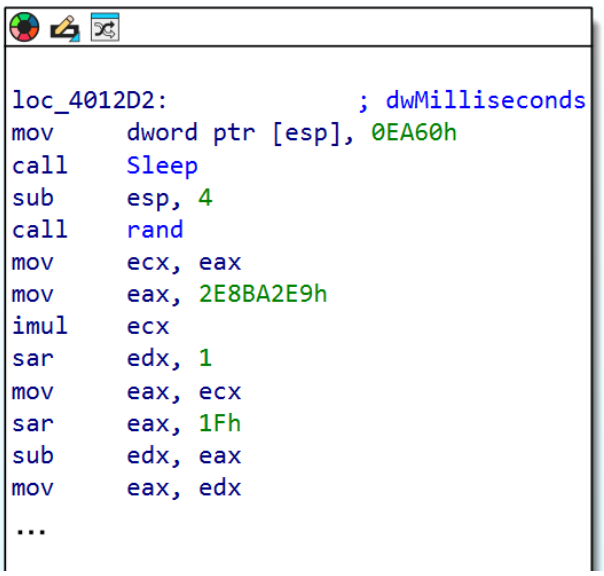
```
0x401293:  sub  esp, 0x28;  bytes: 0x83 0xec 0x28
0x401296:  call  0x401320;  bytes: 0xe8 0x85 0x00 0x00 0x00
0x40129b:  mov  dword ptr [esp + 0x14], 0x00;  bytes: 0xc7 0x44 0x24 0x14 0x00
↪  0x00 0x00 0x00
0x4012a3:  mov  dword ptr [esp + 0x10], 0x00;  bytes: 0xc7 0x44 0x24 0x10 0x00
↪  0x00 0x00 0x00
0x4012ab:  mov  dword ptr [esp + 0xc], 0x00;  bytes: 0xc7 0x44 0x24 0x0c 0x00
↪  0x00 0x00 0x00
```

```
0x4012b3:  mov  dword ptr [esp + 0x8], 0x4016c2;  bytes: 0xc7 0x44 0x24 0x08
↪  0xc2 0x16 0x40 0x00
0x4012bb:  mov  dword ptr [esp + 0x4], 0x00;  bytes: 0xc7 0x44 0x24 0x04 0x00
↪  0x00 0x00 0x00
0x4012c3:  mov  dword ptr [esp], 0x00;  bytes: 0xc7 0x04 0x24 0x00 0x00 0x00
↪  0x00
0x4012ca:  call  0x402360;  bytes: 0xe8 0x91 0x10 0x00 0x00
0x4012cf:  sub  esp, 0x18;  bytes: 0x83 0xec 0x18
0x4012d2:  mov  dword ptr [esp], 0xea60;  bytes: 0xc7 0x04 0x24 0x60 0xea 0x00
↪  0x00
0x4012d9:  call  0x402370;  bytes: 0xe8 0x92 0x10 0x00 0x00
0x4012de:  sub  esp, 0x4;  bytes: 0x83 0xec 0x4
0x4012e1:  call  0x4022c0;  bytes: 0xe8 0xda 0x0f 0x00 0x00
0x4012e6:  mov  ecx, eax;  bytes: 0x89 0xc1
0x4012e8:  mov  eax, 0x2e8ba2e9;  bytes: 0xb8 0xe9 0xa2 0x8b 0x2e
0x4012ed:  imul  ecx;  bytes: 0xf7 0xe9
0x4012ef:  sar  edx, 0x1;  bytes: 0xd1 0xfa
0x4012f1:  mov  eax, ecx;  bytes: 0x89 0xc8
0x4012f3:  sar  eax, 0x1f;  bytes: 0xc1 0xf8 0x1f
0x4012f6:  sub  edx, eax;  bytes: 0x29 0xc2
0x4012f8:  mov  eax, edx;  bytes: 0x89 0xd0
0x4012fa:  mov  dword ptr [ebp - 0x4], eax;  bytes: 0x89 0x45 0xfc
0x4012fd:  mov  edx, dword ptr [ebp - 0x4];  bytes: 0x8b 0x55 0xfc
0x401300:  mov  eax, edx;  bytes: 0x89 0xd0
0x401302:  shl  eax, 0x2;  bytes: 0xc1 0xe0 0x2
0x401305:  add  eax, edx;  bytes: 0x01 0xd0
0x401307:  add  eax, eax;  bytes: 0x01 0xc0
0x401309:  add  eax, edx;  bytes: 0x01 0xd0
0x40130b:  sub  ecx, eax;  bytes: 0x29 0xc1
0x40130d:  mov  eax, ecx;  bytes: 0x89 0xc8
0x40130f:  mov  dword ptr [ebp - 0x4], eax;  bytes: 0x89 0x45 0xfc
0x401312:  mov  eax, dword ptr [ebp - 0x4];  bytes: 0x8b 0x45 0xfc
0x401315:  mov  dword ptr [esp], eax;  bytes: 0x89 0x04 0x24
0x401318:  call  0x401816;  bytes: 0xe8 0xf9 0x04 0x00 0x00
0x40131d:  jmp  0x4012d2;  bytes: 0xeb 0xb3
0x40131f:  nop  ;  bytes: 0x90
0x401320:  push  ebp;  bytes: 0x55
0x401321:  mov  ebp, esp;  bytes: 0x89 0xe5
0x401323:  sub  esp, 0x28;  bytes: 0x83 0xec 0x28
```

```
0x401326:  mov  dword ptr [ebp - 0x8], 0x00;  bytes: 0xc7 0x45 0xf8 0x00 0x00
↪  0x00 0x00
0x40132d:  mov  dword ptr [esp], 0x00;  bytes: 0xc7 0x04 0x24 0x00 0x00 0x00
↪  0x00
0x401334:  call  0x402380;  bytes: 0xe8 0x47 0x10 0x00 0x00
0x401339:  sub  esp, 0x4;  bytes: 0x83 0xec 0x4
0x40133c:  mov  dword ptr [esp + 0x8], 0x104;  bytes: 0xc7 0x44 0x24 0x08 0x04
↪  0x01 0x00 0x00
0x401344:  mov  dword ptr [esp + 0x4], 0x405030;  bytes: 0xc7 0x44 0x24 0x04
↪  0x30 0x50 0x40 0x00
0x40134c:  mov  dword ptr [esp], eax;  bytes: 0x89 0x04 0x24
0x40134f:  call  0x402390;  bytes: 0xe8 0x3c 0x10 0x00 0x00
0x401354:  sub  esp, 0xc;  bytes: 0x83 0xec 0xc
0x401357:  mov  dword ptr [esp + 0x4], 0x4040be;  bytes: 0xc7 0x44 0x24 0x04
↪  0xbe 0x40 0x40 0x00
0x40135f:  mov  dword ptr [esp], 0x405030;  bytes: 0xc7 0x04 0x24 0x30 0x50
↪  0x40 0x00
0x401366:  call  0x4022d0;  bytes: 0xe8 0x65 0x0f 0x00 0x00
0x40136b:  mov  dword ptr [0x405014], eax;  bytes: 0xa3 0x14 0x50 0x40 0x00
0x401370:  cmp  dword ptr [0x405014], 0x00;  bytes: 0x83 0x3d 0x14 0x50 0x40
↪  0x00 0x00
0x401377:  jne  0x401385;  bytes: 0x75 0xc
0x401379:  mov  dword ptr [esp], 0x00;  bytes: 0xc7 0x04 0x24 0x00 0x00 0x00
↪  0x00
0x401380:  call  0x4022b0;  bytes: 0xe8 0x2b 0x0f 0x00 0x00
0x401385:  mov  dword ptr [esp + 0x8], 0x2;  bytes: 0xc7 0x44 0x24 0x08 0x02
↪  0x00 0x00 0x00
```

This fragment, extracted from Viruses, seems to implement a periodic worker loop: the program allocates stack space, initializes parameters, and invokes a function before entering an infinite cycle. Inside the loop, the instruction `mov dword ptr [esp], 0xea60` followed by a call could indicate an evasion technique. We disassembled the entire malware using IDA Free [2] and confirmed that the function at `0x402370` is in fact a `Sleep`, as can be seen in Figure 6.6. This means that the program calls `Sleep(60)`, which is frequently used by malware software as evasion technique, in particular this technique is called "Time-based Evasion" and is used to evade dynamic analysis.

Additionally, the initialization routine (starting at `0x401320`) implements a simple self-extracting dropper behavior. It prepares a buffer of size `0x104` (260) bytes, corresponding to the Windows constant `MAX_PATH`, and passes it to a function (`GetModuleFileNameA`,



```
loc_4012D2:                ; dwMilliseconds
mov     dword ptr [esp], 0EA60h
call    Sleep
sub     esp, 4
call    rand
mov     ecx, eax
mov     eax, 2E8BA2E9h
imul    ecx
sar     edx, 1
mov     eax, ecx
sar     eax, 1Fh
sub     edx, eax
mov     eax, edx
...
```

Figure 6.6: Screen of assembly code returned by IDA Free

address `0x402390`) that retrieves the path of its own executable file and opens it. Looking at the disassembled code returned by IDA Free, we saw that these instructions are part of a function that writes some of the program bytes into a new file named `\exc.exe` and executes it using `ShellExecuteA`. Essentially, the program contains a secondary executable appended to itself and extracts and runs it at runtime.

Example 3: File Infector Routine

```
0xe8 0xa2 0x0e 0x00 0x00 0xa3 0x34 0x51 0x40 0x00 0xa1 0x14 0x50 0x40 0x00 0x89
0x44 0x24 0x0c 0xc7 0x44 0x24 0x08 0x01 0x00 0x00 0x00 0xa1 0x3c 0x51 0x40 0x00
0x89 0x44 0x24 0x04 0xa1 0x34 0x51 0x40 0x00 0x89 0x04 0x24 0xe8 0x66 0x0e 0x00
0x00 0xa1 0x14 0x50 0x40 0x00 0x89 0x04 0x24 0xe8 0x49 0x0e 0x00 0x00 0xc7 0x45
0xfc 0x00 0x00 0x00 0x00 0x8b 0x45 0xfc 0x3b 0x05 0x3c 0x51 0x40 0x00 0x7d 0x45
0xa1 0x34 0x51 0x40 0x00 0x03 0x45 0xfc 0x80 0x38 0x2a 0x75 0x31 0x8b 0x45 0xfc
0x03 0x05 0x34 0x51 0x40 0x00 0x40 0x80 0x38 0x2a 0x75 0x22 0x8b 0x45 0xfc 0x03
0x05 0x34 0x51 0x40 0x00 0x83 0xc0 0x02 0x80 0x38 0x2a 0x75 0x11 0x8b 0x45 0xfc
0x03 0x05 0x34 0x51 0x40 0x00 0x83 0xc0 0x03 0x89 0x45 0xf8 0xeb 0x07 0x8d
```

```
0x4013d9:  call  0x402280;  bytes: 0xe8 0xa2 0x0e 0x00 0x00
0x4013de:  mov  dword ptr [0x405134], eax;  bytes: 0xa3 0x34 0x51 0x40 0x00
0x4013e3:  mov  eax, dword ptr [0x405014];  bytes: 0xa1 0x14 0x50 0x40 0x00
0x4013e8:  mov  dword ptr [esp + 0xc], eax;  bytes: 0x89 0x44 0x24 0x0c
0x4013ec:  mov  dword ptr [esp + 8], 1;  bytes: 0xc7 0x44 0x24 0x08 0x01 0x00
↪  0x00 0x00
0x4013f4:  mov  eax, dword ptr [0x40513c];  bytes: 0xa1 0x3c 0x51 0x40 0x00
```

```
0x4013f9:  mov  dword ptr [esp + 4], eax;  bytes: 0x89 0x44 0x24 0x04
0x4013fd:  mov  eax, dword ptr [0x405134];  bytes: 0xa1 0x34 0x51 0x40 0x00
0x401402:  mov  dword ptr [esp], eax;  bytes: 0x89 0x04 0x24
0x401405:  call  0x402270;  bytes: 0xe8 0x66 0x0e 0x00 0x00
0x40140a:  mov  eax, dword ptr [0x405014];  bytes: 0xa1 0x14 0x50 0x40 0x00
0x40140f:  mov  dword ptr [esp], eax;  bytes: 0x89 0x04 0x24
0x401412:  call  0x402260;  bytes: 0xe8 0x49 0x0e 0x00 0x00
0x401417:  mov  dword ptr [ebp - 4], 0;  bytes: 0xc7 0x45 0xfc 0x00 0x00 0x00
↪  0x00
0x40141e:  mov  eax, dword ptr [ebp - 4];  bytes: 0x8b 0x45 0xfc
0x401421:  cmp  eax, dword ptr [0x40513c];  bytes: 0x3b 0x05 0x3c 0x51 0x40
↪  0x00
0x401427:  jge  0x40146e;  bytes: 0x7d 0x45
0x401429:  mov  eax, dword ptr [0x405134];  bytes: 0xa1 0x34 0x51 0x40 0x00
0x40142e:  add  eax, dword ptr [ebp - 4];  bytes: 0x03 0x45 0xfc
0x401431:  cmp  byte ptr [eax], 0x2a;  bytes: 0x80 0x38 0x2a
0x401434:  jne  0x401467;  bytes: 0x75 0x31
0x401436:  mov  eax, dword ptr [ebp - 4];  bytes: 0x8b 0x45 0xfc
0x401439:  add  eax, dword ptr [0x405134];  bytes: 0x03 0x05 0x34 0x51 0x40
↪  0x00
0x40143f:  inc  eax;  bytes: 0x40
0x401440:  cmp  byte ptr [eax], 0x2a;  bytes: 0x80 0x38 0x2a
0x401443:  jne  0x401467;  bytes: 0x75 0x22
0x401445:  mov  eax, dword ptr [ebp - 4];  bytes: 0x8b 0x45 0xfc
0x401448:  add  eax, dword ptr [0x405134];  bytes: 0x03 0x05 0x34 0x51 0x40
↪  0x00
0x40144e:  add  eax, 2;  bytes: 0x83 0xc0 0x02
0x401451:  cmp  byte ptr [eax], 0x2a;  bytes: 0x80 0x38 0x2a
0x401454:  jne  0x401467;  bytes: 0x75 0x11
0x401456:  mov  eax, dword ptr [ebp - 4];  bytes: 0x8b 0x45 0xfc
0x401459:  add  eax, dword ptr [0x405134];  bytes: 0x03 0x05 0x34 0x51 0x40
↪  0x00
0x40145f:  add  eax, 3;  bytes: 0x83 0xc0 0x03
0x401462:  mov  dword ptr [ebp - 8], eax;  bytes: 0x89 0x45 0xf8
0x401465:  jmp  0x40146e;  bytes: 0xeb 0x07
0x401467:  lea  eax, [ebp - 4];  bytes: 0x8d 0x45 0xfc
```

This signature was extracted from Viruses. The examined assembly fragment performs a sequential scan over a memory buffer. After a series of initialization calls, the program checks for the presence of four consecutive bytes with value `0x2a` (*), and, upon

detecting this sequence, stores the address of the match before terminating the search. This behavior could belong to several categories of routines commonly found in malware, such as infection-marker checks in file infectors, internal marker identification, and loader/unpacker mechanisms. In particular, a sequence like **** is frequently used by various malware families as an internal marker to delimit configuration data, encrypted or compressed payloads, or custom sections appended to a host file. Some file-infecting malware also employ such markers to determine whether a file has already been infected, both to avoid multiple infections, which could increase the probability of detection by the victim, and to locate previously injected code.

6.7. Discussion

This chapter presents the experiments performed on DAMANI's extracted signatures. These experiments evaluate reliability, necessity, and real-world consistency of the extracted signatures.

First, we demonstrate signatures reliability: when malware samples are reduced to the extracted signature regions only, the CNN mimicking model continues to classify them as malicious with very high accuracy. Additionally, injecting the signatures into benign binaries increases their malicious score. These results confirm that the extracted byte patterns actively reflect malicious semantics learned by the mimicking model.

Second, we evaluate signatures necessity. Removing only the extracted signatures' bytes from malware samples leads to a drastic drop in malicious classification, while removing an equivalent number of randomly selected bytes does not produce the same effect. This comparison shows that the extracted regions are not arbitrary portions of the binary: they are responsible for triggering the learned detection logic.

Third, we assess real-world consistency. The comparison with the publicly available signatures database of ClamAV reveals a high similarity between signatures extracted by DAMANI and actual antivirus signatures, providing an external sanity check. Moreover, the distribution of signatures across antivirus' True Positives, False Negatives, and True Negatives shows that DAMANI captures antivirus-specific decision behavior, including misclassification patterns, rather than simply learning malware characteristics.

Finally, we analyze common signatures across different antivirus engines and performed a disassembly of the most frequent patterns. While this does not constitute a full semantic study, it provides additional contextual insight into the nature of the extracted code fragments, supporting the interpretation that DAMANI isolates meaningful byte patterns.

Overall, the experimental results demonstrate that DAMANI is capable of extracting byte-level patterns that are both sufficient and necessary for the mimicked detection process, and that these patterns exhibit measurable consistency with real antivirus logic. This validation establishes the framework as a viable tool for structured analysis of black-box malware detection systems.

7 | Limitations

Although the proposed DAMANI framework demonstrates that it is possible to approximate and analyze static detection patterns used by antivirus engines, we acknowledge some limitations related to the complexity of antivirus classification criterion, and to the intrinsic limitations of deep learning models.

Missing Direct Validation on Antivirus Engines. The extracted signatures are primarily validated through the CNN mimicking models rather than by directly testing them against the real antivirus engines. While the mimicking models are trained to approximate antivirus behavior, they remain an approximation and may not perfectly capture all aspects of the real detection logic, especially when dynamic analysis, heuristics, or contextual information are involved. Directly injecting extracted signatures into real malware and evaluating them against commercial antivirus products would provide stronger validation. However, such experiments are difficult to perform reliably due to the closed-source nature of antivirus software, differences in scanning environments, and safety constraints when manipulating live malware. We perform some preliminary tests using the actual antivirus engines, but the classification assigned to incomplete or modified binaries proved to be unreliable for learning intrinsic decision criterion, causing the usage of a mimicking model to be necessary.

Image Representation of Binaries. In this thesis, malware binaries are converted into images of fixed dimensions in order to be processed by 2D convolutional neural networks. Although PNG encoding itself is lossless, the transformation of binaries into fixed-size images introduces structural distortions. Resizing, and padding, may modify the original file, while the 2D reshaping enforces a spatial organization that does not naturally exist in the sequential byte stream. In particular, bytes that are distant in the original layout may become neighbors in the same column of the image, generating artificial correlations between unrelated regions of the binary. As a result, the CNN may learn patterns that are influenced by the imposed geometry rather than by the true semantics of the executable. Moreover, the image-based representation abstracts away

semantic PE information such as section boundaries, imports, and control-flow structure, which may also contribute to antivirus detection decisions.

Limited Coverage of Our Analysis. The extraction phase focuses on a restricted number of malware families in order to enable stable autoencoder training and family-specific analysis. While this choice improves interpretability, it limits the generality of the extracted signatures and reduces the representativeness of the evaluation with respect to the entire malware ecosystem. Antivirus engines typically rely on signatures spanning thousands of malware families and obfuscation strategies, whereas DAMANI analyzes only a reduced subset of them. As a consequence, the extracted patterns not fully capture the diversity of real-world detection logic. Scaling the framework to larger and more diverse malware collections would allow the extraction of broader detection patterns and enable cross-family analysis. This would require larger datasets, improved clustering and family assignment strategies, and increased computational resources. Moreover, DAMANI was evaluated using three representative antivirus engines, but other commercial and open-source solutions exist in practice. Extending the analysis to a wider range of antivirus products, despite increasing the computational workload, would provide a more comprehensive view of common detection strategies and would increase the information that DAMANI is able to extract.

Framework Limitations. DAMANI is designed and evaluated exclusively on Windows Portable Executable (PE) files. Other malware formats such as scripts, documents, archives, ELF binaries, or mobile applications are not considered. The framework has not been used on non-PE malware, and the analysis performed in this thesis only reflects the detection logic applied to Windows executables.

The autoencoder operates on fixed-size image representations with a relatively limited resolution (e.g., 300×300 bytes). As a result, only a portion of large binaries can be processed. This limits the applicability of the framework to very large executables and may bias extraction toward early or resized portions of the binary.

Training multiple CNNs and autoencoders for each antivirus and malware family is computationally expensive and may limit scalability when extending the framework to many engines or very large datasets.

8 | Future Works

The DAMANI framework has demonstrated the feasibility of approximating and interpreting the static detection logic of antivirus engines through deep learning. However, as discussed in the limitations, several challenges remain regarding validation, scalability, and representation. Future research directions aim to address these constraints to enhance the framework's robustness, versatility, and real-world applicability.

Bridge the Gap with Antivirus. A primary limitation of the current study is the reliance on mimicking models for validation, as direct tests against commercial antivirus engines yielded suboptimal results (we provide further information in Appendix A). Future works could find a way to interact directly with the antivirus engine during the signature extraction, allowing the framework to learn directly from the antivirus instead of a mimicking model.

Binary Representation Change. The conversion of binaries into fixed-dimension 2D images can introduce structural distortions and artificial spatial correlations. To mitigate this, future research should explore sequential and semantic-aware architectures.

- **1D-CNN:** One Dimensional Convolutional Neural Networks parse the input file as a raw byte sequence, without the need for a 2D spatial dimension. This family of models, when applied to medium-large binary files, requires complex training;
- **Transformers for Binaries:** Adopting architecture similar to BERT (Bidirectional Encoder Representations from Transformers) or GPT (Generative Pre-trained Transformer) process binaries as sequences of bytes would capture long-range dependencies, and the true sequential nature of the code without imposing an artificial 2D grid structure;
- **Graph Neural Networks (GNNs):** Instead of treating the binary as a raw byte stream, extracting the Control Flow Graph (CFG) or Function Call Graph (FCG) and processing it with GNNs would preserve the semantic structure of the program, making the analysis robust against layout randomization and padding.

More Malware Families and More Antivirus Engines. The current analysis is limited to a subset of five malware families and three antivirus engines. Scaling the framework is essential for generalization:

- **Dataset Integration:** Future deployments should utilize larger or more specific repositories to cover other malware families, including rare or highly obfuscated variants;
- **Multi-Engine Correlation:** By training on a wider array of commercial and open-source engines, the framework could be used to identify patterns detected by multiple engines, offering deeper insights into the proprietary heuristics of different security companies;
- **Same-Engine Changes:** another promising direction concerns the temporal evolution of antivirus signatures. Antivirus engines are continuously updated, and detection logic changes over time. Analyzing how extracted signatures evolve across different versions of the same antivirus would provide insight into signature robustness, update strategies, and the stability of static detection mechanisms.

Support for Different Extensions. Since DAMANI is currently restricted to Windows PE files, extending the pipeline to support ELF (Linux), Mach-O (macOS), and other formats would provide a more generalized framework.

DAMANI for Adversarial Malware Generation. Finally, DAMANI could be extended toward automatic adversarial malware generation, where extracted signatures are automatically obfuscated in malware to minimize detectability while preserving functionality. Such a framework would allow the study of antivirus resilience and the development of stronger defensive mechanisms against adversarial evasion.

9 | Conclusions

The goal of this thesis was to investigate whether it is possible to approximate and analyze the static detection logic of antivirus software by extracting representative byte-level signatures from black-box engines. Rather than designing a new malware detector, the focus was on understanding and interpreting the patterns that real antivirus products use to classify binaries as malicious.

To achieve this objective, we proposed DAMANI, a framework that treats antivirus engines as black-box classifiers and uses a deep learning model to mimic their behavior. Portable Executable binaries were collected from public repositories, labeled using multiple antivirus engines, and transformed into image representations suitable for a Two-Dimensional Convolutional Neural Network. This deep learning model was trained to reproduce the antivirus decisions, while a Convolutional Autoencoder was employed to extract the regions of the input that preserve malicious classifications. These regions were post-processed to obtain signature-like byte sequences, which were then analyzed across malware families and across different antivirus engines.

Experimental evaluation demonstrated that the extracted signatures are not random substrings, but contribute to the mimicked antivirus decisions. Malware samples composed only of the extracted regions remain classified as malicious by the mimicking model, whereas removing those regions significantly reduces the maliciousness score. Additional experiments showed that the presence of signatures correlates with true positive detections rather than generic malware classification, confirming that DAMANI captures antivirus-specific behavior. In the end, cross-engine analysis revealed both shared and engine-specific detection patterns.

Looking forward, signature extraction frameworks such as DAMANI can become valuable tools for researchers. They may help assess the robustness of antivirus detection logic, compare different engines at a semantic level, or study how detection strategies evolve over time. In a broader perspective, combining black-box learning, explainability, and cross-engine analysis opens the door to more transparent and interpretable security systems.

Bibliography

[1] Google Colab. URL https://colab.google/.

[2] Ida Free. URL https://hex-rays.com/ida-free.

[3] Desktop Operating System Market Share Worldwide, 2025. URL https://gs.statcounter.com/os-market-share/desktop/worldwide.

[4] Windows users, beware: 7× more malware than macOS, 2025. URL https://surfshark.com/research/chart/malware-cases-windows-macOS.

[5] RapidFuzz, 2026. URL https://doi.org/10.5281/zenodo.13245733.

[6] Softonic, 2026. URL https://it.softonic.com/.

[7] TensorFlow Guide, 2026. URL https://www.tensorflow.org/guide/keras/serialization_and_saving?utm_source=chatgpt.com.

[8] VirtualBox, 2026. URL https://www.virtualbox.org/.

[9] Yara: The pattern matching swiss knife for malware researchers, 2026. URL https://virustotal.github.io/yara/.

[10] M. S. Akhtar and T. Feng. Malware Analysis and Detection Using Machine Learning Algorithms. *Symmetry*, 14(11):2304, Nov. 2022. ISSN 2073-8994. doi: 10.3390/sym14112304. URL https://www.mdpi.com/2073-8994/14/11/2304. Number: 11 Publisher: Multidisciplinary Digital Publishing Institute.

[11] M. Alasli and T. Ghaleb. *Review of Signature-based Techniques in Antivirus Products*. Apr. 2019. doi: 10.1109/ICCISci.2019.8716381. Pages: 6.

[12] N. Alshahwan, E. T. Barr, D. Clark, G. Danezis, and H. D. Menéndez. Detecting Malware with Information Complexity. *Entropy*, 22(5):575, May 2020. ISSN 1099-4300. doi: 10.3390/e22050575. URL https://www.mdpi.com/1099-4300/22/5/575. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

[13] Avast Software. AI and Machine Learning in Avast Technology, 2024. URL https://www.avast.com/en-eu/technology/ai-and-machine-learning.

[14] Avast Software. Avast Premium Security, 2026. URL `https://www.avast.com/it-it/premium-security`.

[15] J. Aycock. *Computer Viruses and Malware*. Springer Science & Business Media, 2006.

[16] H. Benbrahim and A. Behloul. Malware Classification on Malimg Using MobileNet and LSTM for Efficient Detection. In *2024 1st International Conference on Innovative and Intelligent Information Technologies (IC3IT)*, pages 1–6, Dec. 2024. doi: 10.1109/IC3IT63743.2024.10869415. URL `https://ieeexplore.ieee.org/abstract/document/10869415`.

[17] A. Bensaoud, J. Kalita, and M. Bensaoud. A survey of malware detection using deep learning. *Machine Learning with Applications*, 16:100546, June 2024. ISSN 2666-8270. doi: 10.1016/j.mlwa.2024.100546. URL `https://www.sciencedirect.com/science/article/pii/S2666827024000227`.

[18] M. Botacin, F. D. Domingues, F. Ceschin, R. Machnicki, M. A. Zanata Alves, P. L. De Geus, and A. Grégio. AntiViruses under the microscope: A hands-on perspective. *Computers & Security*, 112:102500, Jan. 2022. ISSN 01674048. doi: 10.1016/j.cose.2021.102500. URL `https://linkinghub.elsevier.com/retrieve/pii/S0167404821003242`.

[19] A. Calleja, J. Tapiador, and J. Caballero. The malsource dataset: Quantifying complexity and code reuse in malware development. *IEEE Transactions on Information Forensics and Security*, 14(12):3175–3190, 2019. doi: 10.1109/TIFS.2018.2885512.

[20] J. Chandy. Review on Malware, Types, and its Analysis. *International Journal for Research in Applied Science and Engineering Technology*, 10(12):386–390, Dec. 2022. ISSN 23219653. doi: 10.22214/ijraset.2022.47887. URL `https://www.ijraset.com/best-journal/review-on-malware-types-and-its-analysis`.

[21] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and D. Chen. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

[22] Cisco Talos. ClamAV: Open Source Antivirus Engine, 2025. URL `https://www.clamav.net/`.

[23] Cisco Talos. ClamAV Documentation, 2025. URL `https://docs.clamav.net/`.

[24] A. Cletus, A. A. Opoku, and B. A. Weyori. An Evaluation of Current Malware Trends and Defense Techniques: A Scoping Review with Empirical Case Studies. *Journal*

of Advances in Information Technology, pages 649–671, 2024. ISSN 17982340. doi: 10.12720/jait.15.5.649-671.

[25] L. Demetrio, B. Biggio, and F. Roli. Practical Attacks on Machine Learning: A Case Study on Adversarial Windows Malware. *IEEE Security & Privacy*, 20(5):77–85, Sept. 2022. ISSN 1540-7993, 1558-4046. doi: 10.1109/MSEC.2022.3182356. URL `http://arxiv.org/abs/2207.05548`. arXiv:2207.05548 [cs].

[26] G. Digregorio, S. Maccarrone, M. D'Onghia, L. Gallo, M. Carminati, M. Polino, and S. Zanero. Tarallo: Evading Behavioral Malware Detectors in the Problem Space. In F. Maggi, M. Egele, M. Payer, and M. Carminati, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 128–149, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-64171-8. doi: 10.1007/978-3-031-64171-8_7.

[27] W. Fleshman, E. Raff, R. Zak, M. McLean, and C. Nicholas. Static Malware Detection & Subterfuge: Quantifying the Robustness of Machine Learning and Current Anti-Virus. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–10, Nantucket, MA, USA, Oct. 2018. IEEE. ISBN 978-1-7281-0155-2. doi: 10.1109/MALWARE.2018.8659360. URL `https://ieeexplore.ieee.org/document/8659360/`.

[28] Gen Digital. Avast Announces Agreement to Acquire AVG for $1.3B, July 2016. URL `https://newsroom.gendigital.com/2016-07-07-avast-announces-agreement-to-acquire-avg-for-1-3b`.

[29] Gen Digital. Norton Antivirus Solutions, 2026. URL `https://it.norton.com/`.

[30] M. D. Gloria. ANNtivirus: dissecting Antivirus programs through Neural Network explainability.

[31] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[32] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. doi: 10.4258/hir.2016.22.4.351.

[33] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi. A survey of methods for explaining black box models. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[34] B. T. Hammad, N. Jamil, I. T. Ahmed, Z. M. Zain, and S. Basheer. Robust Malware Family Classification Using Effective Features and Classifiers. *Applied Sciences*, 12

(15):7877, Jan. 2022. ISSN 2076-3417. doi: 10.3390/app12157877. URL https://www.mdpi.com/2076-3417/12/15/7877. Number: 15 Publisher: Multidisciplinary Digital Publishing Institute.

[35] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, second edition, 2009. doi: 10.1111/j.1751-5823.2009.00095_18.x.

[36] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1994.

[37] K. He and D.-S. Kim. Malware Detection with Malware Images using Deep Learning Techniques. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 95–102, Aug. 2019. doi: 10.1109/TrustCom/BigDataSE.2019.00022. URL https://ieeexplore.ieee.org/abstract/document/8887303. ISSN: 2324-9013.

[38] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[39] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[40] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1125–1134, 2017.

[41] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[42] S. Kiyol, D. Franchini, E. Moreno, R. Kowalski, W. Fernandez, and A. Milosevic. Ransomware Detection Using LSTM Networks and File Entropy Analysis: A Sequence-Based Approach, Oct. 2024.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, 2012.

[44] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[45] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[46] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[47] K. Li, W. Guo, F. Zhang, and J. Du. GAMBD: Generating adversarial malware against MalConv. *Computers & Security*, 130:103279, July 2023. ISSN 0167-4048. doi: 10.1016/j.cose.2023.103279. URL https://www.sciencedirect.com/science/article/pii/S016740482300189X.

[48] X. Ling, L. Wu, J. Zhang, Z. Qu, W. Deng, X. Chen, Y. Qian, C. Wu, S. Ji, T. Luo, J. Wu, and Y. Wu. Adversarial attacks against Windows PE malware detection: A survey of the state-of-the-art. *Computers & Security*, 128:103134, May 2023. ISSN 01674048. doi: 10.1016/j.cose.2023.103134. URL https://linkinghub.elsevier.com/retrieve/pii/S0167404823000445.

[49] Malware Bazaar. Malware Bazaar Database, 2026. URL https://bazaar.abuse.ch/.

[50] R. Martin, R. Pava, and S. Mishra. Analyzing machine learning algorithms for antivirus applications: a study on decision trees, support vector machines, and neural networks. *Issues in Information Systems*, 25:455–465, Oct. 2024.

[51] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. In *International Conference on Artificial Neural Networks*, pages 52–59. Springer, 2011.

[52] H. Menendez. Malware: The Never-Ending Arms Race. *Open Journal of Cybersecurity*, 1:1–25, Sept. 2021. doi: 10.46723/ojc.1.1.3.

[53] H. D. Menéndez and J. L. Llorente. Mimicking Anti-Viruses with Machine Learning and Entropy Profiles. *Entropy*, 21(5):513, May 2019. ISSN 1099-4300. doi: 10.3390/e21050513. URL https://www.mdpi.com/1099-4300/21/5/513.

[54] Microsoft. Advanced Technologies in Microsoft Defender Antivirus, 2024. URL https://learn.microsoft.com/en-us/defender-endpoint/adv-tech-of-mdav.

[55] Microsoft. Microsoft Defender for Individuals, 2026. URL https://www.microsoft.com/it-it/microsoft-365/microsoft-defender-for-individuals. Accessed: 2026-01-21.

[56] M. Miraoui and M. B. Belgacem. Binary and multiclass malware classification of windows portable executable using classic machine learning and deep learning. *Frontiers in Computer Science*, Volume 7 - 2025, 2025. ISSN 2624-

9898. doi: 10.3389/fcomp.2025.1539519. URL https://www.frontiersin.org/journals/computer-science/articles/10.3389/fcomp.2025.1539519.

[57] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2009.

[58] D. Quarta, F. Salvioni, A. Continella, and S. Zanero. Extended Abstract: Toward Systematically Exploring Antivirus Engines. In C. Giuffrida, S. Bardin, and G. Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 393–403, Cham, 2018. Springer International Publishing. ISBN 978-3-319-93411-2. doi: 10.1007/978-3-319-93411-2_18.

[59] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. Malware Detection by Eating a Whole EXE, Oct. 2017. URL http://arxiv.org/abs/1710.09435. arXiv:1710.09435 [stat].

[60] S. Rani, K. Tripathi, and A. Kumar. Machine learning aided malware detection for secure and smart manufacturing: a comprehensive analysis of the state of the art. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, Oct. 2023. ISSN 1955-2505. doi: 10.1007/s12008-023-01578-0. URL https://doi.org/10.1007/s12008-023-01578-0.

[61] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[62] M. S. A. A. Sakti, M. M. Hata, Z. H. Bolhan, and M. Y. Darus. Enhancing AI Malware Detection Using Neural Network with Binary Data Analysis. pages 53–63. Atlantis Press, Dec. 2024. ISBN 978-94-6463-589-8. doi: 10.2991/978-94-6463-589-8_7. URL https://www.atlantis-press.com/proceedings/invent-24/126005557. ISSN: 2352-538X.

[63] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 618–626, 2017.

[64] A. Sharma, P. Malacaria, and M. Khouzani. Malware Detection Using 1-Dimensional Convolutional Neural Networks. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 247–256, June 2019. doi: 10.1109/EuroSPW.2019.00034. URL https://ieeexplore.ieee.org/abstract/document/8802378.

[65] A. S. Shatnawi, Q. Yassen, and A. Yateem. An Android Malware Detection Approach Based on Static Feature Analysis Using Machine Learning Algorithms. *Procedia Computer Science*, 201:653–658, Jan. 2022. ISSN 1877-0509. doi: 10.1016/j.procs.2022.03.086. URL https://www.sciencedirect.com/science/article/pii/S1877050922004999.

[66] I. Sorokin. Comparing files using structural entropy. *Journal in Computer Virology*, 7(4):259–265, Nov. 2011. ISSN 1772-9904. doi: 10.1007/s11416-011-0153-9. URL https://doi.org/10.1007/s11416-011-0153-9.

[67] N. Syuhada Selamat and F. H. Mohd Ali. Comparison of malware detection techniques using machine learning algorithm. *Indonesian Journal of Electrical Engineering and Computer Science*, 16(1):435, Oct. 2019. ISSN 2502-4760, 2502-4752. doi: 10.11591/ijeecs.v16.i1.pp435-440. URL http://ijeecs.iaescore.com/index.php/IJEECS/article/view/19967.

[68] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2019. URL https://proceedings.mlr.press/v97/tan19a.html.

[69] E. Venkata Pawan Kalyan, A. Purushottam Adarsh, S. Sai Likith Reddy, and P. Renjith. Detection Of Malware Using CNN. In *2022 Second International Conference on Computer Science, Engineering and Applications (ICCSEA)*, pages 1–6, Sept. 2022. doi: 10.1109/ICCSEA54677.2022.9936225. URL https://ieeexplore.ieee.org/document/9936225/?arnumber=9936225.

[70] Virus Share. Virus Share Database, 2026. URL https://virusshare.com/.

[71] X. Xing, X. Jin, H. Elahi, H. Jiang, and G. Wang. A Malware Detection Approach Using Autoencoder in Deep Learning. *IEEE Access*, 10:25696–25706, 2022. ISSN 2169-3536. doi: 10.1109/ACCESS.2022.3155695. URL https://ieeexplore.ieee.org/abstract/document/9723074. Conference Name: IEEE Access.

[72] D. Zhan, Y. Duan, Y. Hu, W. Li, S. Guo, and Z. Pan. MalPatch: Evading DNN-Based Malware Detection With Adversarial Patches. *IEEE Transactions on Information Forensics and Security*, 19:1183–1198, 2024. ISSN 1556-6021. doi: 10.1109/TIFS.2023.3333567. URL https://ieeexplore.ieee.org/document/10319738/?arnumber=10319738. Conference Name: IEEE Transactions on Information Forensics and Security.

[73] J. Zhang. Machine Learning With Feature Selection Using Principal Component

Analysis for Malware Detection: A Case Study, Feb. 2019. URL http://arxiv.org/abs/1902.03639. arXiv:1902.03639 [cs].

[74] Y. Zhang, H. Li, Y. Zheng, S. Yao, and J. Jiang. Enhanced DNNs for malware classification with GAN-based adversarial training. *Journal of Computer Virology and Hacking Techniques*, 17(2):153–163, June 2021. ISSN 2263-8733. doi: 10.1007/s11416-021-00378-y. URL https://doi.org/10.1007/s11416-021-00378-y.

[75] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2921–2929, 2016.

A | Tests on Antivirus Engines

In addition to the experiments conducted on the mimicking CNN models, we performed preliminary tests to validate the signatures extracted by DAMANI directly with antivirus engines. The objective of these tests was to verify whether the extracted byte sequences were sufficient to trigger detection when evaluated by the antivirus software. We now describe these tests and discuss why they cannot be considered reliable, leading us to the usage of the mimicking model for the validation tests in Section 6.3 and Section 6.4.

The majority of these experiments replicate the tests performed on the CNN mimicking models. Several manipulations were applied to malware and benign samples to evaluate the effect of the extracted signatures. Specifically, we performed the following tests:

- **Signatures-only test:** all bytes outside the extracted signature regions were set to zero, keeping only the malicious regions unmodified;
- **Signature removal test:** only the extracted malicious bytes were zeroed out, while the rest of the binary was preserved. This is the only test where we expect no malicious binaries detected;
- **Random bytes removal test:** a random subset of bytes was set to zero to evaluate whether detection changes were caused by generic structural alterations rather than by the removal of signature-specific regions. Differently from the CNN-based experiment explained in Chapter 6, we progressively reduced the number of modified bytes. Surprisingly, even when only a small number (between 50 and 100) of randomly selected bytes were zeroed out, the antivirus engine no longer classified the file as malicious, suggesting a high sensitivity to binary alterations;
- **Signature injection into benign files:** extracted signatures were injected into randomly selected benign binaries by overwriting bytes at corresponding positions;
- **Ad hoc section injection:** since most of the previous manipulations corrupted the executable structure and invalidated the PE format, we also created a minimal benign executable and inserted one extracted signature at a time into a dedicated, custom section, ensuring that the binary remained functional and structurally valid.

Across all experiments, none of the manipulated binaries were detected as malicious by the original antivirus engines. These results must be interpreted carefully: in fact, several factors may explain the difference between the mimicking-model validation and the real-antivirus behavior.

First, modern antivirus engines rely on more than just static byte-level signatures: they may combine multiple detection components, including heuristic analysis, structural checks, metadata validation, unpacking routines, and potentially even behavioral or emulation-based mechanisms. Therefore, the presence of some malicious byte sequences alone may not be sufficient to trigger detection in isolation.

Second, the extracted regions identified by DAMANI may correspond to substrings or fragments of larger proprietary signatures. If the original detection rule search for the exact same signature contained in the database or spans a wider context, incorporating positional, structural, or semantic constraints, isolating only a portion of it would not create the full detection condition.

Third, direct binary manipulation may disrupt subtle structural properties of the executable that are implicitly considered during scanning, even when the file remains syntactically valid. In contrast, the mimicking CNN operates purely on pixel-level representations and does not account for higher-level file semantics or multi-stage detection pipelines.

Overall, these experiments highlight the limitation of directly analyzing proprietary antivirus engines in a black-box setting. As experimentally observed, even small binary modifications, such as zeroing a limited number of bytes, can lead to unpredictable or undefined detection behavior in real antivirus systems.

For this reason, the adoption of a mimicking model is necessary for our work. The mimicking CNN provides a stable and differentiable approximation of the antivirus decision boundary, allowing controlled manipulation of the input. Without such an approximation, deep learning-based signature extraction would not be feasible, since real antivirus engines do not expose intermediate scores, gradients, or structured feedback.

Therefore, the results obtained when directly testing the extracted signatures on real antivirus engines do not invalidate the DAMANI framework. Rather, they reinforce its core assumption: meaningful analysis of proprietary detection logic requires first constructing an interpretable and stable mimicking model. DAMANI should in fact be interpreted as a framework for approximating and studying static detection behavior under black-box constraints, not as a tool for reconstructing exact antivirus detection logic.

List of Figures

List of Tables