Universitat de Lleida

Going Deep into the Cat and the Mouse Game: Deep Learning for Malware Classification

Daniel Gibert Llauradó

http://hdl.handle.net/10803/671776

Universitat de Lleida

Ph.D. THESIS

Going Deep into the Cat and the Mouse Game: Deep Learning for Malware Classification

Doctoral Program in Engineering and Information Technology

by

Daniel Gibert Llauradó

Thesis Supervisor
Jordi Plances Cid

Thesis Director
Carles Mateu Piñol
Jordi Planes Cid

University of Lleida

November, 2020

Going Deep into the Cat and the Mouse Game: Deep Learning for Malware Classification

by

Daniel Gibert Llauradó

Submitted to the University of Lleida on November, 2020

Abstract

The fight against malware has never stopped since the dawn of computing. This fight has turned out to be a never-ending and cyclical arms race: as security analysts and researchers improve their defenses, malware developers continue to innovate, find new infection vectors and enhance their obfuscation techniques. Lately, due to the massive growth of malware streams, new methods have to be devised to complement traditional detection approaches and keep pace with new attacks and variants.

The aim of this thesis is the design, implementation, and evaluation of machine learning approaches for the task of malware detection and classification, due to its ability to handle large volumes of data and to generalize to never-before-seen malware.

This thesis is structured into four main parts. The first part provides a systematic and detailed overview of machine learning techniques to tackle the problem of malware detection and classification. This dissertation presents the following contributions that extend and complement previous work:(1) it provides a complete description of the methods and features in a traditional machine learning workflow for malware detection and classification; (2) it explores the challenges and limitations of traditional machine learning; (3) it analyzes recent trends and developments in the field with special emphasis on deep learning approaches; (4) it presents the research issues and unsolved challenges of the state-of-the-art techniques; and (5) it discusses new directions of research.

The second part is devoted to automating the feature engineering process through deep learning. Traditional machine learning approaches in the literature rely on the manual extraction of hand-crafted features defined by experts. However, these solutions depend almost entirely on the ability of the domain experts to extract characterizing features that accurately represent malware, and depending on the type of features extracted, such as n-gram features, the feature extraction process becomes a very time-consuming and memory-intensive process. Deep learning replaces the feature engineering process with an underlying system, which typically consists of a neural network with multiple layers, that performs both feature learning and classification. With deep learning one can start with raw data, as features will be automatically created by the network during the training procedure. This is achieved by stacking one or more convolutional layers, where the first ones learn to extract n-gram like features from the hexadecimal representation of malware's binary content and its the assembly language source code.

The third part of this thesis is devoted to investigating mechanisms to combine multiple modalities of information to increase the robustness of deep learning classifiers. Modalities are, essentially, channels of information. These data from multiple sources are semantically correlated, and sometimes provide complementary information to each other, thus reflecting patterns that are not visible when working with individual modalities on their own. Consequently, by only taking as input the raw bytes or opcodes a great deal of useful information for classification is overlooked. Subsequently, this thesis investigates how to combine various sources of information in deep learning architectures using an intermediate fusion strategy, and it presents a wide and deep learning framework, named HYDRA, that combines the benefits of feature engineering and deep learning.

The fourth part of this dissertation discusses the main issues and challenges faced by security researchers such as the availability of public benchmarks for malware research, and the problems of class imbalance, concept drift and adversarial learning. To this end, it provides an extensive evaluation of deep learning approaches for malware classification against common metamorphic techniques, and it explores their usage to augment the training set and reduce class imbalance. The metamorphic techniques analyzed are the following: (1) the dead code insertion technique, (2) the registers reassignment technique, (3) the subroutine reordering technique and (4) the code reordering through jumps technique.

Thesis Supervisor
Jordi Plances Cid

Thesis Director
Carles Mateu Piñol
Jordi Planes Cid

Resumen

La lucha contra el software malicioso no se ha interrumpido desde los inicios de la era digital, resultando en una carrera armamentística, cíclica e interminable; a medida que los analistas de seguridad y investigadores mejoran sus defensas, los desarrolladores de software malicioso siguen innovando, hallando nuevos vectores de infección y mejorando las técnicas de ofuscación. Recientemente, debido al crecimiento masivo y continuo del malware, se requieren nuevos métodos para complementar los existentes y así poder proteger los sistemas de nuevos ataques y variantes.

El objetivo de esta tesis doctoral es el diseño, implementación y evaluación de métodos de aprendizaje automático para la detección y clasificación de software malicioso, debido a su capacidad para manejar grandes volúmenes de datos y su habilidad de generalización.

La tesis se ha estructurado en cuatro partes. La primera parte presenta las siguientes contribuciones que extienden y complementan los estudios realizados hasta la fecha: (1) proporciona una descripción completa de los métodos y características empleados para la detección y clasificación de software malicioso; (2) explora los retos y limitaciones de los algoritmos de aprendizaje automático; (3) analiza las tendencias y avances, con especial énfasis en los métodos de aprendizaje profundo; (4) introduce los desafíos sin resolver de la técnicas actuales; y (5) presenta nuevas líneas de investigación.

La segunda parte consiste en la automatización del proceso de extracción de características mediante aprendizaje profundo. Por un lado, los algoritmos de aprendizaje automático se basan en la extracción manual de características definidas por expertos. Sin embargo, el desempeño de estas soluciones está sujeto casi exclusivamente de la habilidad de los expertos de determinar un conjunto de propiedades clave que permitan representar con mayor precisión al software malicioso, y dependiendo del tipo de características, como por ejemplo, n-grams, el proceso de extracción es muy costoso en términos de tiempo y memoria. Por lo contrario, el aprendizaje profundo reemplaza este proceso por un sistema, que típicamente está formado por una red neuronal con múltiples capas o estratos, que lleva a cabo el aprendizaje de características y posterior clasificación simultáneamente. Esto es debido a que con aprendizaje profundo, el sistema puede recibir los datos en bruto, dado que las características serán creadas automáticamente por la red neuronal durante el proceso de entrenamiento. Esto se logra mediante el apilamiento de una o más capas convolucionales, donde las primeras aprenden a extraer n-grams de la representación hexadecimal del código binario y del código ensamblador.

La tercera parte consiste en la investigación de mecanismos para combinar múltiples modalidades o fuentes de información y así, incrementar la robustez de los modelos de clasificación. Esto es debido a que los datos que se reciben de distintas fuentes de información pueden estar semánticamente correlacionados y, a veces, proporcionan información complementaria entre sí, reflejando patrones no visibles cuando se trabaja únicamente con una modalidad de información. Por consiguiente, si solo se emplea como entrada los bytes o los códigos de operación, se prescinde de información relevante, e.j. la tabla de importación de direcciones (IAT). Subsecuentemente, durante el desarrollo de esta tesis, se ha investigado como combinar varias modalidades de información en arquitecturas de aprendizaje profundo utilizando una estrategia de fusión intermedia, y se ha desenvolupado un framework, denominado HYDRA,

que combina la extracción manual de características y aprendizaje profundo.

La cuarta parte de esta tesis presenta los principales problemas y retos a los que se enfrentan los analistas de seguridad, como el problema de la desigualdad entre el número de muestras por familia, el aprendizaje adverso, entre otros. Asimismo, proporciona una extensa evaluación de los distintos métodos de aprendizaje profundo contra varias técnicas de ofuscación, y analiza la utilidad de estas para aumentar el conjunto de entrenamiento y reducir la desigualdad de muestras por familia. Más concretamente, las técnicas analizadas son las siguientes: (1) la técnica de inserción de código muerto; (2) la técnica de intercambio de registros; (3) la técnica de reordenación de rutinas; y (4) la técnica de reordenación de código mediante instrucciones "jump".

Resum

La lluita contra el programari maliciós no s'ha interromput mai des dels inicis de l'era digital, esdevenint una carrera armamentística cíclica i interminable; a mesura que els analistes en seguretat i investigadors milloren les seves defenses, els desenvolupadors de programari maliciós continuen innovant, trobant nous vectors d'infecció i millorant les tècniques d'ofuscació. Recentment, degut al creixement massiu i continu del programari maliciós, es requereixen nous mètodes per a complementar els existents i així poder protegir satisfactòriament els sistemes de nous atacs i variants.

L'objectiu d'aquesta tesis doctoral és el disseny, implementació i avaluació de mètodes d'aprenentatge automàtic per a la detecció i classificació de programari maliciós, a causa de la seva capacitat per a manipular grans volums de dades així com la seva habilitat de generalització.

La recerca s'ha estructurat en quatre parts. La primera part presenta les següents contribucions que estenen i complementen els estudis realitzats fins a data d'avui: (1) proporciona una descripció completa dels mètodes i característiques utilitzats per a la detecció i classificació de programari maliciós; (2) explora les limitacions dels algorismes d'aprenentatge automàtic; (3) analitza les noves tendències i avenços, amb especial èmfasi en els algorismes d'aprenentatge profund; (4) introdueix els reptes sense resoldre de les tècniques actuals; i (5) presenta noves línies d'investigació.

La segona part consisteix en l'automatització del procés d'extracció de característiques utilitzant tècniques d'aprenentatge profund. Per una banda, els algorismes d'aprenentatge automàtic es basen en l'extracció manual de característiques definides pels experts en seguretat i, conseqüentment, el seu rendiment està subjecte a l'habilitat d'aquests experts en determinar propietats clau que permetin representar amb major precisió el programari maliciós. No obstant, segons el tipus de característiques, com per exemple n-grams, el procés d'extracció d'aquestes es computacionalment molt costós i està limitat per dos factors: (1) la complexitat temporal, el temps estimat d'execució del procés, i (2) la complexitat espacial, l'ús de memòria (RAM) dels algorismes. Per altra banda, l'aprenentatge profund reemplaça aquest procés d'extracció per un sistema, típicament format per una xarxa neuronal artificial amb múltiples capes o estrats, que du a terme l'aprenentatge de les característiques i posterior classificació simultàniament. Això és degut a que amb aprenentatge profund, el sistema pot rebre les dades en brut, donat que les característiques es crearan automàticament per la xarxa neuronal durant l'entrenament. Aquest succés s'aconsegueix mitjançant l'apilament d'una o més capes convolucionals, on les primeres aprenen a extreure n-grams de la representació hexadecimal del codi binari o del codi ensamblador.

La tercera part consisteix en la investigació de mecanismes per a combinar múltiples modalitats o fonts d'informació per a incrementar la robustesa dels classificadors basats en aprenentatge profund. Això és degut a que les dades que es reben de diverses fonts d'informació acostumen a estar semànticament correlacionades i, a vegades, proporcionen informació complementària entre si; reflectint patrons no visibles quan només es processa una única modalitat d'informació. Conseqüentment, si només s'utilitza com a entrada els bytes o els codis d'operació, s'omet informació rellevant, p.e. la taula d'importació d'adreces (IAT), la qual s'utilitza per a cercar funcions implementades per mòduls externs. Subseqüentment, durant el desenvolupament d'aquesta tesis, s'ha investigat com combinar múltiples modali-

tats d'informació en arquitectures d'aprenentatge profund utilitzant una estratègia de fusió intermèdia, i s'ha desenvolupat un framework, denominat HYDRA, que combina l'extracció manual de característiques i aprenentatge profund.

La quarta part d'aquesta tesis presenta els principals problemes i reptes als que s'enfronten els analistes en seguretat, com el problema de la desigualtat entre el nombre de mostres per família, l'aprenentatge advers, entre altres. Tanmateix, proporciona una extensa avaluació dels diferents mètodes d'aprenentatge automàtic contra vàries tècniques d'ofuscació, i analitza la utilitat d'aquestes per a augmentar el conjunt de dades d'entrenament i reduir la desigualtat de mostres per família. Més concretament, les tècniques analitzades són: (1) la tècnica d'inserció de codi mort, (2) la tècnica d'intercanvi de registres, (3) la tècnica de reordenació de rutines i (4) la tècnica de reordenació de codi mitjançant instruccions "jump".

Acknowledgments

The development of this PhD thesis could not have been possible without the help, collaboration and understanding of many people. The following lines are dedicated to them.

First and foremost, I wish to express my deepest gratitude to my directors, Dr. Jordi Planes and Dr. Carles Mateu for giving me the opportunity to work under their guidance, their support and for generously sharing their knowledge with me, so I could advance my research. I would also like to thank the other members of the GREia research group, and specially Dr. Teresa Alsinet, Dr. Josep Argelich, Dr. Ramón Béjar and Dr. Cèsar Fernández, for teaching me with their experience and patience.

I would also like to thank Dr. Ruben Martins, Dr. Matt Fredrikson and Dr. Joao Marques-Silva for letting me conduct my research in their groups and providing me freedom and flexibility to do so.

Although the people mentioned above are very important in the process of developing this thesis, I would have never been able to finish it without the support of family and friends. I wish to show my gratitude to my family, especially my mother, for always supporting me and giving me the opportunities that have made me who I am. Last but not least, I would like to appreciate the support of my labmates, especially Sergi Vila, who provided stimulating discussions and distractions to rest my mind outside of my research.

Finally, I would like to acknowledge the financial support provided by the University of Lleida, particularly in the award of the pre-doctorate grant from 2017 to 2020. In addition, this thesis has received a grant for its linguistic revision from the Language Institute of the University of Lleida (2020 call) and has been partially funded by the Spanish MICINN Projects TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, PID2019-111544GB-C22.

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

AE	Autoencoder
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
APK	Android Application Package
AROW	Adaptive Regularization of Weights
ARRF	Attribute Relation File Format
AUC	Area Under the Curve
AV	Anti-virus
BIG	BigData Innovators Gathering
BOA	Bisector of Area
C&C	Command and Contol
CFG	Control Flow Graph
CNN	Convolutional Neural Network
CPU	Central Processing Unit
COA	Centroid of Area
CW	Confidence Weighted Learning
DAE	Denoising Autoencoder
DBN	Deep Belief Network
DBSAN	Density-based Clustering of Application with Noise
DDOS	Distributed Denial-of-Service
DFN	Dynamic Feed-forward Network
DLL	Dynamic Link Library
DNS	Domain Name System
DRN	Deep Residual Network
DS	Decision Stump
DT	Decision Tree
DTW	Dynamic Time Warping
ELF	Executable and Linkable Format
ELU	Exponential Linear Unit
ENT	Entropy
FCG	Function Call Graph
FPR	False Positive Rate
GAN	Generative Adversarial Network
GDPR	General Data Protection Programming Language

GLU	Gated Linear Unit
GR	Generalized Resolution
GRU	Gated Recurrent Unit
HAN	Hierarchical Attention Network
HCN	Hierarchical Convolutional Network
HCNN	Hierarchical Convolutional Neural Network
HTTP	Hypertext Transfer Protocol
IAT	Import Address Table
ICCG	Inter-Component Communication Graph
IDA	Interactive Disassembler
IDF	Inverse Document Frequency
IETF	Internet Engineering Task Force
IG	Information Gain
IMG	Image
IN	Intersection
IP	Internet Protocol
IT	Inference Tree
IoT	Internet of Things
JAR	Java Archive
K-NN	K-Nearest Neighbour
LBP	Local Binary Pattern
LE	Linear Executable
LOM	Largest of Maximum
LR	Logistic Regression
LSTM	Long Short Term Memory
MD5	Message-Digest Algorithm 5
MI	Mutual Information
MISC	Miscellaneous
ML	Machine Learning
MOM	Mean of Maximum
MP	Modus Ponens
NB	Naïve Bayes
NHERD	Normal Herd
NN	Neural Network
OOA	Object-oriented Association
OPC	Opcodes
OS	Operating System
PA-I	Passive Aggressive I
PA-II	Passive Aggressive II
PCA	Principal Component Analysis
PE	Portable Executable
PGL+	Possibilistic Logic Programming Language
Q1	First Quartile
Q2	Second Quartile
RAM	Random Access Memory
RB	Recursive Bipartition
REG	Register

RF	Random Forest
RL	Reinforcement Learning
ReLU	Rectifier Linear Unit
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
SEC	Section
SELU	Scaled Exponential Linear Units
SFN	Statoc Feed-forward Network
SHA-1	Secure Hash Algorithm 1
SIEM	Security Information and Event Manager
SJR	Scimago Journal Rank
SMO	Sequential Minimal Optimization
SOM	Smallest of Maximum
SU	Semantical Unification
SVN	Support Vector Machine
SW	Shockwave/Flash
TCP	Transmission Control Protocol
TF	Term Frequency
TF-IDF	Term Frequency - Inverse Document Frequency
TPR	True Positive Rate
UDP	User Datagram Protocol
UN	Uncertainty
URL	Uniform Resource Locator
US$1	United States Dollar
VT	Voted Perceptron

Chapter 1

Introduction

1.1 The Global Cybercrime Industry

The global cybercrime industry is estimated to be a US$1 trillion industry [50] and has been growing year after year. The underground services market is maturing at increased rates, providing malicious software, cyber-capabilities, and products to other criminals, gangs, and even nation states. It has evolved into a powerful ecosystem, built to exploit every opportunity and weakness in an increasingly connected world. According to the Global Risk Reports of 2019 [1], prepared by the World Economic Forum, cyber-attacks are listed among the most severe global threats, along with weather extremes, climate change and natural disasters. The rise of cyber dependency of people, things and organizations provides cybercriminals with a vast range of lucrative targets to exploit. The focus of cybercriminals is not limited to large companies or private individuals, but also affects industries and critical infrastructures, such as electrical and nuclear plants, information systems for financial institutions and health care providers. Some estimates [2] predict that the cost of cybercrime to the world would be $6 trillion annually by 2021, rising from $3 trillion in 2015. This estimate includes damage and destruction of data, stolen money, lost productivity, theft of personal, financial data and intellectual property, fraud, disruption of the normal course of business, forensic investigation and reputational harm.

The Internet Engineering Task Force (IETF) defined a cyberattack as "*an assault on system security that derives from an intelligent threat, i.e., an intelligent act that is a deliberate attempt (especially in the sense of a method or technique) to evade security services and violate the security policy of a system*"[3]. Cybercriminals use a variety of methods to launch a cyberattack, including but not limited to malware, phishing, denial of service, man-in-the-middle attack, SQL injection and zero-day exploit.

In this thesis we will take a deeper look into malware, and the many mechanisms to protect against it.

[1]`http://www3.weforum.org/docs/WEF_Global_Risks_Report_2019.pdf`
[2]`https://cybersecurityventures.com/`
[3]`https://www.rfc-editor.org/info/rfc2828`

1.1.1 Taxonomy of Malware

Malicious software, also known as malware, is any kind of software that is specifically designed to disrupt, damage or gain unauthorized access to a computer system or network. Depending on the purposes and proliferation systems, malware can be divided into various, not mutually exclusive categories.

- Adware. Malware designed to automatically generate online advertisements. This type of malware generates revenue for its developer by displaying advertisements on the user interface or the screen.
- Backdoor. Computer software that is designed to bypass a system's security mechanism and install itself on a computer to allow the attacker to access it.
- Bot. Software created to automatically perform specific operations such as Distributed Denial of Service (DDoS) attacks or distribute other malware. Bots are part of a botnet, a network of interconnected devices, which are controlled using command and control (C&C) software.
- Downloader. A downloader program's purpose is to download and install additional malicious programs.
- Launcher. A launcher is a computer program designed to stealthily launch other malicious programs.
- Ransomware. Malicious software that restricts user access to the computer system by encrypting the files or locking down the system while demanding a ransom for its release.
- Rootkit. Malware designed to conceal the existence of other malicious programs.
- Spyware. Computer software that spies and collects sensitive information without permission from a victim's computer. Examples include key-loggers, password gravers and sniffers.
- Trojan. A Trojan is a type of malicious software that disguises itself as legitimate software to trick users into downloading and installing malware on their systems.
- Virus. Malicious software that can propagate itself from device to device.
- Worm. A type of virus that exploits vulnerabilities of the operating system to spread. The major difference between worms and viruses is the ability of worms to independently self-replicate and spread while viruses depend on human activity.

A brief look at the history of malicious software reminds us that the presence of malware threats has been with us since the dawn of computing. The earliest documented virus appeared during the 1970s. It was known as the Creeper Worm and was an experimental self-replicating program that copied itself to remote systems and displayed the message: "I'm the creeper, catch me if you can". Later, in the early 80s, Elk Cloner appeared, a boot-sector virus that targeted Apple II computers. From these simple beginnings, a massive industry was born and, since then, the fight against malware has never stopped. This fight has turned out to be a never-ending and cyclical arms race: as security analysts and researchers improve their defenses, malware developers continue to innovate, find new infection vectors and enhance their obfuscation techniques. Malware threats have been expanding vertically (i.e. numbers and volumes) and horizontally (i.e. types and functionality) due to the opportunities provided by technological advances. Internet, social networks, smartphones, Internet of Things (IoT) devices and so on, make it possible to create smart and sophisticated malware. In recent years, ransomware and cryptomining malware emerged as the most prolific types, with Cerber and Locky holding computers all over the globe to ransom while Cryptoloot used the victim's computing power to mine for crypto without their knowledge. Even though malware targeting computer systems still predominate in the ecosystem, mobile and IoT malware is on the rise. According to Symantec [16], mobile malware variants increased by 54% in 2017, while IoT attacks had a 600% increase, with the Mirai botnet and its variants serving as the vehicle for some of the most potent DDoS attacks in history [47].

1.2 Cyberdefenses

To keep up with malware, security analysts and researchers need to constantly improve their cyber-defenses. One essential element is endpoint protection. Endpoint protection provides a suite of security programs including, but not limited to, firewall, URL filtering, email protection, anti-spam and sandboxing. Specifically, anti-malware software provides the last layer of defense. Anti-virus (AV) engines are responsible for preventing, detecting and removing malicious software installed on the endpoint device. Traditionally, AV solutions relied on signature-based and heuristic-based methods. A signature is an algorithm or hash that uniquely identifies specific malware while heuristics is a set of rules determined by experts after analyzing the behavior of malware. However, both approaches require the malware to be analyzed prior to the definition of these rules and heuristics. The goal of malware analysis is to provide information about the characteristics, purpose and behavior of a given piece of software. There are two types of analysis: (1) static analysis and (2) dynamic analysis.

1.2.1 Static Analysis

Static analysis consists of examining the code or structure of the executable without running it. This kind of analysis can confirm whether a file is malicious and provides basic information about its functionality. Common static analysis approaches are:

- Finding sequences of characters or strings. Searching through the strings of a program is the simplest way to obtain hints about its functionality. Strings extracted from the binary can contain references to filepaths of files modified or accessed by the executable, URLs which the program accesses, domain names, IP addresses, attack commands, names of Windows dynamic link libraries (DLLs) loaded, registry keys, and so on.

- Gathering the linked libraries and functions of an executable, as well as the metadata about the file included in the headers. These data provide information about code libraries and functionalities common to many programs, that programmers link so that they do not need to re-implement a certain functionality. The names of these Windows functions can give us an idea of what the executable does.

- Searching for packed/encrypted code. Malware writers usually use packing and encryption to make their files more difficult to analyze. Software programs that have been packed or encrypted usually contain very few strings and higher entropy compared to legitimate programs.

- Disassembling the program, i.e. translating machine code into assembly language. This reverse-engineering process loads the executable into a disassembler to discover what the program does.

1.2.2 Dynamic Analysis

Dynamic analysis involves executing the program and monitoring its behavior on the system. This is typically performed when static analysis has reached a dead end, either due to obfuscation, or to having exhausted the available static analysis techniques. Unlike static analysis, it traces the real actions executed by the program. However, the analysis must be run in a safe environment so as not to expose the system to unnecessary risks, where the system is both the machine running the analysis tool and the rest of the machines on the network. To this end, dedicated physical or virtual machines are set up. Physical machines must be set up on air-gapped networks, that is isolated networks where machines are disconnected from the Internet or any other network, to prevent malware from spreading. The main downside of physical machines is this scenario with no Internet connection, as many malicious programs depend on Internet connection for updates, command and control and other features. The second option is to set up virtual machines to perform dynamic analysis. A virtual machine emulates a computer system and provides the functionality of a physical computer. The operating system (OS) running in the virtual machine is kept isolated from the host OS and thus, malware running on a virtual machine cannot harm the host OS. There are several all-in-one software products based on sandbox technology that can be used to perform basic dynamic analysis. The most well-known is the Cuckoo Sandbox [4], an open source automated malware analysis system. This modular sandbox provides capabilities to trace Application Programming Interface (API) function calls, analyze network traffic or perform memory analysis. Alternatively, there is a wide list of utilities to dynamically analyze malware and perform advanced and specific monitoring of some functionalities. Process Monitor [5], or procmon, is a tool for Windows that monitors certain registry, file system, network, process and thread activity. Process Explorer [6] shows the information about which handles and DLL processes are opened or loaded into the operating system. Regshot [7] is a registry compare utility that

[4]https://cuckoosandbox.org/

[5]https://docs.microsoft.com/en-_us/sysinternals/downloads/procmon

[6]https://docs.microsoft.com/en-_us/sysinternals/downloads/process-explorer

[7]https://sourceforge.net/p/regshot/wiki/Home/

allows snapshots of registries to be taken and compared. NetCat [8] is a networking utility that can be used to monitor data transmission over a network. Wireshark [9] is an open source sniffer that allows packets to be captured and network traffic to be intercepted and logged. Another indispensable software utility is debuggers. A debugger is used to examine the execution of another program. They provide a dynamic view of a program as it runs. The primary debugger of choice for malware analysts is OllyDbg [10], an x86 debugger that is free and has many plugins to extend its capabilities.

Both types of analysis have their advantages and limitations and they complement each other. Static analysis is faster but, if malware is successfully concealed using code obfuscation techniques, it could evade detection. Contrarily, polymorphic and metamorphic techniques used to evade static analysis hardly evades dynamic analysis as it monitors and analyzes the runtime execution of a program. However, dynamic analysis is slower and computational intensive and can also be evaded with fingerprinting techniques to detect the presence of sandboxes by looking for artifacts or characteristics that could reveal the virtual machine [2]. When malware detects that it is running in a virtual machine or sandbox it will execute differently in a way that hides its malicious behavior. Nevertheless, traditional malware detection and malware analysis are unable to keep pace with new attacks and variants.

1.2.3 Malware Evolution

The diversity, sophistication and availability of malicious software pose enormous challenges for securing networks and computer systems from attacks. Malware is constantly evolving and forces security analysts and researchers to keep pace by improving their cyberdefenses. The proliferation of malware increased due to the use of polymorphic and metamorphic techniques used to evade detection and hide its true purpose. Polymorphic malware uses a polymorphic engine to mutate the code while keeping the original functionality intact. Packing and encryption are the two most common ways to hide code. On the one hand, packers hide the real code of a program through one or more layers of compression. Then, at runtime the unpacking routines restore the original code in memory and execute it. On the other hand, crypters encrypt and manipulate malware or part of its code, to make it harder for researchers to analyze the program. A crypter contains a stub used to encrypt and decrypt malicious code. Metamorphic malware rewrites its code to an equivalent whenever it is propagated. Malware authors may use multiple transformation techniques including, but not limited to, register renaming, code permutation, code expansion, code shrinking and garbage code insertion. The combination of the aforementioned techniques resulted in rapidly growing malware volumes, making forensic investigations of malware cases time-consuming, costly and more difficult.

As a result, organizations are facing the daunting challenge of dealing with millions of attacks a day. In addition, organizations are also experiencing a shortage of cybersecurity skills and talent [61]. The identified issues present a unique opportunity for machine learning to significantly impact and change the cybersecurity landscape due to its ability to handle large volumes of data [21] and to generalize

[8] `http://netcat.sourceforge.net/`

[9] `https://www.wireshark.org/`

[10] `http://www.ollydbg.de/`

to never-before-seen malware.

1.3 The Promise of Machine Learning for Tackling the Problem of Malware Detection and Classification

Decades ago the number of malware threats was relatively low and simple handcrafted rules were often enough to detect threats. Lately, due to the massive growth of malware streams, anti-malware solutions have not been allowed to rely solely on expensive hand-designed rules. Consequently, machine learning (ML) has become an appealing signature-less approach for detecting and classifying malware due to its ability to generalize in relation to never-before-seen malware. Traditional machine learning approaches in the literature [59, 78, 64, 67, 38, 65, 3, 79, 37, 19, 54, 69] rely mainly on feature engineering to extract a set of discriminative features that provide a feature vector representation of malware that a classifier uses to determine the maliciousness of an executable. However, these solutions depend almost entirely on the ability of the domain experts to extract characterizing features that accurately represent the malware. Feature extraction is a very time-consuming process, and in particular, n-gram extraction where $n \geq 3$, because although features are good for malware classification, they are impractical to compute in an industrial malware classification system. For instance, considering $n = 3$, the number of possible n-grams is $256^3 = 16.777.216$. This leads to two main problems. First, the resulting feature vector is too large to keep in memory, even if malware n-grams tend to follow a Zipfian distribution [65]. Second, the machine learning model will be affected by the curse of dimensionality [8, 12] which means that the number of samples in the dataset that need to be accessed to estimate a function with a given level of accuracy grows exponentially with the underlying dimensionality. Therefore, feature selection or dimensionality reduction methods must be applied prior to training the model, increasing the computational time required to perform the feature extraction process. As a result, alternatives to feature engineering are required to build sustainable malware detection and classification systems.

Chapter 2

Aim and Objectives of the Thesis

The aim of the thesis is to study alternatives to manual feature engineering for the task of malware detection and classification. To achieve this goal, research was divided into two main parts. The first part consisted of the automation of the feature engineering process through deep learning. Deep learning replaces the feature engineering process by an underlying system, which typically consists of a neural network (NN) with multiple layers, that performs both feature learning and classification. With deep learning one can start with raw data as features will be automatically created by the network during the training procedure. When speaking of malware, and more specifically, malicious software targeting the Windows operating system, raw data refer to either the hexadecimal representation of malware's binary content or the assembly language source code of the executable. Thus, we explored both representations during the development of the thesis.

The second part explores mechanisms to combine multiple modalities of information to increase the robustness of deep learning classifiers. Modalities are, essentially, channels of information. These data from multiple sources are semantically correlated, and sometimes provide complementary information to each other, thus reflecting patterns that are not visible when working with individual modalities on their own. Consequently, by only taking as input the raw bytes or opcodes a great deal of useful information for classification is overlooked, such as structural information of the Portable Executable (PE) file, the import address table (IAT) which is used as a lookup table when the application is calling a function from a different module, etc. Subsequently, in the second part of the thesis various ways to combine multiple modalities of information in deep learning architectures are investigated.

To sum up, the research objectives were:

- To review the state-of-the-art approaches in the literature for malware detection and classification.
- To automate the costly manual feature engineering process in traditional machine learning approaches.
- To develop mechanisms for the combination of multiple sources of information in deep learning architectures.

Chapter 3

Thesis Structure

This thesis is presented as a compendium of four journal articles, six articles published in international peer-reviewed conferences (two conference articles ranked as Core A and two ranked as Core B) and a patent application. Regarding the journal articles, two are published in journals belonging to the First Quartile (Q1), one is published in a journal belonging to the Second Quartile (Q2) according to the Scimago Journal Rank (SJR) and the other one is in the reviewing phase. An overview of the publications is presented in Figure 3.1. For more information about the articles published in international peer-reviewed conferences we refer the reader to Chapter 10. In this regard, the dissertation is organized as follows:

Chapter 1 is an introductory section of this thesis. This chapter presents an overview of the global cybercrime industry and the existing cyberdefenses to keep malware at bay.

The aim and objectives of this thesis are presented in Chapter 2 and the entire scheme of the doctoral thesis is provided in Chapter 3.

Chapter 4 provides an introduction to the task of malware classification and describes the limitations associated with manual feature engineering and, in particular, n-gram based features. In this context, Chapter 4 presents various research articles published in the proceedings of peer-reviewed international conferences that introduce alternatives to manual feature engineering. To this end, these alternatives use deep learning to replace the feature engineering process with an underlying system that performs both feature learning and classification. The articles can be divided into two groups, depending on the nature of the input data: (i) opcode-based approaches [24, 27] and (ii) byte-based approaches [31, 26]. On the one hand, Gibert et al. [24] introduce an architecture to learn n-gram like features from the malware's assembly language instructions. In addition, D. Gibert et al [27] present a hierarchical neural network architecture to deal with the hierarchical structure of programs. On the other hand, Gibert et al. [31] encode the content of a malicious program as an entropy stream, where each value describes the amount of entropy of a small chunk of code in a specific location of the file, while Gibert et al. [26] propose a method to encode the raw byte sequences using denoising autoencoders. Lastly, D. Gibert et al [29] introduce a bimodal approach to categorize malware into families based on both modalities of data: (1) the byte sequence representing the malware's binary content, and (2) the assembly language instructions extracted from the assembly language source code of malware.

Chapter 5 corresponds to the article named *The rise of machine learning for* *detection and classification of malware: Research developments, trends and challenges* [30]. The contributions of the research article are: (1) it presents an overview of the methods and features in a traditional machine learning workflow for malware detection and classification, (2) it explores the challenges and limitations of traditional machine learning and (3) it analyzes recent trends and developments in the field with special emphasis on deep learning approaches. Furthermore, (4) it presents the research issues and unsolved challenges of the state-of-the-art techniques and (5) it discusses the new directions of research. The survey helps researchers to have an understanding of the malware detection field and of the new developments and directions of research explored by the scientific community to tackle the problem.

Chapter 6 corresponds to the article named *Using convolutional neural networks for classification of malware represented as images* [32], which provides a way to categorize malware into groups and identify its family based on state-of-the-art image recognition techniques. In this article, malicious software is visualized as gray scale images and classified into families based on a set of discriminant patterns extracted by a convolutional neural network.

Chapter 7 describes a patent application, whose co-ownership belongs to Leap in Value, S.L and the University of Lleida, and is currently under revision and not published. The invention described presents a computer-implemented method, system and computer program for identifying a malicious file that combines different types of analysis, processes and procedures that allow a malicious file to be detected and classified.

Chapter 8 corresponds to the article named *HYDRA: a multimodal deep learning framework for malware classification* [28]. It introduces a baseline framework to address the task of malware classification by combining multiple modalities of information. This framework consists of both hand-engineered and end-to-end components to combine the benefits of feature engineering and deep learning so that malware characteristics are effectively represented. This is achieved through a modular architecture that can be broken down into three subnetworks, according to the different types of input in the system: (1) the list of Windows API function calls, (2) the sequence of assembly language instructions representing malware's assembly language source code and (3) the sequence of hexadecimal values representing malware's binary content.

Chapter 9 provides a comprehensive analysis of the main issues and challenges that face security researchers in order to build sustainable malware detection and classification systems such as the problem of class imbalance and adversarial learning. Correspondingly, it provides an extensive evaluation of the performance of deep learning approaches for malware classification against common metamorphic techniques: (1) the dead code insertion technique, (2) the register reassignment technique, (3) the subroutine reordering technique and (4) the code reordering through jumps technique. Moreover, the usage of the aforementioned metamorphic techniques to augment the training set and reduce class imbalance is investigated.

Chapter 10 summarizes the research activities carried out during the dissertation and presents an android malware detection approach based on the function call graph representation of an application [25]. This conference article has been included in Chapter 10 as its target is Android malware instead of Windows malware.

Finally, Chapter 11 provides the concluding remarks of this thesis and presents some future lines of research.

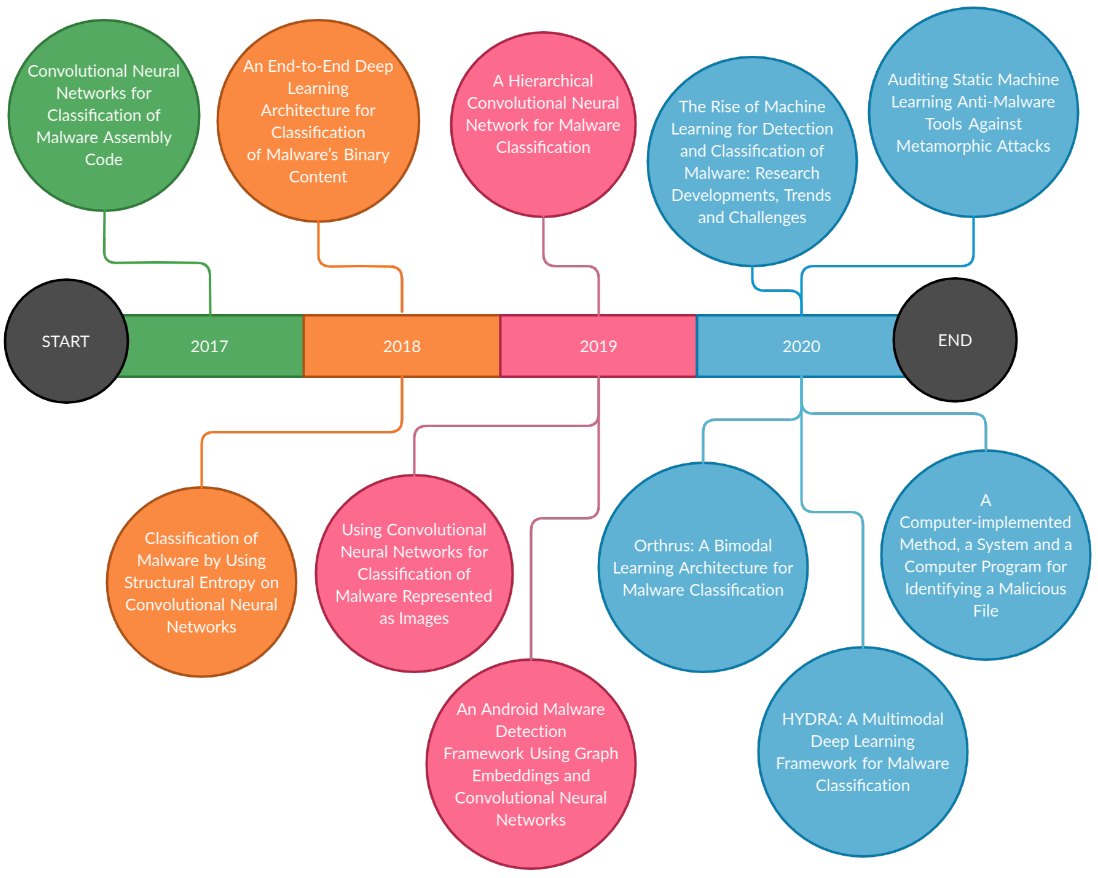

Convolutional Neural Networks for Classification of Malware Assembly Code
An End-to-End Deep Learning Architecture for Classification of Malware's Binary Content
A Hierarchical Convolutional Neural Network for Malware Classification
The Rise of Machine Learning for Detection and Classification of Malware: Research Developments, Trends and Challenges
Auditing Static Machine Learning Anti-Malware Tools Against Metamorphic Attacks
START
2017
2018
2019
2020
END
Classification of Malware by Using Structural Entropy on Convolutional Neural Networks
Using Convolutional Neural Networks for Classification of Malware Represented as Images
Orthrus: A Bimodal Learning Architecture for Malware Classification
A Computer-implemented Method, a System and a Computer Program for Identifying a Malicious File
An Android Malware Detection Framework Using Graph Embeddings and Convolutional Neural Networks
HYDRA: A Multimodal Deep Learning Framework for Malware Classification

Figure 3.1: Dissertation timeline.

Chapter 4

End-to-End Learning as an Alternative to N-Gram Approaches

4.1 The Task of Malware Classification

Because of the variety of malware functionalities, it is important not only to detect malicious software, but also to differentiate between different kinds of malware in order to provide a better understanding of their capabilities. This task is known as malware classification. Distinguishing and classifying different types of malware is an important task as it provides information to better understand how malware has infected the computers or devices, its threat level and how to protect against it. Notice that the features extracted from a computer program are useful both for detecting if it is malicious and for classifying it into families, and the only difference between machine learning solutions for detection or classification of malware lies in the output returned by the system implemented. On the one hand, a malware detection system outputs a single value $y = f(x)$, in the range from 0 to 1, which indicates the maliciousness of the executable. On the other hand, a classification system outputs the probability of a given executable belonging to each output class or family, $y \in \mathbb{R}^N$, where N indicates the number of different families.

Nevertheless, the task of malware detection and classification has not received the same attention in the research community as other applications, where rich labeled datasets exist, including image classification, speech recognition, etc. Due to legal restrictions, benign binaries are not shared, as they are often protected by copyright laws and thus, researchers cannot share the binaries used in their research. Contrarily, malicious binaries are shared through web sites such as VirusShare and VXHeaven. However, unlike other domains where data may be labeled very quickly and in many cases by a non-expert, determining whether a file is malicious and its corresponding family or class can be a time-consuming process, even for security experts. Furthermore, services like VirusTotal specifically restrict sharing the vendor anti-malware labels with the public. Thus, for reproducibility purposes, the machine learning solutions proposed in this thesis have been evaluated on the data provided by Microsoft for the Big Data Innovators Gathering Challenge [66] of 2015, a high-quality public labeled benchmark. A complete description of the dataset is provided in the next section.

4.1.1 The Microsoft Malware Classification Challenge

Table 4.1: Class distribution in the Microsoft Malware Classification Challenge dataset [66].

Family	#Samples	Type
Ramnit	1541	Worm
Lollipop	2478	Adware
Kelihos_ver3	2942	Backdoor
Vundo	475	Trojan
Simda	42	Backdoor
Tracur	751	TrojanDownloader
Kelihos_ver1	398	Backdoor
Obfuscator.ACY	1228	Any kind of obfuscated malware
Gatak	1013	Backdoor

Microsoft provided almost half a terabyte of malicious software targeting the Windows OS for the Big Data Innovators Gathering Challenge [66] of 2015. Nowadays, the dataset is hosted on Kaggle[1] and is publicly accessible. The dataset has become the standard benchmark to evaluate machine learning techniques for the task of malware classification. The set of samples represents 9 different malware families, where each sample is identified by a hash and its class, an integer representing one of the 9 malware families to which the malware belongs: (1) RAMNIT, (2) LOLLIPOP, (3) KELIHOS_VER3, (4) VUNDO, (5) SIMDA, (6) TRACUR, (7) KELIHOS_VER1, (8) OBFUSCATOR.ACY and (9) GATAK. Figure 4.1 displays the distribution of classes of the training data. Notice that the number of instances of some families significantly outnumbers the number of instances of other families.

4.1.2 The Portable Executable File Format

For the 32-bit and 64-bit version of the Windows operating system, the executables, object code, DLLs, FON Font files and others are represented with the Portable Executable (PE) file format, with the PE32 format standing for Portable Executables of 32-bit while PE32+ refers to the Portable Executables file format for 64-bit architectures.

Portable Executables encapsulate the information necessary for a Windows operating system to manage the executable code. This includes dynamic library references for linking, API export and import tables, resource management data and threat-local storage data. A PE file consists of a number of headers and sections that tell the dynamic linker how to map the file into memory. See Figure 4.1. The PE Header contains information about the executable such as the number of sections, the size of the "PE Optional Header", characteristics of the file, etc [2]. It also contains the import address table (IAT), which is a lookup table used by the application when calling a function in a different module. In addition, a Portable Executable file has various sections that contain the code and data of the executable including, but not limited to, the following:

[1]`https://www.kaggle.com/c/malware-classification/`

[2]`https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files\#PE_Header`

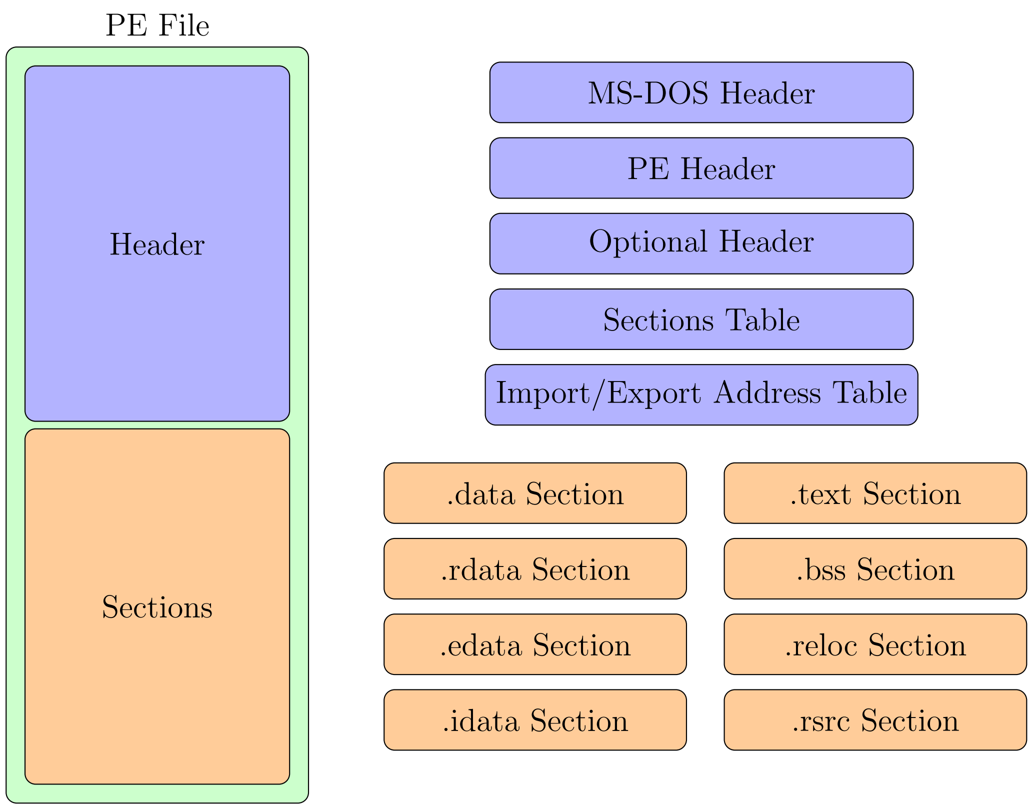


Figure 4.1: Portable Executable (PE) file format.

- The `.data` section. This section is used to declare initialized data or constants that do not change at runtime.
- The `.bss` section. This section is used for declaring variables and contains uninitialized data.
- The `.text` section. This section keeps the actual code of the program.
- The `.rsrc` section. This section contains all the resources of the program.
- The `.rdata` section. This section holds the debug directory which stores the type, size and location of various types of debug information stored in the file.
- The `.idata` section. This section contains information about functions and data that the program imports from DLLs.
- The `.edata` section. This section contains the list of the functions and data that the PE file exports for other programs.
- The `.reloc` section. This section holds a table of base relocations. A base relocation is an adjustment to an instruction or initialized variable value that is needed if the loader could not load the file where the linker assumed it would.

More information on the PE file format can be found in the documentation provided by Microsoft [3].

[3] `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`

4.2 N-Gram Approaches for Malware Detection and Classification

In machine learning, a workflow is an iterative process that involves gathering available data, cleaning and preparing the data, building models, validating and deploying into production. See Fig 4.2. Accordingly, the data preparation process of traditional machine learning approaches involves preprocessing the executable to extract a set of features that provide an abstract view of the software. Afterwards, the features are used to train a model to solve the task at hand. For a complete description of the features used for malware detection and classification the reader is referred to Section 5.

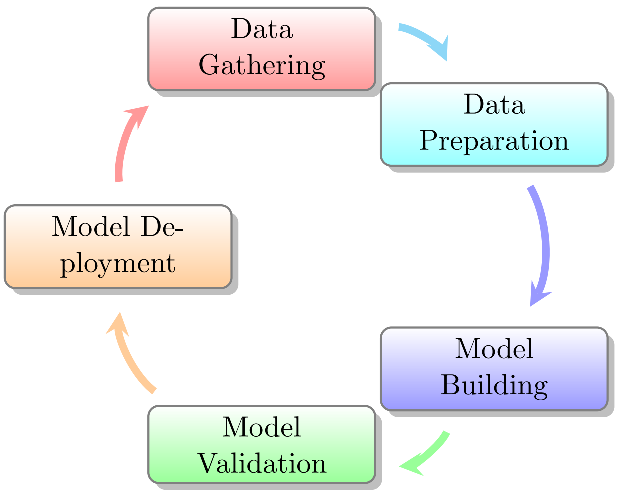
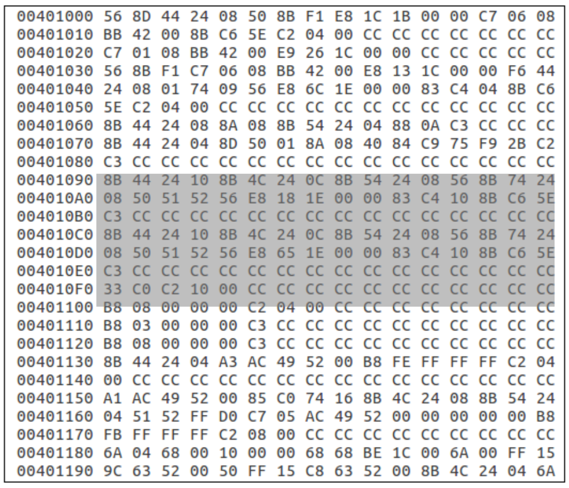


Figure 4.2: Machine learning workflow.

The most common type of features for malware detection and classification is n-grams. An n-gram is a contiguous sequence of n items from a given sequence of text. N-grams can be extracted from the hexadecimal representation of malware's binary content and from the assembly language source code. On the one hand, the hexadecimal representation represents the binary content of an executable as a sequence of bytes (base-16 number representation with digits [0-9] and [A-F]). See Figure 4.3. Alternatively, the assembly language source code contains the symbolic machine code of an executable with metadata information as function calls, memory allocation and variable information. See Figure 4.4. Thus, byte n-grams [78, 64] and opcode n-grams [67, 38] refer to the unique combination of all n consecutive bytes and opcodes as individual features, respectively, where an opcode refers to the name of a specific instruction, i.e. "ADD", "MUL", "PUSH", etc, without its arguments. N-gram based approaches construct a feature vector containing an abstract representation of malware, where each element in the vector indicates the number of appearances of a particular n-gram in the sequence of text (or the ratio). Consequently, the length of the feature vector depends on the number of unique n-grams, which increases exponentially with n. For instance, if we want to extract byte n-grams with $n = 3$, the number of possible n-grams is $256^3 = 16.777.216$. This leads to two main problems. First, the resulting feature vector is too large to keep in memory, even if malware n-grams tend to follow a Zipfian distribution [65]. Second, the machine learning model will be affected by the curse of dimensionality [8, 12]



```
00401000 56 8D 44 24 08 50 8B F1 E8 1C 1B 00 00 C7 06 08
00401010 BB 42 00 8B C6 5E C2 04 00 CC CC CC CC CC CC CC
00401020 C7 01 08 BB 42 00 E9 26 1C 00 00 CC CC CC CC CC
00401030 56 8B F1 C7 06 08 BB 42 00 E8 13 1C 00 00 F6 44
00401040 24 08 01 74 09 56 E8 6C 1E 00 00 83 C4 04 8B C6
00401050 5E C2 04 00 CC CC CC CC CC CC CC CC CC CC CC CC
00401060 8B 44 24 08 8A 08 8B 54 24 04 88 0A C3 CC CC CC
00401070 8B 44 24 04 8D 50 01 8A 08 40 84 C9 75 F9 2B C2
00401080 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401090 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010A0 08 50 51 52 56 E8 18 1E 00 00 83 C4 10 8B C6 5E
004010B0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010C0 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010D0 08 50 51 52 56 E8 65 1E 00 00 83 C4 10 8B C6 5E
004010E0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010F0 33 C0 C2 10 00 CC CC CC CC CC CC CC CC CC CC CC
00401100 B8 08 00 00 00 C2 04 00 CC CC CC CC CC CC CC CC
00401110 B8 03 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401120 B8 08 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401130 8B 44 24 04 A3 AC 49 52 00 B8 FE FF FF FF C2 04
00401140 00 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401150 A1 AC 49 52 00 85 C0 74 16 8B 4C 24 08 8B 54 24
00401160 04 51 52 FF D0 C7 05 AC 49 52 00 00 00 00 00 B8
00401170 FB FF FF FF C2 08 00 CC CC CC CC CC CC CC CC CC
00401180 6A 04 68 00 10 00 00 68 68 BE 1C 00 6A 00 FF 15
00401190 9C 63 52 00 50 FF 15 C8 63 52 00 8B 4C 24 04 6A
```

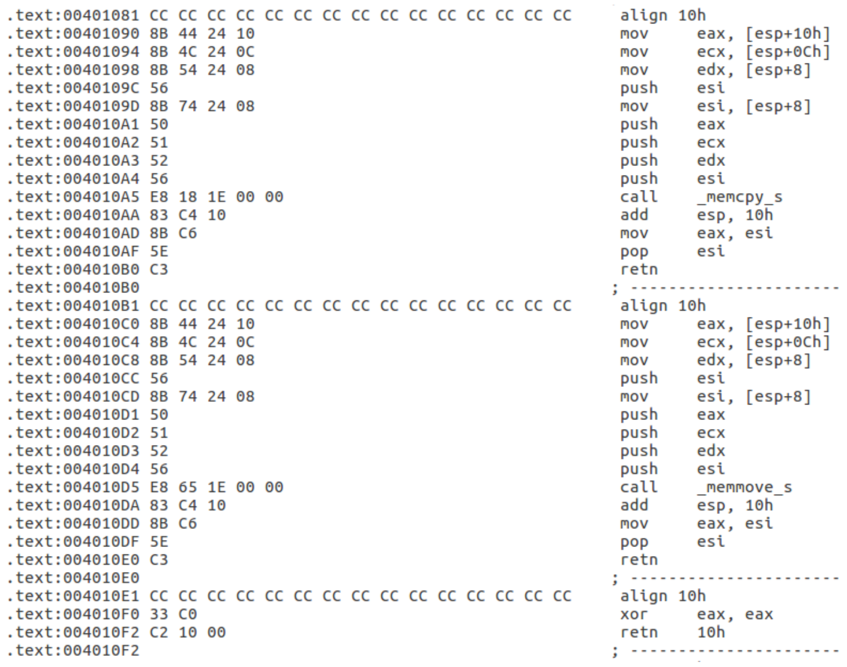
Figure 4.3: Hexadecimal view of a PE file.

which means that the number of samples in the dataset that need to be accessed to estimate a function with a given level of accuracy grows exponentially with the underlying dimensionality. Therefore, feature selection or dimensionality reduction methods must be applied prior to training the model. While both methods are used for reducing the number of features, there is an important difference between them: Feature selection is simply selecting and excluding given features without changing them, while dimensionality reduction transforms features into a lower dimension. More specifically, for the task of malware detection and classification, the most common dimensionality reduction method is the hashing trick [64, 38].

1. Feature Selection. Feature selection is the process of selecting a subset of relevant features from the initial input space for use in model construction. A common approach is to rank the features based on high information gain entropy in decreasing order [67, 78]. Information Gain (IG), also referred as Mutual Information (MI), is an index of statistical dependence between two variables [71]. In the case of a classification task, it measures the dependence between a feature X and the target variable Y. This is done by measuring how much knowing one of these variables reduces uncertainty about the other. The Mutual Information between two variables is a non-negative value, which measures the dependency between the variables. For two independent variables, their mutual information will be 0. Otherwise, for dependent variables, higher mutual information mean higher dependency.

2. The Hashing Trick. Feature hashing, also known as the hashing trick, is a method for handling sparse, high-dimensional feature vectors by using a hash function to determine the feature's location in a lower-dimensional vector. It can be seen as a random projection of the input space $A \in \mathbb{R}^n$ to a low dimensional space $B \in \mathbb{R}^m$, where $m \ll n$. More specifically, given an array of size N that counts the number of times each n-gram occurred, and a hash function, the hashing trick maps each n-gram to a location in a lower dimensional array.

Afterwards, the resulting low-dimensional feature vector is used for training a classification algorithm.



```
.text:00401081 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:00401090 8B 44 24 10                                      mov     eax, [esp+10h]
.text:00401094 8B 4C 24 0C                                      mov     ecx, [esp+0Ch]
.text:00401098 8B 54 24 08                                      mov     edx, [esp+8]
.text:0040109C 56                                               push    esi
.text:0040109D 8B 74 24 08                                      mov     esi, [esp+8]
.text:004010A1 50                                               push    eax
.text:004010A2 51                                               push    ecx
.text:004010A3 52                                               push    edx
.text:004010A4 56                                               push    esi
.text:004010A5 E8 18 1E 00 00                                   call    _memcpy_s
.text:004010AA 83 C4 10                                         add     esp, 10h
.text:004010AD 8B C6                                            mov     eax, esi
.text:004010AF 5E                                               pop     esi
.text:004010B0 C3                                               retn
.text:004010B0                                                 ; ---------------------
.text:004010B1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010C0 8B 44 24 10                                      mov     eax, [esp+10h]
.text:004010C4 8B 4C 24 0C                                      mov     ecx, [esp+0Ch]
.text:004010C8 8B 54 24 08                                      mov     edx, [esp+8]
.text:004010CC 56                                               push    esi
.text:004010CD 8B 74 24 08                                      mov     esi, [esp+8]
.text:004010D1 50                                               push    eax
.text:004010D2 51                                               push    ecx
.text:004010D3 52                                               push    edx
.text:004010D4 56                                               push    esi
.text:004010D5 E8 65 1E 00 00                                   call    _memmove_s
.text:004010DA 83 C4 10                                         add     esp, 10h
.text:004010DD 8B C6                                            mov     eax, esi
.text:004010DF 5E                                               pop     esi
.text:004010E0 C3                                               retn
.text:004010E0                                                 ; ---------------------
.text:004010E1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010F0 33 C0                                            xor     eax, eax
.text:004010F2 C2 10 00                                         retn    10h
.text:004010F2                                                 ; ---------------------
```

Figure 4.4: Assembly view of the grayed part in Figure 4.3. The first column represents the address, the second column the byte sequence and the third column the mnemonics sequence.

4.3 Automatic Feature Extraction with Deep Learning

The need for manual feature engineering can be obviated by automated feature learning. Deep learning replaces the feature engineering process by an underlying system which typically consists of a neural network with multiple layers, that performs both feature learning and classification. With deep learning, one can start with raw data as features will be automatically created by the neural network when it learns. The main distinction between deep learning approaches for malware detection and classification is based on what they used as raw data. In particular, the researcher investigated the application of convolutional neural networks to automatically perform feature engineering on both the hexadecimal representation of malware's binary content [31, 26, 32] and its assembly language instructions [24, 27]. Below, a detailed description is provided of the methodology and results for each of the aforementioned research articles published during the development of this thesis.

4.3.1 Convolutional Neural Networks for Classification of Malware Assembly Code

The main idea behind this conference paper is to build a static classifier to group malware into families based on their assembly language source code without relying on the manual extraction of n-gram features. The assembly language source code of a computer program is the low-level representation of the program's statements and machine code instructions. Therefore, the problem of malware classification can be modeled as a text classification task by preprocessing the assembly files and extracting their assembly language instructions. The simplest representation is to retain only the mnemonic of the instruction. That is, on encountering the instruction *add esp 10h* we simply extract the *add* mnemonic. The main argument behind this representation is that it will generalize better as it would not be affected by small permutations in the arguments and thus, the obfuscation technique known as register reassignment would not alter the output of the classifier. Primarily, this obfuscation technique switches registers from generation to generation without altering the behavior of the program code.

To build the static classifier, Gibert et al [24] introduced a shallow convolutional neural network (CNN) architecture that extracts n-gram like features from malware's machine instructions. The overall structure of the shallow architecture is presented in Figure 4.3.1. Below, its layers are described in greater detail.

Input layer. The input of the network is the assembly language source code of a computer program represented as a concatenation of mnemonics

$$x_{1:n} = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

where n is the length of the program and $x_i \in \mathbb{R}^k$ corresponds to the i-th mnemonic in the program.

Embedding layer. Every mnemonic is represented as a low-dimensional vector of real values (word embedding). The rationale behind using distributed representations to encode the mnemonics is to better capture the semantic information about comparable operations or analogous meaning.

Convolutional layers. A convolution operation involves a filter $w \in \mathbb{R}^{hk}$ where h is the number of mnemonics to which it is applied and k is the size of the word embedding. In particular, filters are applied to sequences containing from 2 to 7 mnemonics.

A feature c_i is generated from a window of mnemonics $x_{i:i+h-1}$ (it comprises all mnemonics between position i and $i + h - 1$) and is defined as

$$c_i = f(w \cdot x_{i:i+h-1} + b),$$

where f is a rectifier linear unit (ReLU) function and b the bias term.

Max-pooling layers. The maximum value $\hat{c} = \max\{c\}$ is taken as the feature corresponding to the filter by applying the max pooling operator over the feature map.

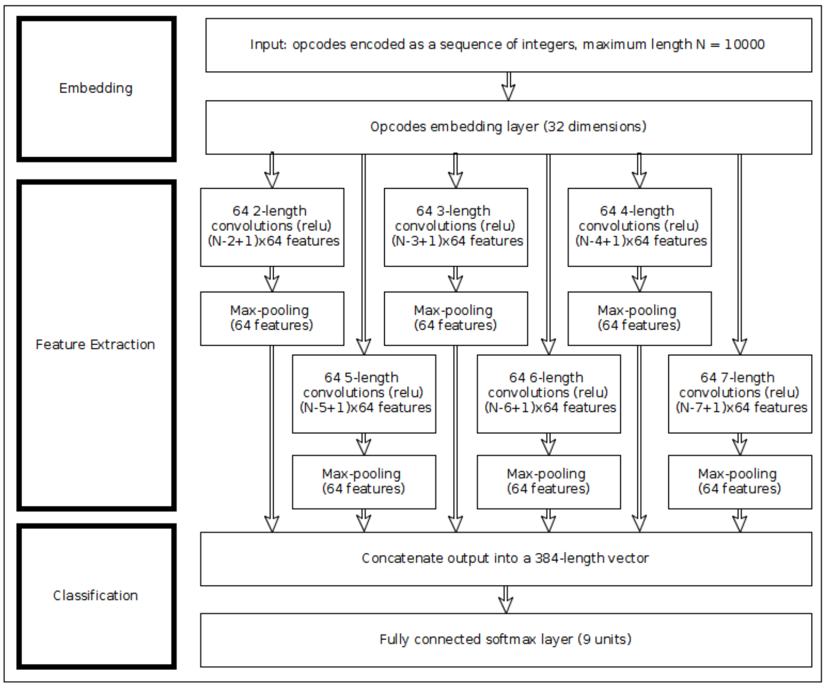

Embedding
Feature Extraction
Classification
Input: opcodes encoded as a sequence of integers, maximum length N = 10000
Opcodes embedding layer (32 dimensions)
64 2-length convolutions (relu) (N-2+1)x64 features
64 3-length convolutions (relu) (N-3+1)x64 features
64 4-length convolutions (relu) (N-4+1)x64 features
Max-pooling (64 features)
64 5-length convolutions (relu) (N-5+1)x64 features
64 6-length convolutions (relu) (N-6+1)x64 features
64 7-length convolutions (relu) (N-7+1)x64 features
Concatenate output into a 384-length vector
Fully connected softmax layer (9 units)

Figure 4.5: Shallow convolutional neural network architecture.

Softmax layer. The extracted features are passed to a fully-connected softmax layer whose output is the normalized probability distribution over families.

The main takeaways of this research article are as follows: (1) the required time for the feature extraction and classification process is lower than the computational time required to extract the N-grams for $N \geq 2$, as shown in Table 4.3.1; (2) although state-of-the-art multimodal approaches outperform our classifier, it has greater predictive power in comparison to opcode-based approaches in the literature and achieves higher classification accuracy than almost every subset of features in Ahmadi et al. [3]. See Table 4.3.1.

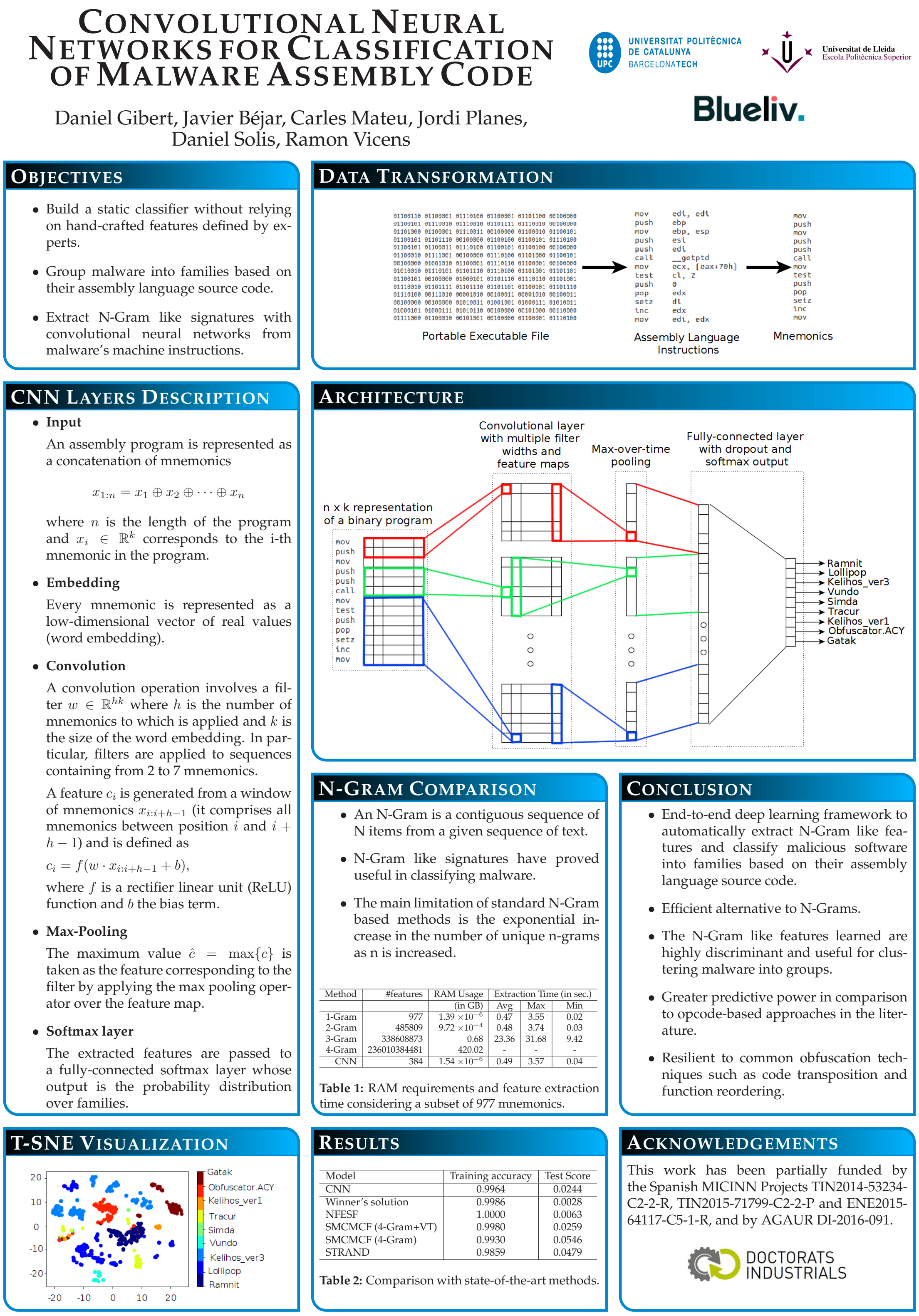
For a complete description of the experimentation the reader is referred to the original research article. This publication was honored by the Best Poster Award.

Table 4.2: Feature extraction comparison of our shallow cnn against n-grams.

Method	#features	RAM Usage	Extraction Time (in sec.)		
		(in GB)	Avg	Max	Min
1-Gram	977	1.39×10^{-6}	0.47	3.55	0.02
2-Gram	485809	9.72×10^{-4}	0.48	3.74	0.03
3-Gram	338608873	0.68	23.36	31.68	9.42
4-Gram	236010384481	420.02	-	-	-
CNN	384	1.54×10^{-6}	0.49	3.57	0.04

Table 4.3: Comparison of our CNN against other methods.

Model	Training accuracy	Private Score
CNN	0.9964	0.0244
Winner's solution	0.9986	0.0028
NFESF [3]	1.0000	0.0063
SMCMCF [37]	0.9980	0.0259
SMCMCF (4-Gram features) [37]	0.9930	0.0546
STRAND [19]	0.9859	0.0479



Convolutional Neural Networks for Classification of Malware Assembly Code

Daniel Gibert, Javier Béjar, Carles Mateu, Jordi Planes, Daniel Solis, Ramon Vicens

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

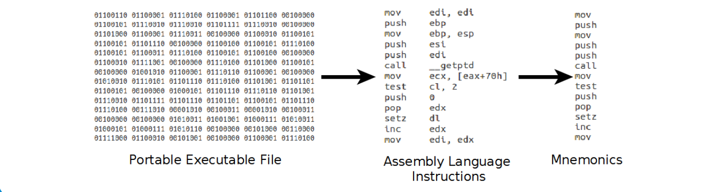
Universitat de Lleida Escola Politècnica Superior

Blueliv.

Objectives

- Build a static classifier without relying on hand-crafted features defined by experts.
- Group malware into families based on their assembly language source code.
- Extract N-Gram like signatures with convolutional neural networks from malware's machine instructions.

Data Transformation



CNN Layers Description

- **Input**

  An assembly program is represented as a concatenation of mnemonics

  $$x_{1:n} = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

  where n is the length of the program and $x_i \in \mathbb{R}^k$ corresponds to the i-th mnemonic in the program.

- **Embedding**

  Every mnemonic is represented as a low-dimensional vector of real values (word embedding).

- **Convolution**

  A convolution operation involves a filter $w \in \mathbb{R}^{hk}$ where h is the number of mnemonics to which is applied and k is the size of the word embedding. In particular, filters are applied to sequences containing from 2 to 7 mnemonics.

  A feature c_i is generated from a window of mnemonics $x_{i:i+h-1}$ (it comprises all mnemonics between position i and $i+h-1$) and is defined as

  $c_i = f(w \cdot x_{i:i+h-1} + b)$,

  where f is a rectifier linear unit (ReLU) function and b the bias term.

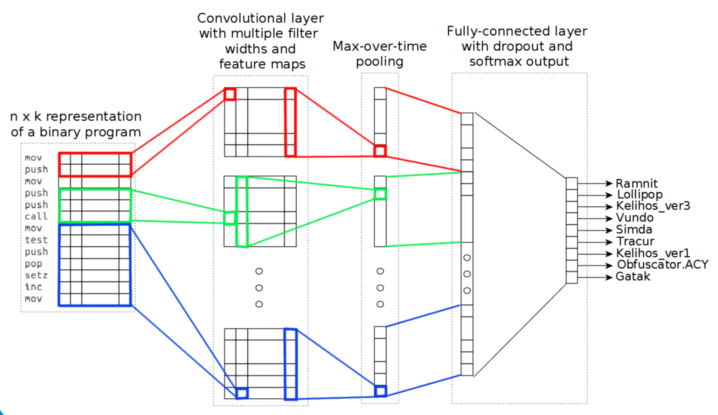
- **Max-Pooling**

  The maximum value $\hat{c} = \max\{c\}$ is taken as the feature corresponding to the filter by applying the max pooling operator over the feature map.

- **Softmax layer**

  The extracted features are passed to a fully-connected softmax layer whose output is the probability distribution over families.

Architecture


Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully-connected layer with dropout and softmax output
n x k representation of a binary program
mov
push
mov
push
push
call
mov
test
push
pop
setz
inc
mov
Ramnit
Lollipop
Kelihos_ver3
Vundo
Simda
Tracur
Kelihos_ver1
Obfuscator.ACY
Gatak

N-Gram Comparison

- An N-Gram is a contiguous sequence of N items from a given sequence of text.
- N-Gram like signatures have proved useful in classifying malware.
- The main limitation of standard N-Gram based methods is the exponential increase in the number of unique n-grams as n is increased.

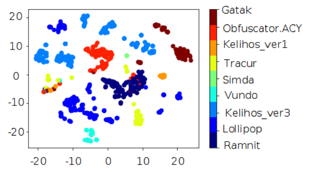
Method	#features	RAM Usage	Extraction Time (in sec.)		
		(in GB)	Avg	Max	Min
1-Gram	977	1.39×10^{-6}	0.47	3.55	0.02
2-Gram	485809	9.72×10^{-4}	0.48	3.74	0.03
3-Gram	338608873	0.68	23.36	31.68	9.42
4-Gram	236010384481	420.02	-	-	-
CNN	384	1.54×10^{-6}	0.49	3.57	0.04

Table 1: RAM requirements and feature extraction time considering a subset of 977 mnemonics.

Conclusion

- End-to-end deep learning framework to automatically extract N-Gram like features and classify malicious software into families based on their assembly language source code.
- Efficient alternative to N-Grams.
- The N-Gram like features learned are highly discriminant and useful for clustering malware into groups.
- Greater predictive power in comparison to opcode-based approaches in the literature.
- Resilient to common obfuscation techniques such as code transposition and function reordering.

T-SNE Visualization



Results

Model	Training accuracy	Test Score
CNN	0.9964	0.0244
Winner's solution	0.9986	0.0028
NFESF	1.0000	0.0063
SMCMCF (4-Gram+VT)	0.9980	0.0259
SMCMCF (4-Gram)	0.9930	0.0546
STRAND	0.9859	0.0479

Table 2: Comparison with state-of-the-art methods.

Acknowledgements

This work has been partially funded by the Spanish MICINN Projects TIN2014-53234-C2-2-R, TIN2015-71799-C2-2-P and ENE2015-64117-C5-1-R, and by AGAUR DI-2016-091.

DOCTORATS INDUSTRIALS

Recent Advances in Artificial Intelligence Research and Development 221
I. Aguiló et al. (Eds.)
IOS Press, 2017
© 2017 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-61499-806-8-221

Convolutional Neural Networks for Classification of Malware Assembly Code

Daniel GIBERT [a,c], Javier BÉJAR [b], Carles MATEU [c], Jordi PLANES [c], Daniel SOLIS [a], and Ramon VICENS [a]
[a] *Blueliv, Spain*
[b] *Technical University of Catalonia, Spain*
[c] *University of Lleida, Spain*

Abstract. Traditional signature-based methods have started becoming inadequnate to deal with next generation malware which utilize sophisticated obfuscation (polymorphic and metamorphic) techniques to evade detection. Recently, research efforts have been conducted on malware detection and classification by applying machine learning techniques. Despite them, most methods are build on shallow learning architectures and rely on the extraction of hand-crafted features. In this paper, based on assembly language code extracted from disassembled binary files and embedded into vectors, we present a convolutional neural network architecture to learn a set of discriminative patterns able to cluster malware files amongst families. To demonstrate the suitability of our approach we evaluated our model on the data provided by Microsoft for the BigData Innovators Gathering 2015 Anti-Malware Prediction Challenge. Experiments show that the method achieves competitive results without relying on the manual extraction of features and is resilient to the most common obfuscation techniques.

Keywords. Convolutional Neural Network, Malware Classification, Deep Learning

1. Introduction

Despite that the number of new malware grows exponentially every year [2], the methods used to defend against this threat remain almost unchanged. Traditionally, anti-virus solutions primarily relied in signature-based methods which struggle to keep up with the evolution of malware. A signature is an algorithm or hash that uniquely identifies a specific malware or a family. However, malware authors always try to stay a step ahead by creating new malware using metamorphic and polymorphic techniques to do not match signatures, thus to avoid detection.

In recent years, machine learning-based systems have been developed as a solution to defend against malware holding the promise of automating the work of detecting fresh malware. Several methods based on machine learning have been applied for classifying a file as malicious or legitimate or to classify malware in families but all of them rely on the extraction of hand-crafted features. However, the process of feature engineering is time-consuming and requires human resources to determine which features to extract and use for the classification process.

In this paper, we present a novel static approach which does not rely on the extraction of hand-crafted features to classify malware and is robust to modifications in the malware code. The method relies on training a Convolutional Neural Network (CNN), which would automatically extract features from the assembly language source code of malware and then classifies those samples into families. The dataset used to evaluate our approach was provided by Microsoft for the BigData Innovators Gathering (BIG 2015) Anti-Malware Prediction Challenge. The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 introduces our approach for malware classification. Section 4 evaluates the performance of our method in comparison with other approaches presented in the competition. Finally, Section 5 contains our concluding remarks.

2. Related Work

Most of the static detectors use N-Gram based features extracted from byte sequences or instruction opcodes. An N-Gram is a contiguous sequence of N items from a given sequence of text. Tesauro et al. [10] proposed a byte-sequence N-Gram based algorithm for malware classification which was part of the IBM's antivirus scanner. The algorithm extracted a list of trigrams and used a neural network as the classification model. Similar to byte-sequence N-grams, opcode N-gram patterns have been used in the literature to detect and classify malware [8,7]. An opcode (abbreviated from operation code) is the portion of a machine language instruction that specifies the operation to be performed. Almost all approaches in literature based on N-Gram features are composed by three main steps. First, N-Grams are extracted. Second, an algorithm is applied to reduce the dimensionality of the data and to select the most discriminant subset of features. Third, an algorithm classifies the samples based on the features selected in the previous step.

The main drawback of the representation based on the N-Grams is that it is dependent on N, the number of words that each N-Gram will contain. Thus, computing all N-Grams where N is greater than 3 is not feasible (cf. Section 4) due to the RAM requirements and the decrease in performance. For that reason, we present an efficient alternative to N-Gram counts that automatically extracts the most discriminative features from a sequence of opcodes without having to apply any feature selection technique to make the problem tractable.

3. Convolutional Neural Network Model

The model architecture is a variant of Kim's CNN architecture [6]. The main difference between Kim's model and ours is two fold: (i) their is trained by natural language words and ours is trained by opcodes; (ii) and the size and the number of the feature maps in the convolutional layer. In their work, the convolutional layer contains a total of 300 feature maps with 100 feature maps of size $h \times k$ for every $h \in \{3, 4, 5\}$, where h is the number of words the filter comprises and k is the size of the embedding. Instead, our convolutional layer has 64 feature maps of size $h \times k$ for every $h \in \{2, 3, 4, 5, 6, 7\}$.

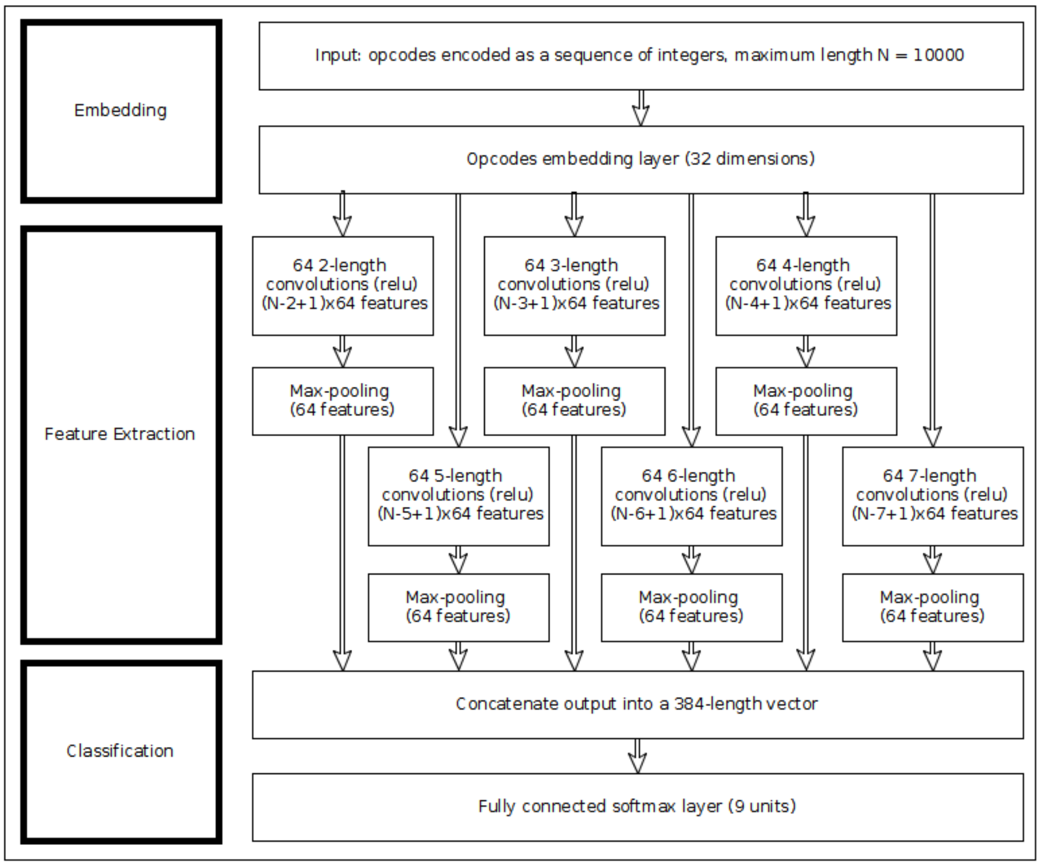

Embedding
Feature Extraction
Classification
Input: opcodes encoded as a sequence of integers, maximum length N = 10000
Opcodes embedding layer (32 dimensions)
64 2-length convolutions (relu) (N-2+1)x64 features
64 3-length convolutions (relu) (N-3+1)x64 features
64 4-length convolutions (relu) (N-4+1)x64 features
Max-pooling (64 features)
Max-pooling (64 features)
Max-pooling (64 features)
64 5-length convolutions (relu) (N-5+1)x64 features
64 6-length convolutions (relu) (N-6+1)x64 features
64 7-length convolutions (relu) (N-7+1)x64 features
Max-pooling (64 features)
Max-pooling (64 features)
Max-pooling (64 features)
Concatenate output into a 384-length vector
Fully connected softmax layer (9 units)

Figure 1. The neural network architecture of our CNN model.

As shown in Figure 1, it has the following layers: (1) Opcodes are embedded as multi-dimensional vectors, one per opcode; (2) Feature maps slide over opcode sequences to find patterns, and the maximum value of each of the feature map activations is assigned into the resulting feature vector; And (3), the convolution-based feature vector is fed into one fully connected softmax layer which outputs the probability distribution over malware families.

The input of the network is an assembly program represented as the concatenation of opcodes, with every opcode represented as a vector of k real numbers (word embedding). The rationale behind using distributed representations of opcodes is to better capture the meaning of opcode as input. The goal is to learn vector representations where those corresponding to opcodes used to transfer data, e.g. `Push` and `Pop`, are similar and distinct from other representations corresponding to opcodes used in arithmetic and logical instructions.

A convolution operation involves a filter $w \in \mathbb{R}^{hk}$ where h is the number of opcodes to which is applied and k is the size of the word embedding. In particular, a feature c_i generated from a window of opcodes $x_{i:i+h-1}$ (it comprises all

Table 1. Feature extraction comparison of our method against N-Grams

Method	#features	RAM Usage	Extraction Time (in sec.)		
		(in GB)	Avg	Max	Min
1-Gram	977	1.39×10^{-6}	0.47	3.55	0.02
2-Gram	485809	9.72×10^{-4}	0.48	3.74	0.03
3-Gram	338608873	0.68	23.36	31.68	9.42
4-Gram	236010384481	420.02	-	-	-
CNN	384	1.54×10^{-6}	0.49	3.57	0.04

opcodes between position i and $i+h-1$ from the sequence of assembly program opcodes) is defined as $c_i = f(w \cdot x_{i:i+h-1} + b)$, where f is a rectifier linear unit (ReLU) function and b the bias term. The purpose of the convolution operation is to extract features from the assembly program while preserving the spatial relationship between opcodes. Thus, a filter is applied to every possible window of opcodes in an assembly program $x_{i,h}, x_{2:h+1}, \ldots, x_{n-h+1:n}$ to produce a feature map $c = [c_1, c_2, \ldots, c_{n-h+1}]$ with $c \in \mathbb{R}^{n-h+1}$. Then, the maximum value is taken as the feature corresponding to the filter by applying the max pooling operator [3] over the feature map. This process is applied for various filters with varying window sizes to obtain multiple features. Finally, the extracted features are passed to a fully connected softmax layer whose output is the probability distribution over families. Dropout [9] was employed as a regularization mechanism, which randomly drops a proportion of units during forward propagation and prevents the co-adaptation between neurons on the fully-connected softmax layer.

4. Evaluation

The data used to evaluate our approach was provided by Microsoft for the Big-Data Innovators Gathering (BIG 2015) Anti-Malware Prediction Challenge. The dataset is composed of 21741 samples, 10868 for training and 10873 for testing, whose malware binaries implement common obfuscation techniques [11] and represent 9 different malware families: `Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY` and `Gatak`.

The first experiment performed is the comparison of our method against N-Gram feature extraction, shown in Table 1. The CNN has been trained using a Nvidia GeForce GTX 1080 Ti. We can observe that our method is similar in time and memory usage to the 1-Gram method.

The second experiment is the comparison of our method in terms of malware classification accurancy. The performance of our model is evaluated using the following formula for the logarithmic loss:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(y_{i,j}log(p_{i,j}) + (1-y_{i,j})log(1-p_{i,j}))$$

where N is the number of observations, M is the number of class labels, log is the natural logarithm, $y_{i,j}$ is 1 if the observation i is in class j and 0 otherwise, and $p_{i,j}$ is the predicted probability that observation i is in class j. Our model achieved a score of 0.0244 in the test set which is an improvement of 98.89% with respect to the equal probability benchmark (logloss=2.1972) which is obtained by submitting 1/9 for every prediction.

In Table 2 we show the comparison results of our CNN against the following methods: Novel Feature Extraction, Selection and Fusion (NFESF) for Effective Malware Family Classification [1]; Scalable Malware Classification with Multi-faceted Content Features (SMCMCF) and threat intelligence [5]; and Polymorphic Malware Detection Using Sequence Classification Methods (STRAND) [4].

Even that the results are quite promising, they are not as good as the ones obtained by the winner's solution or NFESF results [1]. We think the reason is that their approaches relied on the extraction of domain expert hand-crafted features and their combination. However, it outperforms the N-Gram based approach presented in [5] and the results obtained by almost every subset of features in [1]. Additionally, the required time of feature extraction is lower than other approaches and, in particular, the extraction of N-Grams for $N \geqslant 2$, as shown in Table 1. The extraction time for 4-Gram was not able to be computed due to their RAM requirements.

Table 2. Comparison of our CNN against other methods.

Model	Training accuracy	Private Score
CNN	0.9964	0.0244
Winner's solution	0.9986	0.0028
NFESF	1.0000	0.0063
SMCMCF	0.9980	0.0259
SMCMCF (4-Gram features)	0.9930	0.0546
STRAND	0.9859	0.0479

5. Conclusion and Future Work

In the present work, we studied the problem of classifying malware into their corresponding malware families. In order to tackle with this problem, we used one of the most recent and biggest datasets publicly available which was provided by Microsoft for the BigData Innovators Gathering Cup (BIG 2015). As far as we know, we presented the first method that applies convolutional neural networks to automatically generate features from the assembly language source code. The method achieved a logarithmic loss of 0.0244 in the competition, which is an improvement of 98.89% with respect to the equal probability benchmark.

The results obtained are quite promising in terms of accuracy and computational time. Firstly, the accuracy obtained in the training set is higher than those obtained by almost every independent subset of features in [1] and outperforms the results of the N-Gram based approach presented in [5]. Secondly, it provides a good alternative to N-grams, where $N \geq 2$ for larger datasets, as the computational time required to extract the features and classify a sample is almost lower than calculating 2-Gram counts. And finally, the nature of convolutional neural networks has proved to be resilient to the most common obfuscation techniques [11].

Acknowledgments

We would like to thank the Blueliv Labs team, specially Ferran Pichel and Àngel Puigventós for their support and the feedback provided during the development of this work.

This work has been partially funded by the Spanish MICINN Projects TIN2014-53234-C2-2-R, TIN2015-71799-C2-2-P and ENE2015-64117-C5-1-R, and by AGAUR DI-2016-091.

References

[1] Mansour Ahmadi, Giorgio Giacinto, Dmitry Ulyanov, Stanislav Semenov, and Mikhail Trofimov. Novel feature extraction, selection and fusion for effective malware family classification. *CoRR*, abs/1511.04317, 2015.

[2] AV-TEST Institute. Malware statistics. `https://www.av-test.org/en/statistics/malware/`. Accessed: 2017-04-23.

[3] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011.

[4] Jake Drew, Michael Hahsler, and Tyler Moore. Polymorphic malware detection using sequence classification methods and ensembles. *EURASIP Journal on Information Security*, 2017(1):2, 2017.

[5] Xin Hu, Jiyong Jang, Ting Wang, Z. Ashraf, Marc Ph. Stoecklin, and Dhilung Kirat. Scalable malware classification with multifaceted content features and threat intelligence. *IBM Journal of Research and Development*, 60(4):6, 2016.

[6] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[7] Robert Moskovitch, Dima Stopel, Clint Feher, Nir Nissim, and Yuval Elovici. Unknown malcode detection via text categorization and the imbalance problem. *IEEE International Conference on Intelligence and Security Informatics*, pages 156–161, 2008.

[8] Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. In *Security Informatics*. Springer, 2012.

[9] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[10] G.J. Tesauro, J.O. Kephart, and Gregory B Sorkin. Neural networks for computer virus recognition. *IEEE International Conference on Intelligence and Security Informatics*, 11, 1996.

[11] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, BWCCA '10, pages 297–300, Washington, DC, USA, 2010. IEEE Computer Society.

4.3.2 A Hierarchical Convolutional Neural Network for Malware Classification

Gibert et al. [24] trained a learning system feeding as input the sequence of assembly language instructions extracted from the assembly language source code. However, the aforementioned approach does not take into account the hierarchical structure of Portable Executable files. This type of files exhibits various levels of spatial correlation. Adjacent code instructions tend to be related to one another but this is not always the case, as function calls and jump commands produce discontinuities. For instance, the jump instruction transfers the control of the program to a different point in the instruction stream. Moreover, these discontinuities are maintained when treating the binary as a sequence of byte values, thus resulting in unrelated adjacent chunks of bytes. Nevertheless, a Portable Executable consists of functions or macros and each function is defined by a sequential list of instructions. Consequently, by representing an executable as a sequence of instructions without considering the macros, we lose its hierarchical information. As a result, a more natural way to represent the assembly language source code would be as a sequence of sequences, where each sequence contains the mnemonics describing a particular function or procedure.

Thus, to capture the insights at the mnemonics-level and at the function-level, we designed a hierarchical convolutional network to extract hierarchical features about the malware structure. See Figure 4.6. This architecture has two convolutional blocks: (1) one at the mnemonics level and (2) one at the function level.

Mnemonics-level . The mnemonics-level block consists of multiple filters of various sizes, $1 \times F \times K$, where $F \in \{1, 2, 3, 5, 7\}$. That is, the size of the subsequence of mnemonics that the filter can detect ranges from 1 to a maximum of 7 mnemonics. The reason behind applying the convolution operator with filters of different size is because salient and prominent parts in the sequence of instructions can vary in size and location. Consequently, choosing the right kernel size for the convolutional operation might be a difficult choice. Notice that the convolutions are applied to each function individually and subsequently, max-pooling is applied to keep only the strongest activation for each feature map per function.

Function-level . The function-level block takes as input the features extracted from each function as an $N \times L$ array, where N is the number of functions in a given malware sample and L is the size of the feature vector per function, and performs convolutions on the input with filters of size $1 \times L$, $2 \times L$ and $3 \times L$ which extract features from one, two or three functions at a time. Then, we apply both global max-pooling and global average-pooling to generate the program's feature vector representation.

For a detailed description of the architecture the reader is referred to the original publication [27].

The generalization performance of the hierarchical convolutional neural network was compared with state-of-the-art methods in the literature [32, 31, 26, 63, 49, 51, 24, 56, 56, 73] on the Microsoft Malware Classification benchmark [66]. See Table 4.4. Results show that methods that are fed with the raw byte sequences

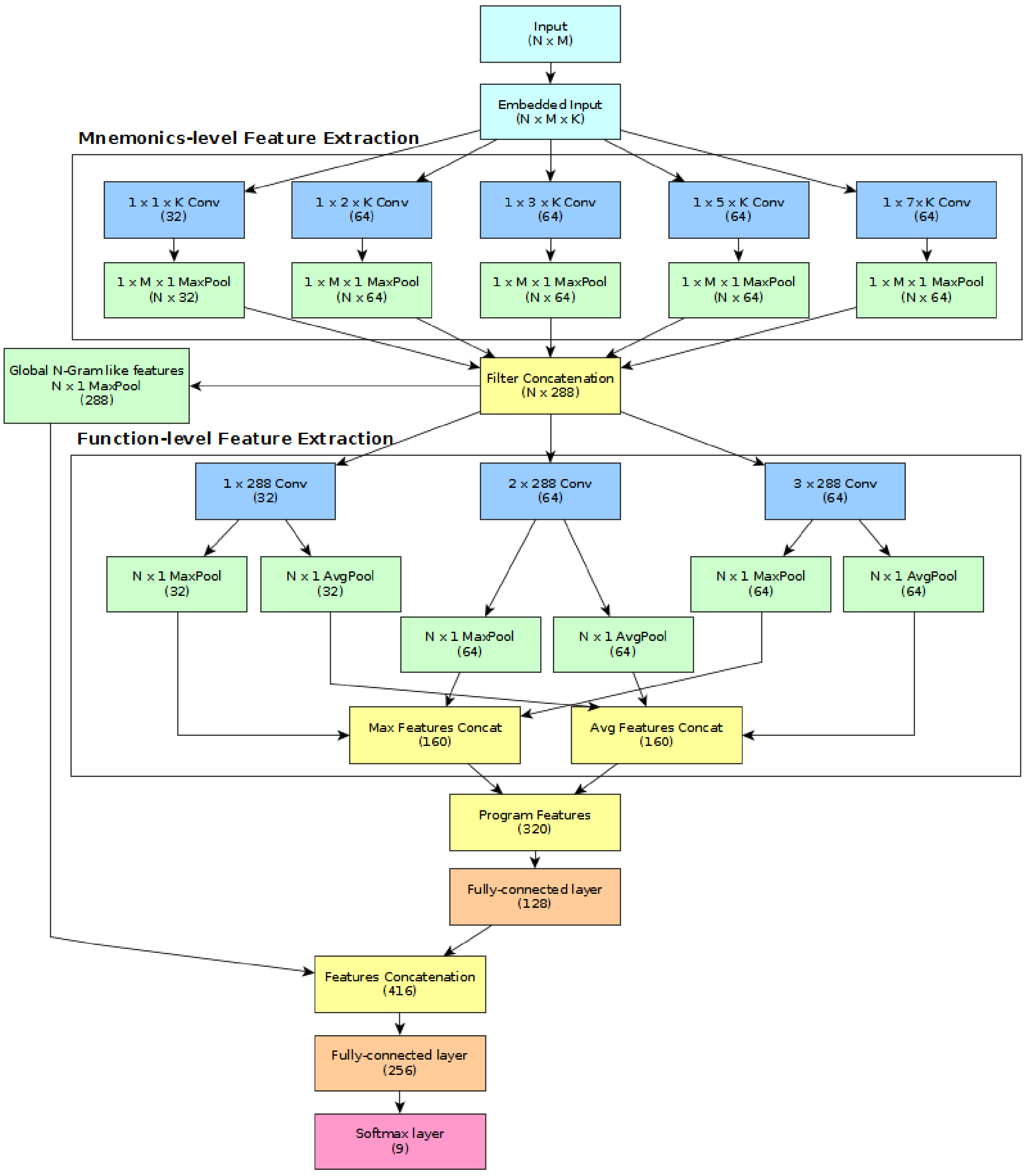

Input (N x M)
Embedded Input (N x M x K)
Mnemonics-level Feature Extraction
1 x 1 x K Conv (32)
1 x 2 x K Conv (64)
1 x 3 x K Conv (64)
1 x 5 x K Conv (64)
1 x 7 x K Conv (64)
1 x M x 1 MaxPool (N x 32)
1 x M x 1 MaxPool (N x 64)
1 x M x 1 MaxPool (N x 64)
1 x M x 1 MaxPool (N x 64)
1 x M x 1 MaxPool (N x 64)
Global N-Gram like features N x 1 MaxPool (288)
Filter Concatenation (N x 288)
Function-level Feature Extraction
1 x 288 Conv (32)
2 x 288 Conv (64)
3 x 288 Conv (64)
N x 1 MaxPool (32)
N x 1 AvgPool (32)
N x 1 MaxPool (64)
N x 1 AvgPool (64)
N x 1 MaxPool (64)
N x 1 AvgPool (64)
Max Features Concat (160)
Avg Features Concat (160)
Program Features (320)
Fully-connected layer (128)
Features Concatenation (416)
Fully-connected layer (256)
Softmax layer (9)

Figure 4.6: Structure of the Hierarchical Convolutional Neural Network (HCNN) for malware classification. Blue, green and yellow boxes represent the convolution, pooling and concatenation operations, respectively.

as input [63, 49] perform worse than those that first compress or encode the information in the byte sequences [31, 26]. Our hypothesis is that, due to the limited amount of training data and the higher complexity of the network, those methods are more prone to overfitting. On the other hand, assembly-based approaches generally perform considerably better than hexadecimal-based approaches, with the exception of the Hierarchical Attention Network (HAN). However, even though the HAN architecture achieved a lower 10-fold cross validation accuracy and macro F1 score, it achieved a lower logarithmic loss with the test set in comparison with the hexadecimal-based approaches. Furthermore, both the Shallow CNNs and the Hierarchical Convolutional Neural Network (HCNN) achieved a higher classification accuracy and macro F1 score, and a lower logloss than other approaches. This was for two reasons: (1) The length of the sequence of mnemonics is much shorter than the length of the raw byte sequences; thus, given the limited training data, the simpler architectures are less prone to overfitting; (2) Both the Shallow CNNs and the Hierarchical Convolutional Network perform convolutions on the mnemonic sequences. Applying convolutions rather than recurrent units is computationally more efficient and is more suitable for our classification task, since we are dealing with very long input sequences. Furthermore, malware authors usually employ a wide range of obfuscation techniques to modify the appearance of executables without modifying their behavior. One of the most common obfuscation techniques is subroutine reordering. That is, the order of the subroutines in the original code is changed randomly. Another common technique is code transposition, which reorganizes the order of the instructions without changing the behavior of the computer program. Thus, convolutional-based networks are more adequate to deal with the problem at hand, since they are able to detect patterns that might be displaced in space through the convolution and max-pooling operations.

Table 4.4: State of the art comparison of deep learning methods on the Microsoft Malware Classification Challenge benchmark. Approaches with a "*" mark indicate that they performed 5-fold cross validation instead of 10-fold cross validation to assess the performance of their method.

	Training 10-fold cross validation		Test
	Accuracy	Macro F1 score	Logarithmic Loss
Hex-based approaches			
CNN IMG [32]	0.975	0.940	0.1844
CNN Entropy [31]	0.9708	0.9314	0.1346
CNN Haar Approximation & Details [31]	0.9828	0.9636	0.1244
Autoencoder + Dilated Residual Network [26]	0.9861	0.9719	0.1063
MalConv [63]	0,9641	0.8902	0.3071
DeepConv [49]	0.9756	0.9071	0.1602
CNN+BiLSTM [51]*	0.9820	0.9605	0.0744
Assembly-based approaches			
Shallow CNN: filters with multiple sizes [24]	0.9917	0.9856	0.0351
Shallow CNN [56]	0.9903	0.9743	0.0515
HAN [73]	0.9742	0.9468	0.0933
Hierarchical Convolutional Network	0.9913	0.9830	0.0419

A Hierarchical Convolutional Neural Network for Malware Classification

Daniel Gibert
Dept. of Computer Science
University of Lleida
daniel.gibert@diei.udl.cat

Carles Mateu
Dept. of Computer Science
University of Lleida
carlesm@diei.udl.cat

Jordi Planes
Dept. of Computer Science
University of Lleida
jordi.planes@diei.udl.cat

***Abstract*—Malware detection and classification is a challenging problem and an active area of research. Particular challenges include how to best treat and preprocess malicious executables in order to feed machine learning algorithms. Novel approaches in the literature treat an executable as a sequence of bytes or as a sequence of assembly language instructions. However, in those approaches the hierarchical structure of programs is not taken into consideration. An executable exhibits various levels of spatial correlation. Adjacent code instructions are correlated spatially but that is not necessarily the case. Function calls and jump commands transfer the control of the program to a different point in the instruction stream. Furthermore, these discontinuities are maintained when treating the binary as a sequence of byte values. In addition, functions might be arranged randomly if addresses are correctly reorganized. To address these issues we propose a Hierarchical Convolutional Network (HCN) for malware classification. It has two levels of convolutional blocks applied at the mnemonic-level and at the function-level, enabling us to extract n-gram like features from both levels when constructing the malware representation. We validate our HCN method on the dataset released for the Microsoft Malware Classification Challenge, outperforming almost every deep learning method in the literature.**

***Index Terms*—Malware Classification, Machine Learning, Deep Learning, Hierarchical Convolutional Neural Network**

I. INTRODUCTION

Malware as a business is on the rise. It is a booming criminal industry worth billions of dollars, involving networks of developers and criminal organizations, that grows more and more every year. In particular, ransomware as a service (RaaS) has been a growing problem, with Cerber as its most prolific family. Regarding the financial industry, the Ramnit family is still the most prevalent banking malware in proportion to the total number of cyber-attacks on banks. The first Ramnit variants emerged in 2010 and it has been evolving over the years to include new capabilities and evasion and anti-detection techniques. This trend has been replicated throughout the malware landscape, with a family originating from a single source base and its variants exhibiting a set of consistent behaviors and characteristics but with increased capabilities and anti-detection techniques. Thus, being able to group malware into families according to these shared characteristics has proven useful for detecting and classifying unseen programs.

This research has been partially funded by the Spanish MICINN Projects TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, and is supported by the University of Lleida. This research article has received a grant for its linguistic revision from the Language Institute of the University of Lleida (2018 call).

Over the past decade, there has been an increase in the research and deployment of machine learning solutions for malware detection due to the confluence of three recent developments: (1) the availability of labeled feeds of malware for research, (2) the bidding down of computational power and (3) the breakthroughs in the machine learning field. As a result, machine learning has become an appealing signature-less approach for detecting malicious software due to its ability to summarize complex relationships among the input data and its subsequent decision-making. On the one hand, traditional machine learning solutions perform feature engineering to manually extract features that act as an abstract representation of malware and serve as input for a classifier. Consequently, the success of machine learning algorithms depends almost entirely on the features extracted. On the other hand, end-to-end learning solutions eliminate the need to extract hand-designed features and, instead, they take an executable as input and try to directly recognize whether or not it is malicious. Nevertheless, those approaches still need to perform some kind of preprocessing process to represent the malware's content in a way the machine learning algorithm can understand. For instance, E. Raff et al. [1] and D. Gibert et al. [2] trained a learning system feeding as input the hexadecimal representation of malware's binary content and the sequence of assembly language instructions extracted from the assembly language source code, respectively. However, the aforementioned solutions do not take into account the hierarchical structure of malware. That is, binary executables exhibit various levels of spatial correlation. In particular, adjacent code instructions tend to be related to one another but this is not always the case, as function calls and jump commands produce discontinuities. For instance, the jump instruction transfers the control of the program to a different point in the instruction stream. Moreover, these discontinuities are maintained when treating the binary as a sequence of byte values, thus, resulting in unrelated adjacent chunks of bytes.

Our primary contribution is a new neural architecture, called Hierarchical Convolutional Network, that is designed to capture the insights about malware structure at the mnemonics level and at the function level. Considering that Portable Executable (PE) files have a hierarchical structure, that is, a computer program is composed of functions and functions consist of instructions, we introduce a hierarchical network architecture to construct a program representation by first building representations of their functions, and then building the later representation by extracting and combining both the mnemonic-level features and the function-level representations, which allows us to retain the hierarchical information of an executable. The generalization performance of our method has been evaluated on the dataset provided by Microsoft for the Big Data Innovators Gathering (BIG 2015) Anti-Malware Prediction Challenge [3]. Furthermore, we present a comparison with deep learning methods in the literature.

The rest of the paper is organized as follows. Section II briefly presents the research related to this work. Section III provides a complete description of the proposed methodology for classifying malware. Section IV describes the results achieved by our method on the Microsoft Malware Classification Challenge benchmark [3] and provides a comparison with the deep learning methods in the literature. Lastly, Section V summarizes the concluding remarks extracted from this work and proposes some future lines of research.

II. Related Work

Traditional machine learning solutions for malware detection extract hand-designed features that provide an abstract representation of the program that is later used for classification. These solutions are greatly dependent on the ability of the domain experts to extract a set of descriptive and discriminant features into which they represent malware. The process that transforms raw data into a feature vector is known as feature engineering. The most common features are bytes and opcode n-grams [4]–[6]. An n-gram is a sequence of n items. By treating a malware file as a sequence of bytes (or opcodes), n-grams are extracted by looking at the unique combination of every n consecutive byte (or opcode) as an individual feature. n-gram based approaches in the literature consist of a three-step process. First, n-grams are extracted. Second, the features are reduced and only a subset of them is selected. Third, an algorithm classifies the samples based on the reduced feature subset. To detect the presence of compressed and encrypted segments, entropy analysis has long been used because packed and encrypted executables usually have higher entropy than native code. For instance, in the study by Lydia et al. [7] the average entropy of native, compressed and encrypted executables was 5.099, 6.801 and 7.175, respectively. However, simple entropy statistics are not enough to detect sophisticated malware, as malware authors are able to conceal packed and encrypted code in a way that they pass through entropy filters without much effort. Thus, researchers started analyzing the structural entropy of executables [8]. In other words, an executable is represented as an entropy time series, where each value measures the entropy over a small chunk of code in a specific location of the file. Portable Executable (PE) files also contain useful information associated with dynamically linked libraries, sections of the programs, etc., that can be used to build descriptive features. In particular, the usage of the Windows Application Programming Interface (Windows API) has long been used as discriminant features for malware detection [9]. Basically, API functions and system calls provide information related to which services of the operating system the executable might access.

Following recent trends in the machine learning field, during the past few years the development of machine learning methods for malware detection has evolved towards deep learning solutions. These solutions replace the traditional machine learning workflow by a fully trainable system with as little preprocessing as possible. A deep learning system takes as input a representation of the executable's content and tries to directly detect malicious software or the family to which it belongs. Neural networks are commonly used in such end-to-end learning systems. In a deep learning setup, an end-to-end model learns all the features that can occur between the original input (x) and the final output (y). For malware detection tasks, an end-to-end model is trained to generate an output prediction (y) from an input executable represented in one format or another. D.Gibert et al [10] presented a convolutional neural network to classify malware based on the representation of its binary content as gray scale images. E. Raff et al [1] and M. Krčál [11] proposed a shallow and a deep convolutional network for detecting malicious software from its raw byte sequences over millions of bytes. On the contrary, instead of feeding a learning algorithm with the raw bytes sequences, D. Gibert et al. [12], [13] proposed firstly compressing or codifying the information using entropy analysis or autoencoders to reduce the usage of computational resources in end-point solutions. B.Kolosnjaji et al. [14] constructed a neural network based on convolutional and recurrent layers to extract features from malware represented as a sequence of API function calls. D. Gibert et al [2] presented a shallow architecture to extract n-gram like signatures of malware represented as a sequence of assembly language instructions.

III. The Problem of Malware Classification

Polymorphic and metamorphic techniques are commonly employed by malware authors to evade detection. These techniques change the appearance of the computer program without modifying its behavior and true purpose. Thus, huge volumes of different files are generated that might be variants of previously known files. Consequently, being able to identify these malicious files belonging to the same *family* is an effective approach to analyzing and classifying such a large amount of files. As a result, malware classification refers to the task of grouping malware into groups and identifying their respective families. The rest of the section is organized as follows. Section III-A introduces the Microsoft Malware Classification Challenge benchmark [3]. Section III-B describes how malware classification can be treated as a document classification task. Lastly, Section III-C presents a hierarchical convolutional network for classifying malware into families.

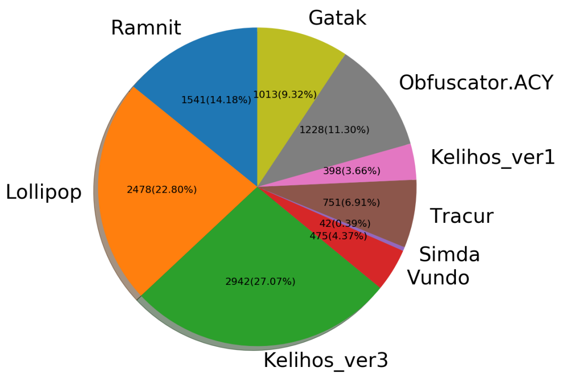
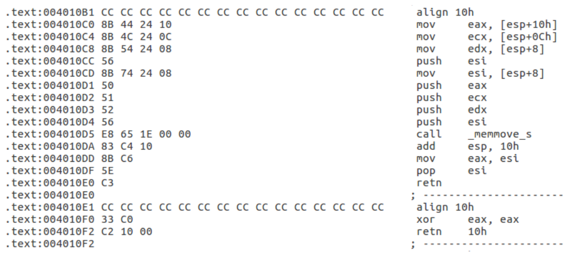


Fig. 1. Class distribution of the Microsoft Malware Classification training set



```
.text:004010B1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010C0 8B 44 24 10                                      mov     eax, [esp+10h]
.text:004010C4 8B 4C 24 0C                                      mov     ecx, [esp+0Ch]
.text:004010C8 8B 54 24 08                                      mov     edx, [esp+8]
.text:004010CC 56                                               push    esi
.text:004010CD 8B 74 24 08                                      mov     esi, [esp+8]
.text:004010D1 50                                               push    eax
.text:004010D2 51                                               push    ecx
.text:004010D3 52                                               push    edx
.text:004010D4 56                                               push    esi
.text:004010D5 E8 65 1E 00 00                                   call    _memmove_s
.text:004010DA 83 C4 10                                         add     esp, 10h
.text:004010DD 8B C6                                            mov     eax, esi
.text:004010DF 5E                                               pop     esi
.text:004010E0 C3                                               retn
.text:004010E0                                                 ; ---------------------
.text:004010E1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010F0 33 C0                                            xor     eax, eax
.text:004010F2 C2 10 00                                         retn    10h
.text:004010F2                                                 ; ---------------------
```

Fig. 2. Assembly View of a Portable Executable file

A. *Microsoft Malware Classification Challenge*

The Microsoft Malware Classification Challenge [3] is currently hosted on Kaggle and it is available to the public for research purposes. It has become the standard benchmark to assess the performance of machine learning algorithms on addressing the task of malware classification. The dataset consists of 10868 malicious samples for training and 10873 samples for testing of malware representing 9 malware families. (See Figure 1) For each sample we are provided with the hexadecimal representation of the malware's binary content and with a metadata manifest, which is the assembly language source code generated using the IDA disassembler tool, a computer program that translates machine language into assembly language.

B. *Malware Classification as a Document Classification Task*

The assembly language source code of a computer program is the low-level representation of the program's statements and machine code instructions. Therefore, the problem of malware classification can be modeled as a text classification task by preprocessing the assembly files and extracting their assembly language instructions. The simplest representation is to retain only the mnemonic of the instruction. That is, on encountering the instruction `add esp 10h` we simply extract the `add` mnemonic. The main argument behind this representation is that it will generalize better as it would not be affected by small permutations in the arguments and thus, the obfuscation technique known as register reassignment would not alter the output of the classifier. Primarily, this obfuscation technique switches registers from generation to generation without altering the behavior of the program code.

Some of the most frequent approaches in the literature for classifying malware from the instructions of the assembly language source code are based on n-gram analysis [5], [15], [16]. In our domain, an n-gram is defined as a contiguous sequence of n mnemonics from a given sequence of instructions. The n-gram features can be used to train a classifier to distinguish between benign and malicious software. However, these approaches are greatly affected by the length of the n-gram. By increasing n, it increases the resulting n-gram features and also the computational resources needed. Furthermore, the number of unique combinations jointly increase exponentially with N. Thus, it is necessary to perform feature selection and reduction to cut down the size of the feature vector, which would be composed of millions of elements for long n-grams.

To deal with long n-grams without consuming an exploding amount of computational resources, the use of convolutional neural networks [2], [17] has recently been proposed to learn to detect n-gram like patterns from a computer program represented as a sequence of mnemonics. Unlike the n-gram based approaches, these approaches do not need to exhaustively enumerate a large number of n-grams during training, as the CNN is able to learn n-gram like signatures through the various convolutional layers. In addition, they also remove the need for hand-designed features, as the features are also learned by the convolutional layers. Thus, it eliminates the need for a pipeline consisting of feature extraction, feature selection/reduction and classification, as both steps are optimized together during supervised network training.

Although the aforementioned neural approaches to malware classification performed considerably well, they do not take into account the hierarchical structure of Portable Executable files. This type of files exhibits various levels of spatial correlation. Adjacent instructions tend to be related to one another but, due to jumps and function calls, this might not always be true. Function calls and jump instructions transfer the control of the program into another address in memory and continue the execution from that address. Furthermore, these discontinuities are maintained on the binary file, but also in its hexadecimal representation. Nevertheless, a Portable Executable consists of functions or macros and each function is defined by a sequential list of instructions. Consequently, by representing an executable as a sequence of instructions without considering the macros, we lose its hierarchical information. On the contrary, a natural way to represent the assembly language source code would be as a sequence of sequences, where each sequence contains the mnemonics describing a particular function or procedure. For instance, the assembly language code in Figure 2 would be decomposed into two sequences of mnemonics (any operands associated with the assembly language instruction are discarded), as follows:

- Function A (from 004010B1 to 004010E0): `[mov,`

`mov, mov, push, mov, push, push, push, push, call, add, mov, pop, retn]`

- Function B (from 004010E1 to 004010F2): `[xor, retn]`

Thus, to capture the insights at the mnemonic-level and at the function-level, we designed a hierarchical convolutional network to extract hierarchical features about malware structure.

C. Hierarchical Convolutional Network for Classifying Malware

The overall structure of the hierarchical convolutional network is shown in Figure 3. Next, the components and structure of the network are described in greater detail.

Notation. The dataset consists of a set of pairs x^i, y^i, where x^i is an executable and y^i denotes the family to which it belongs, where $x^i \in \mathbb{Z}^{N \times M \times I}$, N is the number of functions, M is the number of mnemonics and I is the vocabulary size. The vocabulary is composed of all mnemonics that have appeared at least three times in the training set. Each mnemonic is associated with a number within the range 1 to I. For instance, $x^i_{j,l}$ refers to the one-hot vector of the l-th mnemonic of the j-th function in program i. Note that each mnemonic is represented as a one-hot vector, which is a vector of zeros of size I, with a '1' in the position corresponding to the mnemonic's integer mapping. The remaining mnemonics that are executed less than three times are replaced with the UNK token. In addition, each sequence of mnemonics referring to a function is padded to the same length. Furthermore, empty functions are also padded to represent all programs with the same size. In all our experiments N and M are set to 20000 and 50, respectively. In those cases where the number of mnemonics per function is greater than M, the sequence is split into subsequences of size M.

Embedding layer. One-hot vectors are high-dimensional and sparse. Using such encoding, it is not possible to meaningfully compare mnemonic vectors other than by equality testing. To address this problem, a distributed representation of a mnemonic is used. The embedding layer converts a one-hot vector into a low-dimensional vector representation of size K. Thus, the output of the embedding layer would be a 3-dimensional array of size $N \times M \times K$, where N is the number of functions per program, M is the number of mnemonics per function and K is the embedding size. Each mnemonic would be represented as a distribution of weights across K elements where each element in the vector contributes to the definition of many mnemonics. The embedding space may encode semantic information about comparable operations or analogous meaning. This is achieved by projecting those mnemonics to nearby points in the embedding space.

Mnemonics-level Feature Extraction. This convolutional block receives the $N \times M \times K$ program representation as input. It consists of multiple filters of various sizes, $1 \times F \times K$, where $F \in \{1, 2, 3, 5, 7\}$. That is, the size of the subsequence of mnemonics that the filter can detect ranges from 1 to a maximum of 7 mnemonics. The reason behind applying the convolution operator with filters of different size is because salient and prominent parts in the sequence of instructions can vary in size and location. Consequently, choosing the right kernel size for the convolutional operation might be a difficult choice. Note that the 3-dimensional convolutions are applied for each function individually. Afterwards, max-pooling is applied to keep only the strongest activation of each feature map per function. This is achieved by applying the pooling operator with filters of size $1 \times M \times 1$. As a result, the output is a 2-dimensional vector of size $N \times L$, where L is equal to the number of filters. Additionally, the strongest activations within a sample (not only at the function level) are extracted and gathered in a vector of size L for later usage, hereinafter called G.

Function-level Feature Extraction. This block takes as input the $N \times L$ array outputted by the previous block. Similarly, it performs convolutions on the input with filters of size $1 \times L$, $2 \times L$ and $3 \times L$ which extract features from one, two or three functions at a time. Global max-pooling and global average-pooling are then performed to generate the program's feature vector representation. Both pooling operations are applied with filters of size $N \times 1$. This generates two vectors of size Q, where Q is equal to the sum of the number filters of size $1 \times L$, $2 \times L$ and $3 \times L$, that are concatenated into a single feature vector, hereinafter called V, of size $Q \cdot 2$.

Fully-connected layer. The feature vector V is passed through a fully connected layers with 128 neurons that non-linearly combine the high-level input features into a reduced feature vector.

$$h(V) = \sigma(W_1 V + b_1)$$

where σ is the non-linearity function and W_1 and b_1 are the weights and biases, respectively.

Skip Connection. The result of $h(V)$ is then concatenated with the global n-gram like features extracted at the mnemonics level, G. This allows us to pass that information to the deeper layers. Consequently, the network would assign a given malware to a family considering both simpler n-gram like features and complex hierarchical features.

$$J = h(V) \oplus G$$

where $\oplus$ is the concatenation of vectors, and G is the vector containing the global 1-gram, 2-gram, 3-gram, 5-gram and 7-gram strongest activations at the mnemonics-level.

Fully-connected layer. This layer takes as input vector J and non-linearly combines the features into a low dimensional feature vector P, where $P = h(J) = \sigma(W_2 J + b_2)$, and W_2 and b_2 denote the weights and biases of the fully-connected layer.

Malware Classification. The resulting program feature vector P is the high level representation of the assembly language source code and can be used to classify a malicious program into its corresponding family:

$$probs = s(W_c P + b_c)$$

where probs is a vector of size C ($C = |\, families|$), W_c and b_c are the weights and biases of the layer, and s is the

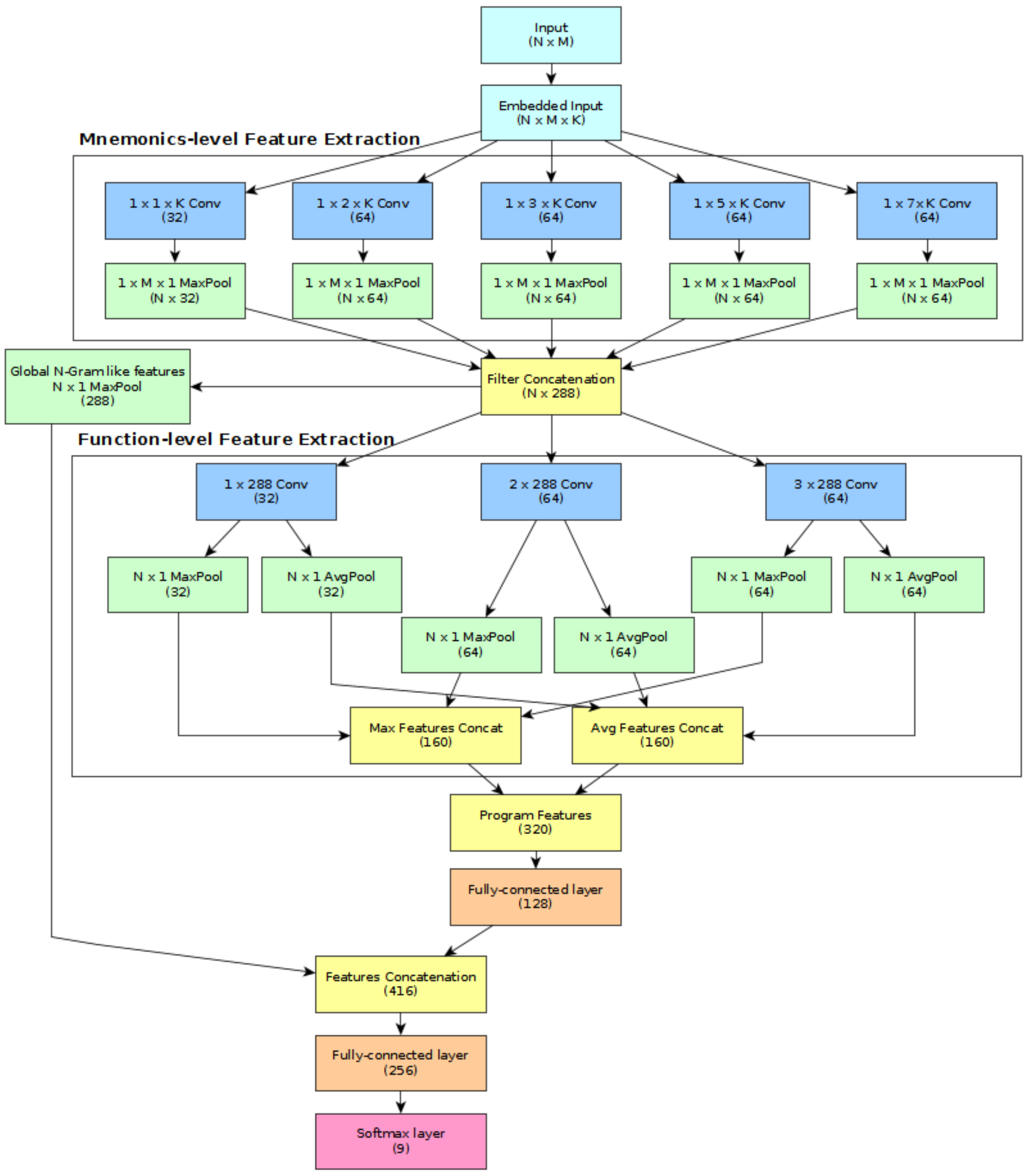

Input (N x M)
Embedded Input (N x M x K)
Mnemonics-level Feature Extraction
1 x 1 x K Conv (32)
1 x 2 x K Conv (64)
1 x 3 x K Conv (64)
1 x 5 x K Conv (64)
1 x 7 x K Conv (64)
1 x M x 1 MaxPool (N x 32)
1 x M x 1 MaxPool (N x 64)
1 x M x 1 MaxPool (N x 64)
1 x M x 1 MaxPool (N x 64)
1 x M x 1 MaxPool (N x 64)
Global N-Gram like features N x 1 MaxPool (288)
Filter Concatenation (N x 288)
Function-level Feature Extraction
1 x 288 Conv (32)
2 x 288 Conv (64)
3 x 288 Conv (64)
N x 1 MaxPool (32)
N x 1 AvgPool (32)
N x 1 MaxPool (64)
N x 1 AvgPool (64)
N x 1 MaxPool (64)
N x 1 AvgPool (64)
Max Features Concat (160)
Avg Features Concat (160)
Program Features (320)
Fully-connected layer (128)
Features Concatenation (416)
Fully-connected layer (256)
Softmax layer (9)

Fig. 3. Structure of the Hierarchical Convolutional Neural Network for Malware Classification. Blue, green and yellow boxes represent the convolution, pooling and concatenation operations, respectively.

softmax function. The purpose of the softmax layer is to output the probability that an executable belongs to one family or another.

The activation function adopted through all the layers is the Exponential Linear Unit (ELU) [18]. Additionally, weights are initialized according to Xavier [19]. Lastly, the dropout rate [20] during training for the fully-connected and convolutional layers was 0.5 and 0.1, respectively.

IV. Evaluation

A. Performance Metrics

The generalization performance of our approach was estimated using 10-fold cross validation and the best model was selected according to the macro F1 score, which is the average of the individual F1 scores obtained for each class:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}$$

where P and R are the precision and recall evaluation metrics. The reason why accuracy is not considered as the only evaluation metric is because there is a large class imbalance in the dataset and thus, we do not want to end up with a model able to correctly predict the value of the majority classes while making mistakes with the critical classes. Consequently, the macro F1 score best meets our requirements.

To evaluate how our approach performed on the test set we used the following multi-class logarithmic loss (logloss):

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(y_{i,j}\log(p_{i,j}) + (1-y_{i,j})\log(1-p_{i,j}))$$

where N is the number of observations, M is the number of class labels, log is the natural logarithm, $y_{i,j}$ is 1 if the observation i belongs to class j and 0 otherwise, and $p_{i,j}$ is the predicted probability that observation i belongs to class j. This metric is used to evaluate the test set because their corresponding labels are not provided. Instead, one should submit a file containing a set of predicted probabilities (one for each class) to Kaggle to evaluate the performance of a model on the test set.

B. Comparison with the state-of-the-art

The generalization performance of our approach has been compared with state-of-the-art methods in the literature on the Microsoft Malware Classification benchmark [3]. These methods can be divided into two groups, depending on the input file type.

1) Hexadecimal-based approaches. This group includes approaches that are based on the hexadecimal representation of malware's binary content. D. Gibert et al. [10] presented a convolutional neural network to classify malware's binary content represented as gray scale images. In addition, in the work of D. Gibert et al. [12] they evaluated the performance of convolutional neural networks and K-nearest neighbor algorithms to classify malware represented as a stream of entropy values. Instead, D. Gibert et al [13] and Quan Le et al. [21] compressed the information described in the byte sequence using denoising autoencoders and scaling methods, respectively. On the contrary, E. Raff et al. [1] and M. Krčál et al. [11] proposed classifying malware using only the raw byte sequences as input. On the one hand, E. Raff et al. [1] presented a shallow convolutional neural architecture composed of a gated convolutional layer followed by a global max pooling layer, a fully connected and a softmax layer. On the other hand, M. Krčál et al. [11] presented a deeper architecture consisting of 9 layers in total, 4 convolutional layers plus a global average pooling layer and 3 fully connected followed by the softmax layer.
2) Assembly-based approaches. This group includes approaches that are based on assembly language source code of the executables. M. McLaughlin [17] and D. Gibert et al [2] proposed detecting malware using a shallow convolutional network. The difference between both methods is that in the work of D. Gibert et al [2] the network consists of only one convolutional layer with filters of various sizes. Furthermore, we implemented a hierarchical attention network based on the work of Z. Yang et al. [22].

Table I presents a comparison between state-of-the-art methods in the literature and our approach. More specifically, it shows the 10-fold cross validation accuracy and macro f1 score achieved with the training set, and the multi-class logarithmic loss with the test set. On the one hand, methods that are fed with the raw byte sequences as input [1], [11] perform worse than those that first compress or encode the information in the byte sequences [12], [13]. Our hypothesis is that due to the limited amount of training data and the higher complexity of the network, those methods are more prone to overfitting. An analysis of these methods with a bigger dataset is necessary to correctly assess their generalization performance. However, there is no other public benchmark of Portable Executable files available for comparison due to copyright laws and restrictions in labeling procedures. To avoid falling into the trap of evaluating our approach using an in-house dataset, for reproducibility purposes we decided to assess its performance with a standard benchmark [3]. On the other hand, assembly-based approaches generally perform considerably better than hexadecimal-based approaches with the exception of the Hierarchical Attention Network (HAN). However, even though the HAN achieved a lower 10-fold cross validation accuracy and macro F1 score, it achieved a lower logarithmic loss with the test set in comparison with the hexadecimal-based approaches. Furthermore, both the *Shallow CNNs* and the *Hierarchical Convolutional Network* achieved a higher classification accuracy and macro F1 score, and a lower logloss than other approaches. This was for two reasons: (1) The length of the sequence of mnemonics is much shorter than the length of the raw byte sequences, thus, given the limited training data, the simpler architectures are less prone to overfitting; (2) Both the *Shallow CNNs* and

TABLE I
STATE OF THE ART COMPARISON OF DEEP LEARNING METHODS ON THE MICROSOFT MALWARE CLASSIFICATION CHALLENGE BENCHMARK. APPROACHES WITH A "*" MARK INDICATE THAT THEY PERFORMED 5-FOLD CROSS VALIDATION INSTEAD OF 10-FOLD CROSS VALIDATION TO ASSESS THE PERFORMANCE OF THEIR METHOD.

	Training 10-fold cross validation		Test
	Accuracy	Macro F1 score	Logarithmic Loss
Hex-based approaches			
CNN IMG [12]	0.975	0.940	0.1844
CNN Entropy [12]	0.9708	0.9314	0.1346
CNN Haar Approximation & Details [12]	0.9828	0.9636	0.1244
Autoencoder + Dilated Residual Network [13]	0.9861	0.9719	0.1063
MalConv [1]	0,9641	0.8902	0.3071
DeepConv [11]	0.9756	0.9071	0.1602
CNN+BiLSTM [21]*	0.9820	0.9605	0.0744
Assembly-based approaches			
Shallow CNN: filters with multiple sizes [2]	0.9917	0.9856	0.0351
Shallow CNN [17]	0.9903	0.9743	0.0515
HAN [22]	0.9742	0.9468	0.0933
Hierarchical Convolutional Network	0.9913	0.9830	0.0419

the *Hierarchical Convolutional Network* perform convolutions on the mnemonic sequences. Applying convolutions rather than recurrent units is computationally more efficient and is more suitable for our classification task, since we are dealing with very long input sequences. Furthermore, malware authors usually employ a wide range of obfuscation techniques to modify the appearance of executables without modifying their behavior. One of the most common obfuscation techniques is subroutine reordering. That is, the order of the subroutines in the original code is changed randomly. Another common technique is code transposition which reorganizes the order of the instructions without changing the behavior of the computer program. Thus, convolutional-based networks are more adequate to deal with the problem at hand , since they are able to detect patterns that might be displaced in space through the convolution and max-pooling operations.

Figure 4 displays the confusion matrix and the normalized confusion matrix achieved by our method. The major contributor to misclassifications is the Obfuscator.ACY family which, according to Microsoft, is malware that can have almost any purpose and has tried to hide its purpose using a combination of obfuscation methods such as encryption, compression, anti-debugging, anti-emulation techniques, etc., in such a way that it could not be detected. In addition, there are some samples in the dataset which have almost no instructions or zero instructions, due to certain issues during the disassembly process, it being very complicated to label them correctly using only the information provided with the assembly language instructions. Consequently, to correctly classify those samples it might be necessary to complement our method with features extracted from the hexadecimal representation of the binary's content or with features not related with the assembly language source code.

V. CONCLUSIONS

In this paper, we propose a Hierarchical Convolutional Network (HCN) to classify malware's assembly language source code in Portable Executable files. The hierarchical structure of the network is best suited to take advantage of the hierarchical structure of Portable Executable files, as it allows n-gram like features to be extracted from both the mnemonics level and the function level. The use of convolutions has proven to be more useful to detect malware, due to the higher dimensional space of the system's input data than recurrent units and attention. In addition, the proposed solution does not rely on costly feature engineering, since it learns the best n-gram like features automatically during training. Experimental results demonstrate that our model performs significantly better than hexadecimal-based approaches in the literature and than the Hierarchical Attention Network, and performs comparably to the shallow CNN with filters of multiple sizes.

A. Future Work

Due to the limitations related to disassembling the binary's content, a future line of research might be to explore multimodal learning for malware classification. Multimodal learning is a model to show the joint representation of different modalities. In the problem of malware detection and classification, one modality might be represented by a different type of features. For instance, API function calls, assembly language instructions, raw bytes sequence, function call graph, etc. This kind of model might be able to discover the relationships between different modalities and build a robust classifier.

REFERENCES

[1] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole EXE," in *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018.*, 2018, pp. 268–276.

[2] D. Gibert, J. Béjar, C. Mateu, J. Planes, D. Solis, and R. Vicens, "Convolutional neural networks for classification of malware assembly code," in *Recent Advances in Artificial Intelligence Research and Development - Proceedings of the 20th International Conference of the Catalan Association for Artificial Intelligence, Deltebre, Terres de l'Ebre, Spain, October 25-27, 2017*, 2017, pp. 221–226. [Online]. Available: https://doi.org/10.3233/978-1-61499-806-8-221

[3] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft Malware Classification Challenge," *ArXiv e-prints*, Feb. 2018.

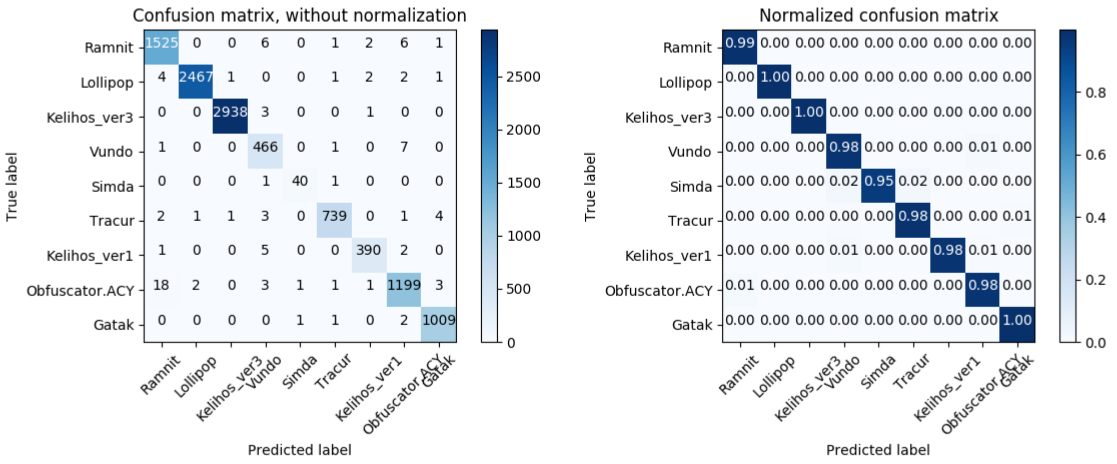


Fig. 4. Confusion matrices with and without normalization of the 10-fold cross classification accuracy achieved by the Hierarchical Convolutional Network on the Microsoft benchmark.

[4] S. Jain and Y. K. Meena, "Byte level n–gram analysis for malware detection," in *Computer Networks and Intelligent Computing*, K. R. Venugopal and L. M. Patnaik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 51–59.

[5] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, pp. 64 – 82, 2013, data Mining for Information Security.

[6] D. Yuxin and Z. Siyi, "Malware detection based on deep learning algorithm," *Neural Computing and Applications*, Jul 2017. [Online]. Available: https://doi.org/10.1007/s00521-017-3077-6

[7] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security Privacy*, vol. 5, no. 2, pp. 40–45, March 2007.

[8] I. Sorokin, "Comparing files using structural entropy," *Journal in Computer Virology*, vol. 7, no. 4, p. 259, Jun 2011.

[9] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware detection based on mining api calls," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 1020–1025.

[10] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Using convolutional neural networks for classification of malware represented as images," *Journal of Computer Virology and Hacking Techniques*, Aug 2018.

[11] M. Krčál, O. Švec, M. Bálek, and O. Jašek, "Deep convolutional malware classifiers can learn from raw executables and labels only," 2018. [Online]. Available: https://openreview.net/forum?id=HkHrmM1PM

[12] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Classification of malware by using structural entropy on convolutional neural networks," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 2018, pp. 7759–7764.

[13] D. Gibert, C. Mateu, and J. Planes, "An end-to-end deep learning architecture for classification of malware's binary content," in *Artificial Neural Networks and Machine Learning – ICANN 2018*, V. Kůrková, Y. Manolopoulos, B. Hammer, L. Iliadis, and I. Maglogiannis, Eds. Cham: Springer International Publishing, 2018, pp. 383–391.

[14] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *AI 2016: Advances in Artificial Intelligence*, B. H. Kang and Q. Bai, Eds. Cham: Springer International Publishing, 2016, pp. 137–149.

[15] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on opcode patterns," *Security Informatics*, vol. 1, no. 1, p. 1, Feb 2012. [Online]. Available: https://doi.org/10.1186/2190-8532-1-1

[16] B. Kang, S. Y. Yerima, K. Mclaughlin, and S. Sezer, "N-opcode analysis for android malware classification and categorization," in *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, June 2016, pp. 1–7.

[17] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn, "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: ACM, 2017, pp. 301–308. [Online]. Available: http://doi.acm.org/10.1145/3029806.3029823

[18] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *CoRR*, vol. abs/1511.07289, 2015.

[19] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[20] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012. [Online]. Available: http://arxiv.org/abs/1207.0580

[21] Q. Le, O. Boydell, B. M. Namee, and M. Scanlon, "Deep learning at the shallow end: Malware classification for non-domain experts," *Digital Investigation*, vol. 26, pp. S118 – S126, 2018.

[22] Z. Yang, D. Yang, C. Dyer, X. He, A. J. Smola, and E. H. Hovy, "Hierarchical attention networks for document classification," in *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, 2016, pp. 1480–1489. [Online]. Available: http://aclweb.org/anthology/N/N16/N16-1174.pdf

4.3.3 Classification of Malware by Using Structural Entropy on Convolutional Neural Networks

To evade detection, malware authors employ a variety of obfuscation techniques to hide malicious code inside executables. The most common are encryption and compression, which are employed in most of the malware samples. In the information security industry, a common practice to detect the presence of encrypted or compressed segments hidden beneath portable executables is entropy analysis. In general, segments of code that have been compressed or encrypted tend to have higher entropy than native code [54].

In information theory, entropy (more specifically, Shannon's entropy) is the expected value of the information contained in each message. Generally speaking, the entropy of a bytes sequence refers to the amount of disorder (uncertainty) or its statistical variation. If occurrences of all values are the same, the entropy will be larger. On the contrary, if certain byte values occur with high probabilities, the entropy value will be smaller. However, the use of simple entropy statistics may not be enough to detect sophisticated malware. Authors sometimes try to conceal encrypted or compressed code in a way that they pass through high entropy filters. For instance, they may add additional padding to reduce the mean file entropy. Thus, researchers [69] started analyzing what is defined as the structural entropy of a file. In other words, each executable file is represented as a stream of entropy values, where each value describes the amount of entropy over a small chunk of code in a specific location of the file. Figure 4.7 displays the structural entropy of various malware executables belonging to two different families. It can be observed that the entropy streams extracted from malware samples belonging to the same family appear to be similar while distinct from those belonging to different families.

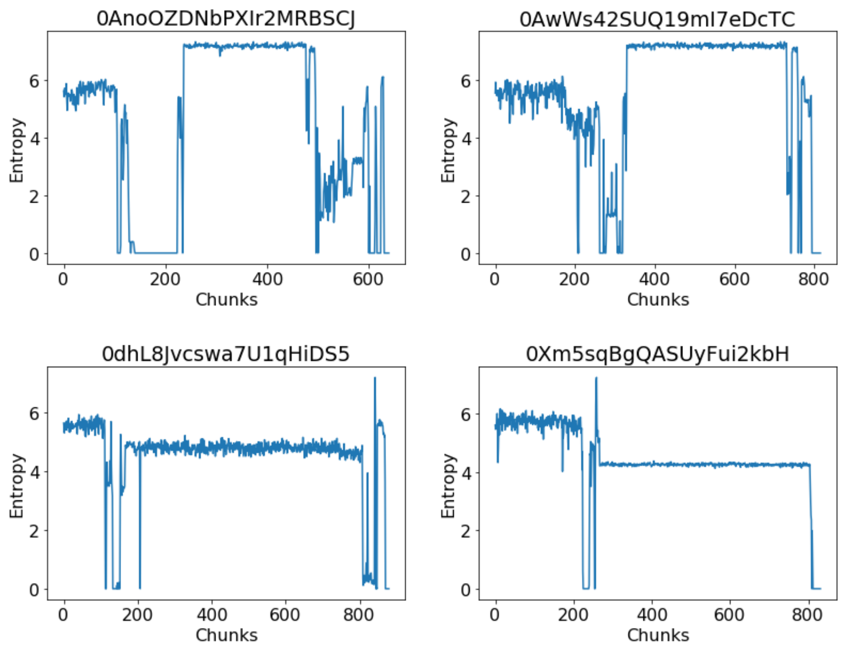


Figure 4.7: Entropy time series from malicious software. Samples from the first row belong to the Ramnit family, whereas samples from the second row belong to the Gatak family. Note the variation of the stream of entropy values between families.

By representing executable files as a stream of entropy values, the task of malware classification can be described as a time series classification problem. For time series classification tasks, the most successful approaches in the literature are: (1) Time domain distance based classifiers and (2) Shapelet-based classifiers. On the one hand, the 1-nearest neighbor using the Dynamic Time Warping (DTW) as distance metric achieves state-of-the-art classification results on the time series classification task [18]. However, the nearest neighbor classification algorithm is limited by its space and classification time complexity and subsequently, it is not suitable in those scenarios where a low prediction time is a requirement, as the prediction time of the nearest neighbor grows linearly as the dataset grows larger. On the other hand, shapelet-based classifiers rely on the extraction of discriminant subsequences of the time series [74] to distinguish the time series by their local variations instead of their global structure. However, the computational complexity for the brute force algorithm is polynomial, for best cases, $O(n^2m^3)$, where n is the number of time series in the dataset and m is the average length of each time series. Alternatively, these shapelets can be learned automatically by formulating the shapelet learning task as an optimization of a classification objective function [34]. The main idea behind this is to directly learn optimal shapelets without needing to explore all possible candidates. This is done by starting with rough initial guesses for the shapelets and by iteratively learning/optimizing the shapelets by minimizing a classification loss function. In our case, the loss function is the multi-class logarithmic loss:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(y_{i,j}log(p_{i,j}) + (1 - y_{i,j})log(1 - p_{i,j}))$$

where N is the number of observations, M is the number of class labels, log is the natural logarithm, $y_{i,j}$ is 1 if the observation i is in class j and 0 otherwise, and $p_{i,j}$ is the predicted probability that observation i is in class j. Furthermore, instead of the learning algorithm presented by Grabocka et al. [34], Gibert et al. [31] proposed to learn the shapelets by stacking convolutional layers, which allows a hierarchical decomposition of the input time series. To do this, the wavelet transform is applied to the entropy time series in order to compress the signal and reduce the noise. See Figure 4.8.

For a more detailed description of the convolutional network architecture and the experimental setup the reader is referred to the original research article [31].

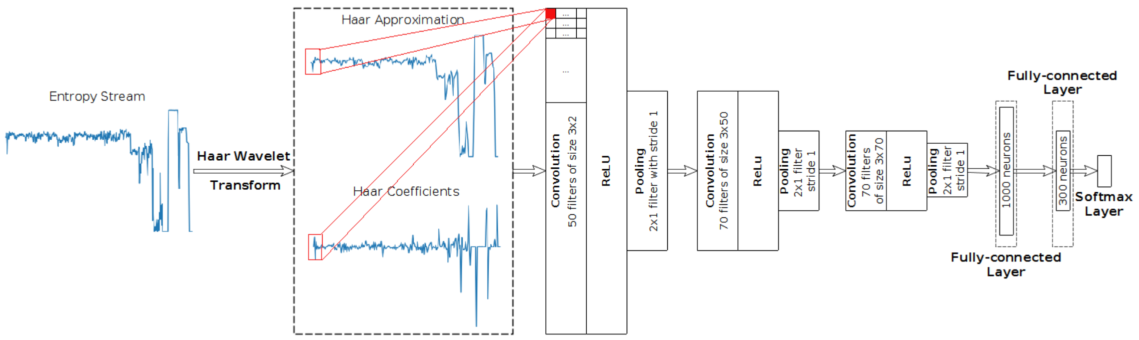

Entropy Stream
Haar Wavelet Transform
Haar Approximation
Haar Coefficients
Convolution 50 filters of size 3x2
ReLU
Pooling 2x1 filter with stride 1
Convolution 70 filters of size 3x50
ReLu
Pooling 2x1 filter stride 1
Convolution 70 filters of size 3x70
ReLu
Pooling 2x1 filter stride 1
1000 neurons
Fully-connected Layer
300 neurons
Fully-connected Layer
Softmax Layer

Figure 4.8: Convolutional network architecture for malware classification based on its structural entropy representation.

Classification of Malware by Using Structural Entropy on Convolutional Neural Networks

Daniel Gibert
Blueliv, Leap in Value
Barcelona, Spain

Carles Mateu
University of Lleida
Lleida, Spain

Jordi Planes
University of Lleida
Lleida, Spain

Ramon Vicens
Blueliv, Leap in Value
Barcelona, Spain

Abstract

The number of malicious programs has grown both in number and in sophistication. Analyzing the malicious intent of vast amounts of data requires huge resources and thus, effective categorization of malware is required. In this paper, the content of a malicious program is represented as an entropy stream, where each value describes the amount of entropy of a small chunk of code in a specific location of the file. Wavelet transforms are then applied to this entropy signal to describe the variation in the entropic energy. Motivated by the visual similarity between streams of entropy of malicious software belonging to the same family, we propose a file agnostic deep learning approach for categorization of malware. Our method exploits the fact that most variants are generated by using common obfuscation techniques and that compression and encryption algorithms retain some properties present in the original code. This allows us to find discriminative patterns that almost all variants in a family share. Our method has been evaluated using the data provided by Microsoft for the BigData Innovators Gathering Anti-Malware Prediction Challenge, and achieved promising results in comparison with the State of the Art.

Introduction

To evade detection, malware authors employ a variety of obfuscation techniques to hide malicious code inside executables. The most common are encryption and compression which are employed in most of the malware samples. In the information security industry, a common practice to detect the presence of encrypted or compressed segments hidden beneath portable executables is entropy analysis. In general, segments of code that have been compressed or encrypted tend to have higher entropy than native code (Lyda and Hamrock 2007).

In information theory, entropy (more specifically, Shannon's entropy) is the expected value of the information contained in each message. Generally speaking, the entropy of a bytes sequence refers to the amount of disorder(uncertainty) or its statistical variation. If occurrences of all values are the same, the entropy will be largest. On the contrary, if certain byte values occur with high probabilities, the entropy value will be smaller.

However, the use of simple entropy statistics may not be enough to detect sophisticated malware. Authors sometimes try to conceal encrypted or compressed code in a way that they pass through high entropy filters. For instance, they may add additional padding to reduce the mean file entropy. Anyway, native, encrypted or compressed segments and padding tend to differ markedly having distinct entropy levels. Thus, researchers (Sorokin 2011) started analyzing what is defined as the structural entropy of a file. In other words, each executable file is represented as a stream of entropy values, where each value describes the amount of entropy over a small chunk of code in a specific location of the file. Figure 1 displays the structural entropy of various malware executables belonging to two different families. It can be observed that the entropy streams extracted from malware samples belonging to the same family appear to be similar while distinct from those belonging to different families.

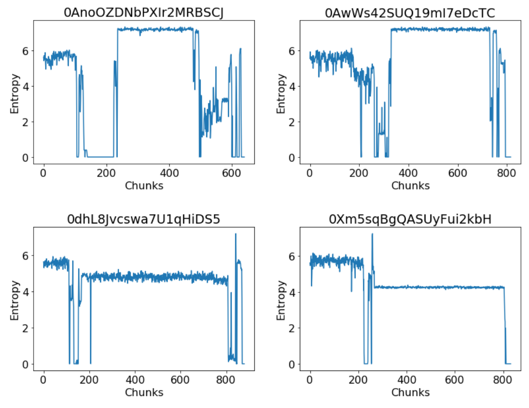


Figure 1: Entropy time series from malicious software. Samples from the first row belong to the Ramnit family, whereas samples from the second row belong to the Gatak family. Note the variation of the stream of entropy values between families.

By representing executable files as a stream of entropy values, the task of malware classification can be described as a time series classification problem (Fu 2011; Xing, Pei, and Keogh 2010; Bagnall et al. 2017).

The approaches most studied have been (1) Time domain distance based classifiers and (2) Shapelet based classifiers. On the one hand, Dynamic Time Warping (DTW) has been widely used as the preferred method to measure the similarity between two temporal sequences that may vary in length. (Ding et al. 2008), evaluated 8 different distance metrics on 38 time series datasets and found that DTW jointly with 1-Nearest Neighbor (NN) outperformed most of them. However, the NN is limited by its space complexity and its classification time complexity. On the other hand, time series shapelets were first used for classification in (Ye and Keogh 2009). Shapelets are discriminant subsequences of time series. The idea is that different classes of time series can be distinguished by their local variations instead of their global structure. Nevertheless, the computational complexity for the brute force search algorithm is polynomial, for best cases, $O(n^2m^3)$, where n is the number of time series in the dataset and m is the average length of each time series. (Grabocka et al. 2014) introduced a more efficient alternative to learn shapelets. In their method, instead of searching among possible candidates from time series segments, they proposed a method to directly learn optimal shapelets without exploring all possible candidates. Their approach, starts by guessing a set of initial shapelets. Then, it iteratively learns the shapelets by minimizing an error function.

Inspired by the recent advances in the deep learning field and the work of (Grabocka et al. 2014), in this paper we propose a file agnostic end-to-end deep nonlinear feature learning and classification based method for categorization of malicious software based on its structural entropy. Our approach has been evaluated using the dataset provided by Microsoft for the Big Data Innovators Gathering (BIG 2015) Anti-Malware Prediction Challenge.

The rest of the paper is organized as follows. Firstly, we introduce our deep learning approach for malware classification. Then, the results of the performance evaluation for our method are presented. And lastly, the paper concludes with our remarks.

Classification of Software's Structural Entropy via Deep Learning

As can be observed in Figure 1, there exists empirical evidence that entropy time series from a given family are visually similar and distinct from those belonging to a different family. This is perhaps the result of reusing the code to create new malware variants. Consequently, this visual similarity motivated us to apply convolutional neural networks for time series classification to automatically learn good feature representations from both time and frequency domains jointly. To do this, a given executable is transformed into a recognizable input by applying the following two-step process:

1. Structural entropy calculation. The entropy of an executable file is computed by splitting its hexadecimal representation (00h–FFh) into non-overlapping chunks of fixed size. In the literature, a common value is 256 bytes. For each chunk of code, the entropy is then computed using Shannon's formula defined as:

$$H(X) = -\sum_{i=1}^{n} p(i) \cdot log_b p(i)$$

   where $H(X)$ is the measured entropy value of a discrete random variable X with values $x_1, \dots, x_j$, j is the number of values in X, $p(i)$ refers to the probability of appearances of the byte value i in X and n is equal to 255, i.e. byte code values are in the range of [0, 255].

2. Discrete Wavelet Transform. The single-level discrete wavelet transform is applied to the entropy time series in order to compress the signal and reduce the noise. The original vector of length N is transformed into two vectors of length $N/2$, named the approximation coefficients and the detail coefficients. In this work, the Haar wavelet transform (Haar 1910) has been used, instead of any other transforms such as Daubechies or Morlet, due to its efficiency in computation.

Network Architecture

The overall architecture of the network is illustrated in Figure 2. The input is a multivariate time series M, defined as $M = \{m_1, m_2\}$, where each element m_i is a univariate time series. A univariate time series is a sequence of data points measured at successive points in time. It is denoted as $T = \{t_1, t_2, \dots, t_n\}$, where n is the length of T. At any time stamp t, $m_t = \{m_{1,t}, m_{2,t}\}$ where $m_{1,t}$ and $m_{2,t}$ are the values of the Haar approximation and Haar coefficient values at time stamp t.

The core of the convolutional neural network consists of three convolutional layers plus two fully-connected layers. The convolutional layers perform feature learning on both univariate series jointly. Then a normal feed-forward network is concatenated at the end of feature learning to perform classification. Specifically, the input is fed into a 3-stage feature extractor which learns hierarchical features through convolution, activation and pooling layers.

Convolution is an operation that takes an input signal and a feature map and produces one output signal. A convolution operation involves a filter $w_k \in \mathbb{R}^{ij}$ where $i \leq w$ and $j \leq h$ and i and j are the width and the height of the an input 2D signal. The output of convolving the k-th kernel of the convolutional layer l, $w_{l,k}$ over a 2D signal is defined as $c = w_{l,k} \times x + b_{l,k}$, where $b_{l,k}$ is the bias of the k-th kernel in layer l. The kernel slides over each value of the input signal, multiplies the corresponding entries of the input signal and the kernel and adds them up. The convolutional layers are composed of 50, 70 and 70 feature maps with 3 by 2, 3 by 50 and 3 by 70 receptive fields for the first, second and third convolutional layers, respectively.

Activation function introduces non-linearity into the network. It takes a single value x and performs a mathematical operation on it. In particular, we adopt the ReLU function $\max(0, x)$ in all activation layers.

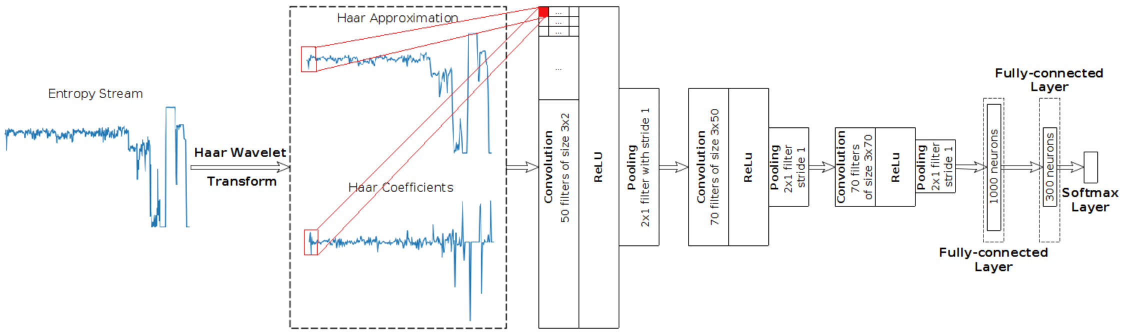

Entropy Stream
Haar Wavelet Transform
Haar Approximation
Haar Coefficients
Convolution 50 filters of size 3x2
ReLU
Pooling 2x1 filter with stride 1
Convolution 70 filters of size 3x50
ReLu
Pooling 2x1 filter stride 1
Convolution 70 filters of size 3x70
ReLu
Pooling 2x1 filter stride 1
1000 neurons
Fully-connected Layer
300 neurons
Fully-connected Layer
Softmax Layer

Figure 2: Convolutional network architecture for malware classification. It is composed by 3 convolutional layers followed by 2 fully-connected layers. The input of the network are two univariate time series, the average and the details vector generated by transforming an entropy stream with Haar wavelets. The output of the network is the predicted class of the malware sample.

Pooling is a function that reduces the spatial size of an input signal. It helps to reduce the amount of parameters and computation in the network as well as to control overfitting. We applied max-pooling with filters of size 2×1 with stride 1, which reduces the input signal by half.

Convolutional layers can be seen as detection filters for the presence of specific features or patterns present in the data. The first layers detect low-level features whereas the last ones detect increasingly complex features. At the end of the extractor, the feature maps are flattened and combined as the input of the subsequent feed-forward layers plus a softmax layer for classification. Particularly, the number of units in the feed-forward layers is equal to 1000 and 300 for the first and the second layer, respectively. To prevent overfitting, dropout (Srivastava et al. 2014) was used and, to improve the stability of our model, an ensemble algorithm was used, named bootstrap aggregating (Breiman 1996).

Resilience to Obfuscation Techniques

By nature, the features learned by the convolutional neural networks are invariant to translation. That is, CNNs are able to detect patterns which may be displaced in space through the convolution and max-pooling operations. The convolution operation provides equivariance to translation. In our domain this means that signal patterns may be recognized at any temporal space. Additionally, the max-pooling operation returns the largest value in its receptive field. Thus, the location of this value, if it is still within the receptive field, do not alters the output of the pooling operation. Thus, both operations together provide invariance to translation. This property is really helpful against detecting the changes produced by the following obfuscation techniques:

Dead-code insertion. This technique adds ineffective instructions, such as the NOP instruction, to the program to change its appearance while maintaining the same functionality. By adding NOP instructions, the average entropy of the executable will decrease, but the entropy of the adjacent chunks containing the actual code will differ greatly from the chunks containing NOP instructions as it can be observed in Figure 3.

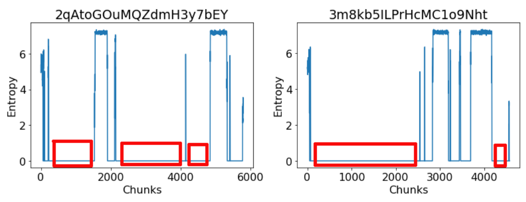


Figure 3: Two samples belonging to the Simda family whose code has been modified by the dead-code insertion technique (chunks highlighted in red).

Code transposition. This technique reorders the sequence of the instructions without changing the behavior of the program. For instance, the instructions

```
1: ADD R1 R2                          1: ADD R3 R4
2: ADD R3 R4     can be replaced by   2: ADD R1 R2
```

If the sequence of instructions is located inside the same chunk of code as it was previously to the reordering, the entropy of the chunk will still be the same.

Subroutine reordering. By applying this technique, the order of the subroutines in the original code is changed randomly. Cf. Figure 4. Being invariant to translation means that the location of the subroutines will not affect the outcome of the classifier, because the network is able to find the patterns independent of their location.

Encryption and packing are the most common methods employed to hide malicious code into executables. These methods transform a series of original bytes into a series of random-looking data bytes. In the information security industry, to detect the presence of encrypted or com-

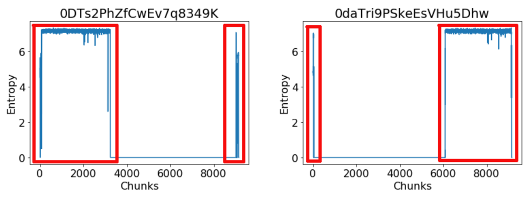


Figure 4: Two samples belonging to the Kelihos_ver3 family. It can be observed that the actual code of the program (highlighted in red) has been reallocated from the start to the end of the file.

pressed segments hidden beneath the executable, code entropy analysis is commonly performed. Typically, files with high entropy are relatively likely to have encrypted or compressed sections inside them (Lyda and Hamrock 2007). Thus, by representing an executable as a stream of entropy values, the presence of encrypted or compressed segments hidden within portable executable files can be detected. Figure 5 shows two samples belonging to the Obfuscator.ACY family. It can be observed that the entropy of different segments varies along the files. Therefore, the local patterns learned by the CNN should be able to detect these changes in the entropy values between encrypted or compressed chunks and the chunks containing the rest of the code.

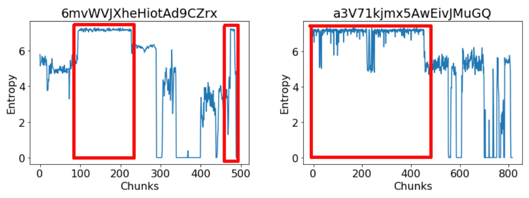


Figure 5: Two samples belonging to the Obfuscator.ACY family. The red box highlights the possible encrypted or compressed segments within the files.

Evaluation

The data used to evaluate our deep learning approach were provided by Microsoft for the BigData Innovators Gathering (BIG 2015) Anti-Malware Prediction Challenge. The dataset is composed of 21741 samples, 10868 for training and 10873 for testing, grouped into 9 different malware families (Cf. Table 1).

Experimental Setup

To estimate the generalization performance of our approach we used K-fold cross validation, where $K = 10$. Additionally, the best model was selected according to the F1 score. That is because classification accuracy alone can be misleading. Sometimes, it may be desirable to select a model with a lower accuracy but with greater predictive power (a.k.a. accuracy paradox). This is true in a problem like ours where there is a large class imbalance, where a model can predict the value of the majority class for all predictions and achieve high classification accuracy while misclassifying samples from the minority or critical classes. In particular, since the task we are trying to solve is a multi-class classification problem we used an adaptation of the score called macro-averaged F1 score, defined as the average of the individual F1 scores obtained for each class. Macroaveraging gives equal weight to each class. Thus, large classes will not dominate small classes.

Table 1: BIG 2015 dataset statistics.

Family	Class ID	#samples	Average size (bytes)	Average length (256 bytes)
Ramnit	1	1541	1482169.56	1597.17
Lollipop	2	2478	5829531.16	6281.75
Kelihos_ver3	3	2942	8982629.66	9679.56
Vundo	4	475	1120945.27	1207.90
Simda	5	42	4552326.09	4905.52
Tracur	6	751	1801152.85	1940.90
Kelihos_ver1	7	398	5051900.48	5443.85
Obfuscator.ACY	8	1228	827118.28	891.29
Gatak	9	1013	2555072.57	2753.31

The experimentation has been divided into three phases. In the first phase, it has been studied how the chunk size influences the output of the network. In the second phase, it has been analyzed how the Haar approximation of the initial entropic signal impacts our classifier. In the third phase we compared our best model with state of the art methods in the literature.

Chunk Size Comparison The size of the malicious programs varies greatly between families. Thus, their corresponding time series differ in length from one family to another, independently of the chunk size, cf. Table 1.

To study which chunk size provides a better trade-off between accuracy and performance, we evaluated three network models, by using as training data the time series obtained after splitting an executable file into chunks of size 256, 1024 and 4096 bytes.

The network architecture is the same as the one described in Figure 2, with the exception of the input layer. In this case, networks are fed with univariate time series containing the stream of entropy values representing an executable file. The percentage of correctly predicted labels over all predictions (accuracy) is 0.9626, 0.9720 and 0.9708 for 256, 1024 and 4096 bytes, respectively. On the contrary, the highest F1 score was achieved by the model trained on the time series obtained after splitting the malicious programs into chunks of 4096 bytes, which is 0.9314. In Table 2 and Table 3, it can be observed that both models failed to predict most of the samples belonging to the Simda family. Additionally, even though the overall accuracy is higher, the number of misclassified samples (54.76%) belonging to the Ramnit family has greatly punished the model trained on the time series obtained after splitting the files in chunks of size 1024.

Haar Wavelet Transform. In the second phase, Haar Wavelet Transform was used to decompose the initial sig-

Table 2: 10-fold cross validation confusion matrix obtained by training the CNN with the time series obtained by splitting a program into chunks of 1024 bytes.

Family	1	2	3	4	5	6	7	8	9
1	1499	8	1	3	0	8	2	15	5
2	14	2447	0	1	0	5	0	2	9
3	0	0	2940	0	0	2	0	0	0
4	8	3	0	455	0	4	0	4	1
5	19	2	0	3	14	0	1	2	1
6	16	5	0	4	0	715	4	6	1
7	5	2	0	0	0	3	388	0	0
8	59	9	4	9	1	23	2	1117	4
9	6	6	0	3	0	6	0	3	989
Accuracy	10564 / 10868 = 0.9720								
F1 score	0.9127								

Table 3: 10-fold cross validation confusion matrix obtained by training the CNN with the time series obtained by splitting a program into chunks of 4096 bytes.

Family	1	2	3	4	5	6	7	8	9
1	1490	9	1	9	1	15	1	10	5
2	12	2445	0	2	0	5	1	2	11
3	0	0	2940	0	0	2	0	0	0
4	8	3	0	455	0	4	0	5	0
5	12	0	1	4	23	1	0	1	0
6	15	9	0	13	0	702	4	4	4
7	2	0	0	0	0	1	395	0	0
8	57	3	3	8	1	40	1	1114	1
9	5	6	0	5	0	9	0	1	987
Accuracy	10551/ 10868 = 0,9708								
F1 score	0.9314								

nals into two sequences describing an approximation of the original signal, plus a set of details (coefficients) representing the localized changes. Then, two models were trained. On the one hand, we trained a model using as input the resulting signal approximations (hereinafter, Haar approximation model). On the other hand, the second model was fed with the approximation of the original signal plus the details sequence (hereinafter, multiresolution model). The F1 score increased slightly from 0.9314 to 0.9621 and 0.9636 for the Haar approximation and multiresolution models, respectively. Observe in Table 4 that the number of samples misclassified belonging to the Simda family has been reduced by more than half (from 19 to 8). In addition, even that the number of incorrectly classified samples belonging to the Obfuscator.ACY family has been reduced from 114 to 70, it is still a major source of error. That's because the Obfuscator.ACY family comprises malware that has been obfuscated by using compression and encryption techniques, among others. In consequence, the malware that lies underneath this obfuscation can have any purpose and in some cases, its structural entropy is very similar to those of samples from the rest of families.

State-of-the-Art Comparison. Many algorithms have been developed for the task of time series classification. From among them, the nearest neighbor (particularly 1-NN) in combination with the Dynamic Time Warping (DTW) similarity metric achieves the state of the art performance. If applied to the training data provided by Microsoft, the resulting 10-fold cross validation accuracy and F1 score

Table 4: 10-fold cross validation confusion matrix obtained by training the CNN with both the approximation and the coefficient signals of the entropy time series after applying the Haar Wavelet Transform.

Family	1	2	3	4	5	6	7	8	9
1	1519	5	1	3	0	4	2	7	0
2	6	2457	0	2	0	1	0	5	7
3	0	0	2941	0	0	1	0	0	0
4	3	1	0	463	0	6	0	2	0
5	1	0	0	3	34	0	1	3	0
6	5	5	0	15	0	720	0	4	2
7	0	0	0	0	0	2	395	1	0
8	29	4	2	8	3	21	1	1158	2
9	2	6	0	4	0	4	0	3	994
Accuracy	10681 / 10868 = 0.9828								
F1 score	0.9636								

achieved by the 1-NN algorithm are higher than ours. On the contrary, if both approaches are evaluated on the test set, our model substantially outperforms the 1-NN algorithm. Table 5 presents an overview of the results obtained by our deep learning approach and the nearest neighbor algorithm in both the training and the test data. The superior predictive power of convolutional networks (CNN) can be observed with respect to the nearest neighbor algorithm. In particular, the CNN reduced the logarithmic loss to 0.124431 while the logloss obtained by the nearest neighbor is equal to 0.367724. Furthermore, if bagging is employed, by averaging the predictions of the 10 models the test logarithmic loss is reduced to 0.075081. Moreover, a great advantage of the CNN over the nearest neighbor is that its prediction time always remains constant (approximately 0.02 seconds per sample). On the contrary, the prediction time of the nearest neighbor grows linearly as the data set grows larger. That is because to predict the label of an unknown sample, it has to be compared with all individuals in the dataset.

Table 5: Comparison of the CNN with the nearest neighbor algorithm.

Method	10-Fold accuracy	F1 score	Test logloss
DTW + K-NN	0.9894	0.9813	0.367724
Haar Transform + DTW + K-NN	0.9870	0.9710	0.458191
CNN Entropy	0.9708	0.9314	0.134624
CNN Multiresolution	0.9828	0.9636	0.124431
Bagging CNN Multiresolution	-	-	0.075081

Transfer Learning. Once trained, the model could be used to generate domain-specific features based on the structural entropy of a malicious program. Instead of classifying executable files into families, we could extract the features learned by the network in the last feed-forward layer. Then, these features could be integrated into a bigger classifier based on different subsets of features. Next, we prove the suitability of this approach by transferring the features learned into an XGBoost classifier and comparing with the results obtained by (Ahmadi et al. 2015). In their work, they extracted a wide range of hand-crafted features from the malicious executables. The subset of features that achieved the best results individually were:

Entropy (ENT). Statistical measures from the structural entropy of malicious programs such as quantiles, percentiles, mean, etc.

Opcodes (OPC). Use of a subset of 93 operation codes based on their commonness or on their frequent use in malicious applications.

Application Programming Interfaces (APIs). Frequency of use of 794 APIs.

Section (SEC). Characteristics from sections such as the total number of lines in each section, the proportion of each section in comparison to the whole file, etc.

Registers (REG). Frequency of use of the registers.

Miscellaneous (MISC). Frequency of 95 manually chosen words from the disassembled code.

Table 6 shows the results obtained in the training data after performing 5-fold cross validation. It can be observed that the model trained on the entropy-based features extracted by the CNN achieved better accuracy and lower logloss than most hand-crafted features, and in particular, the opposite manually extracted entropy-based features.

Table 6: List of feature categories and their evaluation with XGBoost

		5-Fold Cross Validation	
Feature Category	#Features	Accuracy	Logloss
ENT	203	0.9862	0.0505
OPC	93	0.9907	0.0405
API	796	0.9843	0.0610
SEC	25	0.9899	0.0420
REG	26	0.9833	0.0695
MISC	95	0.9917	0.0306
CNN Multiresolution	300	0.9896	0.0369

Conclusions and Future Work

In this work, we presented a file agnostic deep learning system for categorizing malware. As far as we know, it is the first approach that applies deep learning to find discriminant patterns from executable files represented as streams of entropy values. The proposed solution has a number of advantages that help to detect malicious programs efficiently in an enterprise environment. First, it is file agnostic and is based solely on the structural entropy of a file. Second, the nature of the features learned by convolutional neural networks demonstrated robustness against the most common obfuscation techniques. Third, neural networks scale well with the data. In general, the more data provided the better the quality of the model. Fourth, once the entropy values are computed, the prediction time is minimal. The approach has been compared with state-of-the-art methods in the literature for time series classification and demonstrated the superior predictive power of our deep learning approach.

A future direction of research is to study how different mother wavelets affect the model. Applying any other mother wavelet, such as the Daubechies or the Morlet, might lead to higher accuracy. Additionally, the hierarchical entropy-based features learned by the convolutional neural networks could be useful as a subset of features for machine learning models which attempt to identify malware based on distinct types of file features.

Acknowledgements

We would like to thank the Blueliv Labs team, specially Daniel Solís, and Àngel Puigventós for their support and the feedback provided during the development of this work. This work has been partially funded by the Spanish MICINN Projects TIN2014-53234-C2-2-R, TIN2015-71799-C2-2-P and ENE2015-64117-C5-1-R, and by AGAUR DI-2016-091. This research article has received a grant for its linguistic revision from the Language Institute of the University of Lleida (2017 call).

References

Ahmadi, M.; Giacinto, G.; Ulyanov, D.; Semenov, S.; and Trofimov, M. 2015. Novel feature extraction, selection and fusion for effective malware family classification. *CoRR* abs/1511.04317.

Bagnall, A.; Lines, J.; Bostrom, A.; Large, J.; and Keogh, E. 2017. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery* 31(3):606–660.

Breiman, L. 1996. Bagging predictors. *Machine Learning* 24(2):123–140.

Ding, H.; Trajcevski, G.; Scheuermann, P.; Wang, X.; and Keogh, E. 2008. Querying and mining of time series data: Experimental comparison of representations and distance measures. *Proc. VLDB Endow.* 1(2):1542–1552.

Fu, T.-C. 2011. A review on time series data mining. *Engineering Applications of Artificial Intelligence* 24(1):164 – 181.

Grabocka, J.; Schilling, N.; Wistuba, M.; and Schmidt-Thieme, L. 2014. Learning time-series shapelets. In *KDD'14*, 392–401. ACM.

Haar, A. 1910. Zur theorie der orthogonalen funktionensysteme. *Mathematische Annalen* 69(3):331–371.

Lyda, R., and Hamrock, J. 2007. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy* 5(2):40–45.

Sorokin, I. 2011. Comparing files using structural entropy. *Journal in Computer Virology* 7(4):259.

Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15:1929–1958.

Xing, Z.; Pei, J.; and Keogh, E. 2010. A brief survey on sequence classification. *SIGKDD Expl. New.* 12(1):40–48.

Ye, L., and Keogh, E. 2009. Time series shapelets: A new primitive for data mining. In *KDD'09*, 947–956. ACM.

4.3.4 An End-to-End Deep Learning Architecture for Classification of Malware's Binary Content

This research article presents a file agnostic system for malware classification from raw byte sequences. This is accomplished by using denoising autoencoders to learn an encoded representation of malware's binary content that captures the main factors of variation in the bytes sequences. Afterwards, decisions about the input are made by a dilated residual network classifier that, given the encoded representation of the malware's binary content, outputs the family to which it belongs.

This system can be summarized in two phases:

Phase 1: Chunk Encoding. A given malware binary is divided into contiguous, non-overlapping chunks of fixed size. Afterwards, a denoising autoencoder takes as input every chunk of byte values and projects it into a hidden representation of only one value that captures the main factors of variation in the data. The resulting output is a time series $m = \{m^1, m^2, ..., m^n\}$, where m^i corresponds to the encoding of the i-th chunk and n is the number of chunks into which a binary executable has been divided. The idea is similar to the one presented in Gibert et al. [31] but instead of calculating the entropy of a given chunk, each chunk is encoded as a single value using denoising autoencoders. Thus, the encoding function is learned based on the data provided to the autoencoders. As observed in Figure 4.9, the encodings of samples belonging to the same family are similar while distinct to the encodings of samples belonging to a different family. This visual similarity might be the result of reusing code to create new variants and the result of common obfuscation techniques.

Phase 2: Feature Extraction and Classification. The resulting time series is then fed into a dilated residual network which learns descriptive patterns from the encoding of a bytes sequence and classifies a given malware binary into its corresponding family.

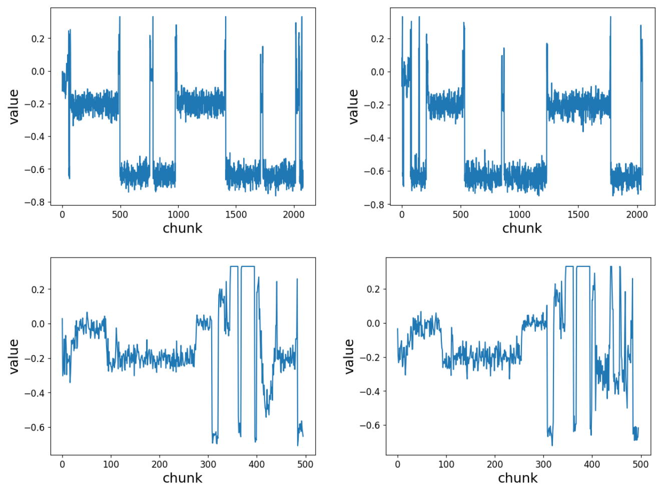


Figure 4.9: Bytes encoding representation. Figures from the first and second row belong to the Simda and Obfuscator.ACY families, respectively.

An End-to-End Deep Learning Architecture for Classification of Malware's Binary Content

Daniel Gibert( ), Carles Mateu, and Jordi Planes

University of Lleida, Jaume II, 69, Lleida, Spain
{daniel.gibert, carlesm, jplanes}@diei.udl.cat

Abstract. In traditional machine learning techniques for malware detection and classification, significant efforts are expended on manually designing features based on expertise and domain-specific knowledge. These solutions perform feature engineering in order to extract features that provide an abstract view of the software program. Thus, the usefulness of the classifier is roughly dependent on the ability of the domain experts to extract a set of descriptive features. Instead, we introduce a file agnostic end-to-end deep learning approach for malware classification from raw byte sequences without extracting hand-crafted features. It consists of two key components: (1) a denoising autoencoder that learns a hidden representation of the malware's binary content; and (2) a dilated residual network as classifier. The experiments show an impressive performance, achieving almost 99% of accuracy classifying malware into families.

Keywords: Malware classification · Deep learning
Denoising autoencoders · Dilated residual networks

1 Introduction

During the last decade, there has been a lot of research and deployment of machine learning techniques to address the problem of malware detection and classification. Machine learning is an attractive signaturless approach to malware detection because of its ability to recognize never-before-seen malware by summarizing complex relationships among the input features and making decisions about it. In traditional machine learning approaches, efforts are spent on manually designing features based on expertise and domain-specific knowledge. These solutions perform feature engineering to extract features that provide an abstract view of malware that a classifier, e.g. neural network, decision tree, support vector machine, etc., use to make a decision. The most effective approaches in the literature are based on N-Gram analysis and entropy analysis. On the one hand, byte N-grams [7] and opcode N-grams [11] are continuous sequences of N items from a given sequence of bytes or opcodes, respectively. The main

V. Kůrková et al. (Eds.): ICANN 2018, LNCS 11141, pp. 383–391, 2018.
https://doi.org/10.1007/978-3-030-01424-7_38

drawback of N-gram based methods is that they are dependent on N and the number of possible combinations increases exponentially with N. To solve this limitation, Gibert et al. [3] proposed a convolutional neural network to automatically learn N-gram like patterns from raw sequences of opcodes, removing the need to exhausivelly enumerate a large number of N-grams. On the other hand, entropy analysis [8] has been used effectively to detect encrypted and compressed executables as they tend to have higher entropy. This characteristic has been exploited by Gibert [4] to group malware into families based on their structural entropy. However, these solutions depend almost entirely on the ability and knowledge of domain experts to extract a set of descriptive and discriminant features into which represent malware.

Instead, the approach presented in this paper neither relies on feature engineering nor on experts' knowledge of the domain. The main contribution of our work is the development of a file agnostic end-to-end deep learning system for malware classification from raw byte sequences. This is accomplished by using denoising autoencoders to learn an encoded representation of the malware's binary content that captures the main factors of variation in the bytes sequences. Afterwards, decisions about the input are made by a dilated residual network classifier that given the encoded representation of the malware's binary content it outputs the family it belongs. The suitability of our approach has been evaluated on a public benchmark provided by Microsoft for the Big-Data Innovators Gathering (BIG 2015) Anti-Malware Prediction Challenge [10]. Experiments demonstrate the greater predictive generalization performance of our approach with respect to the binary-based methods in the literature.

The rest of the paper is organized as follows. Section 2 presents our approach for malware classification. Section 3 describes the experiments and compares our approach with state-of-the-art methods in the literature. Lastly, Sect. 4 contains the concluding remarks and our future line of research.

2 Deep Learning for Malware Classification

In the present paper we describe a file agnostic deep learning system to successfully process and classify malware from raw byte inputs. The system can be summarized in two phases:

Step 1 Chunk Encoding. A given malware binary is divided into contiguous, non-overlapping chunks of fixed size. Afterwards, a denoising autoencoder takes as input every chunk of bytes values and projects it into a hidden representation of only on value that captures the main factors of variation in the data. The resulting output is a time series $m = \{m_1, m_2, ..., m_n\}$, where m_i corresponds to the encoding of the i-th chunk and n is the number of chunks into which a binary executable has been divided. The activation function of the encoding layer is the hyperbolic tangent. Figure 1 displays the encoded version of samples belonging to the Simda and Obfuscator.ACY malware families. You can observe that the encodings of samples belonging to the same family are similar while distinct from the encoding of samples belonging to a different family. This visual similarity is perhaps the result of reusing code to create new variants and the result of common obfuscation techniques. In consequence, by encoding an executable we can detect this local changes while retaining the global structure of the file.

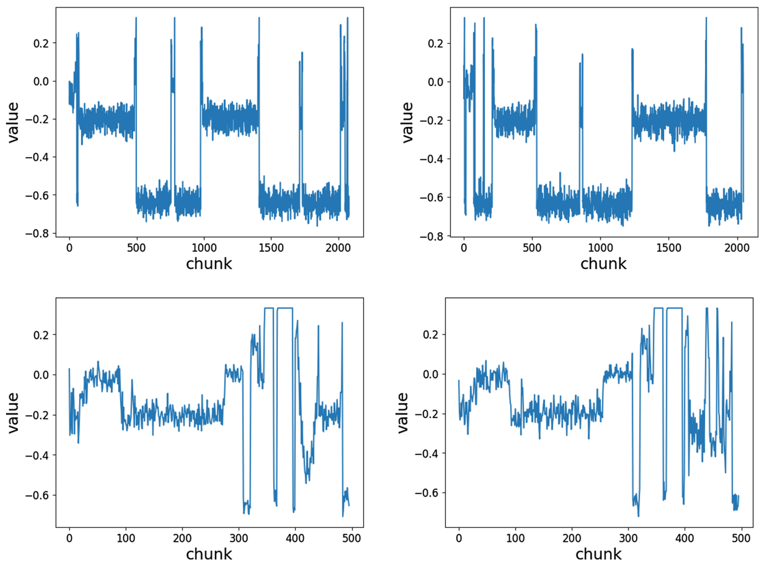


Fig. 1. Bytes encoding representation. Figures from the first and second row belong to the Simda and Obfuscator.ACY families, respectively

Step 2 Feature Extraction and Classification. The resulting time series is fed into a dilated residual network which learns descriptive patterns from the encoding of a bytes sequence and classifies a given malware binary into their corresponding family.

The overall architecture of the network is illustrated in Fig. 2. This architecture corresponds to the network that achieved a higher cross validation accuracy during evaluation. The hyperparameters of the network were selected using a grid search. The input is an univariate time series $m = m_1, m_2, ..., m_n$, where m_i corresponds to the encoding of the i-th chunk. The core of the network consists of 4 custom residual blocks [6] followed by one fully-connected layer and the output layer. The residual blocks perform feature learning while the later fully-connected layer combines the features learned. In particular, each residual block consists of a few stacked convolutional layers whose formulation is as follows:

$$h(x) = \sigma(W_2\sigma(W_1x + b_1) + b_2) + \sigma(W_3x + b_3) \qquad (1)$$

where x and $h(x)$ are the input and output of the residual block, W_i and b_i are the weights and biases of the i-th convolutional layer and σ is the activation function.

The input of each convolutional layer goes through a 3-stage feature extractor which learns hierarchical features through convolution, activation and pooling

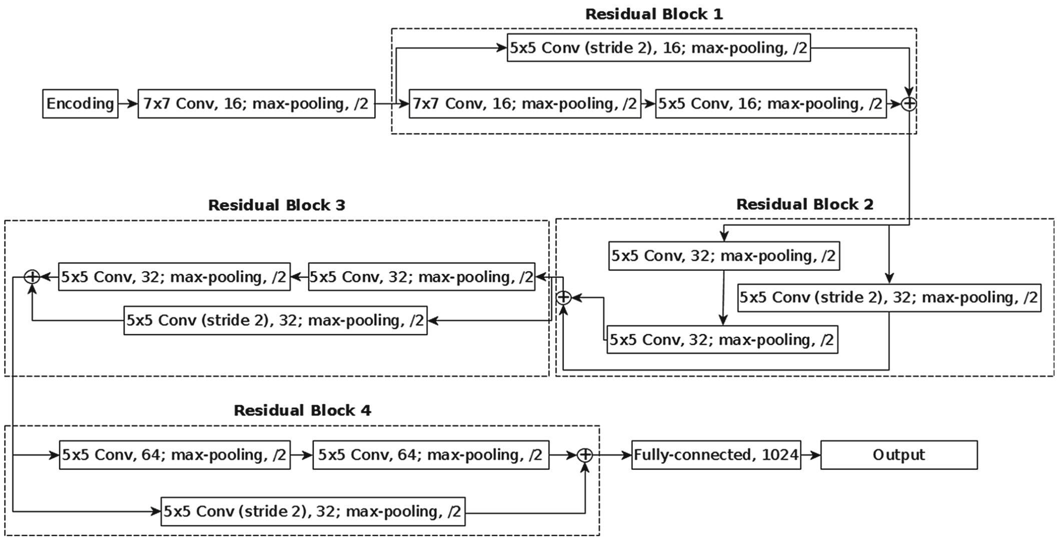

Residual Block 1
Encoding
7x7 Conv, 16; max-pooling, /2
5x5 Conv (stride 2), 16; max-pooling, /2
7x7 Conv, 16; max-pooling, /2
5x5 Conv, 16; max-pooling, /2
Residual Block 3
Residual Block 2
5x5 Conv, 32; max-pooling, /2
5x5 Conv (stride 2), 32; max-pooling, /2
5x5 Conv, 32; max-pooling, /2
5x5 Conv, 32; max-pooling, /2
5x5 Conv, 32; max-pooling, /2
5x5 Conv (stride 2), 32; max-pooling, /2
Residual Block 4
5x5 Conv, 64; max-pooling, /2
5x5 Conv, 64; max-pooling, /2
5x5 Conv (stride 2), 32; max-pooling, /2
Fully-connected, 1024
Output

Fig. 2. Overall architecture of the dilated residual network.

layers. More specifically, in the place of the convolution operation, we calculated a dilated convolution [12]. The activation function adopted in all layers the Exponential Linear Unit. Lastly, the pooling operation of our choice has been the MAX operation.

Afterwards, the feature maps extracted by the residual blocks are combined and fed as the input of the subsequent fully-connected layer plus a softmax layer for classification. Additionally, Xavier's initialization [5] has been used to make sure weights are neither too small or big to propagate accurately the signals. To prevent overfitting we employed dropout, a regularization mechanism that randomly drops a proportion of p units during forward propagation and prevents the co-adaptation between neurons.

3 Evaluation

3.1 Microsoft Malware Classification Challenge

The system has been evaluated on the dataset released by Microsoft for the Big Data Innovators Gathering Anti-Malware Prediction Challenge [10]. The dataset consists of 10868 samples for training and 10873 samples for testing of 9 malware families. For each sample, it is provided a file containing the hexadecimal's representation of the binary content and its corresponding disassembled file generated with IDA Pro.

3.2 Experimental Setup

The generalization performance of our approach has been estimated using 10-fold cross validation. Additionally, the best model has been selected according to the macro-averaged F1 score, which is the average of the individual F1 scores obtained for each class.

$$macro\_F_1 = \frac{1}{q}\sum_{i=1}^{q} F_1^i \tag{2}$$

where q is the number of classes in the dataset and F_1^i is the F1 score of class i. Furthermore, the model has been evaluated on the test set using the multi-class logarithmic loss.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(y_{i,j}log(p_{i,j}) + (1 - y_{i,j})log(1 - p_{i,j})) \tag{3}$$

where N is the number of observations, M is the number of class labels, log is the natural logarithm, $y_{i,j}$ is 1 if the observation i is in class j and 0 otherwise, and $p_{i,j}$ is the predicted probability that observation i is in class j.

3.3 State-of-the-art Comparison

To assess the generalization performance of our approach, we compared our model with state-of-the-art methods in the literature that are based on features extracted from the hexadecimal representation of malware on the Microsoft benchmark. These methods can be divided into two groups, depending on how they represent the information of binary executables.

1. Entropy-based approaches. This group includes approaches that are based on the representation of executable files as a stream of entropy values or their structural entropy. Concretely, Gibert et al. [4] evaluated the performance of both convolutional neural networks and the K-nearest neighbor algorithm.
2. IMG-based approaches. This group includes approaches that represent the binary content of an executable as a gray scale image. Such images are generally constructed by treating each byte of the binary as a gray-scale pixel value. In particular, Ahmadi et al. [1] and Narayanan et al. [9] extracted Haralick and Local Binary Pattern (LBP) features, and Principal Component Analysis (PCA) features, respectively.

Table 1. 10-fold cross validation confusion matrix

Family	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Ramnit	1520	4	0	0	0	1	1	11	4
Lollipop	7	2457	0	1	1	3	1	4	4
Kelihos_ver3	0	0	2940	0	0	1	0	1	0
Vundo	1	2	0	468	0	0	0	4	0
Simda	1	1	0	0	35	2	1	2	0
Tracur	2	2	1	5	1	726	1	7	6
Kelihos_ver1	1	1	0	0	0	0	394	1	1
Obfuscator.ACY	21	14	2	6	0	8	2	1172	3
Gatak	2	2	0	1	0	2	0	2	1005
Accuracy	10717 / 10868 = 0.9861					F1 score	0.9719		

Table 2. Performance comparison of state-of-the-art approaches based on the binary's content of an executable. The approach presented in this article is denoted "AE+DRN". "DTW+K-NN" refers to the K-nearest neighbor algorithm plus the dynamic time warping. "Haar approximation + DTW + K-NN" refers to the aforementioned method trained using the approximation time series obtained after applying the Haar Wavelet Transform to the entropy time series. "CNN entropy" and "CNN haar approximation & details" refer to the convolutional neural networks trained with the structural entropy of executables and the approximation and details coefficients obtained after applying the Haar Wavelet Transform to the structural entropy, respectively. "Haralick features + XGBoost" and "LBP features + XGBoost" refer to the models of Ahmadi et al. [1], which extracted Haralick and Local Binary Pattern features and trained boosted trees for classification. Moreover, Narayanan et al. [9] extracted PCA features and trained different models. "CNN IMG" refers to a convolutional neural network model trained on images of size 128 × 128 pixels. Those approaches that their authors have not tested their performance on the test set or didn't make public the training confusion matrix appear with a '-' mark. Approaches with a '*' mark indicate that they performed 5-fold cross validation instead of 10-fold cross validation.

	10-Fold accuracy	F1 score	Test logloss
Entropy-based approaches			
DTW + K-NN [4]	0.9894	0.9813	0.367724
Haar approximation + DTW + K-NN [4]	0.9870	0.9710	0.458191
CNN entropy [4]	0.9708	0.9314	0.134624
CNN haar approximation & details [4]	0.9828	0.9636	0.124431
IMG-based approaches			
Haralick features + XGBoost [1]*	0.955	-	-
LBP features + XGBoost [1]*	0.951	-	-
12 PCA features + 1-NN [9]	0.966	0.910	-
10 PCA features + SVM [9]	0.946	0.864	-
52 PCA features + SFN1 [9]	0.956	0.864	-
52 PCA features + SFN2 [9]	0.942	0.849	-
52 PCA features + DFN [9]	0.955	0.889	-
CNN IMG	0.975	0.940	0.184483
AE + DRN	0.9861	0.9719	0.106343

Table 1 presents the 10-fold cross validation accuracy and F1 score obtained on the training data. The major contributor to errors is the Obfuscator.ACY family which comprises malware that can have any purpose, whose code has been obfuscated and they couldn't be detected using their respective signatures and heuristics. This is produced because of the similarity in the encoding of some samples of the Obfuscator.ACY family and the rest. This issue affects the methods in the literature that are based on the hexadecimal representation of the binary content. Consequently, to correctly classify the remaining samples it might be necessary to use other type of features such as the assembly language instructions or the Windows API functions invoked.

Table 2 presents a comparison of the performance of state-of-the-art approaches based on the binary's content of an executable. The methods that are performing worse are those that represent the binary content of an executable as a gray scale image. This is because this kind of representation is counterintuitive. Binaries are not images and by constructing them you enforce non-existent 2D spatial correlations. Nevertheless, following recent trends in machine learning, it can be seen that deep learning aproaches outperform those that rely on the use of hand-designed feature extractors such as Haralick and LBP. On the other hand, the entropy-based convolutional neural network models outmatched the K-NN approaches on the test set and demonstrated a clear superior predictive power. Last but not least, our approach outperformed all the other methods on the test set and only the K-NN method achieved a greater macro-averaged F1 score on the training data, which as already mentioned, it failed to generalize to unseen data.

4 Conclusions

In this work we have described an end-to-end deep learning system for malware classification from raw byte sequences. This has been accomplished by learning an encoded representation of the malware's binary content using denoising autoencoders. Afterwards, a dilated residual network classifies the resulting malware's encoding into their corresponding family.

The proposed approach in this paper exhibits strong classification performance compared with the binary-based state-of-the-art methods in the literature. This is due to the exploitation of the visual similarity between malware samples belonging to the same family as the result of reusing code and using common obfuscation techniques to generate new samples. Therefore, the classifier learns descriptive and robust features through stacking various convolutional layers which are later used for classification purposes.

As far as we know, it is the first approach that applies deep learning for encoding malware's binary content. The main idea behind the encoding is to reduce the dimensionality of the input bytes sequence while being able to capture the main factors of variation in the data. The proposed solution has two major advantages with respect traditional machine learning approaches. First, it is file agnostic. That is, even that the solution has been evaluated on malware executables in Portable Executable format, it could be easily deployed for classifying malware in any other file format or targeting any other operative system. Second, it neither relies on costly feature engineering nor on expertise and domain-specific knowledge and thus, the extraction and prediction time are minimal.

4.1 Future Work

Even though machine learning solutions are a promising tool for detecting and classifying malware, they have their limitations. Specifically, they are susceptible to adversarial attacks that try to poison the training procedure or manipulate the binaries to bypass detection [2]. Due to the limitations of binary-based approaches, a future line of research might be studying how to transfer the features learned by the classifier as a subset of input features for M.L. models attempting to classify malware based on distinct types of file features.

Acknowledgments. This research has been partially funded by the Spanish MICINN Projects TIN2014-53234-C2-2-R, TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, and is supported by the University of Lleida.

References

1. Ahmadi, M., Giacinto, G., Ulyanov, D., Semenov, S., Trofimov, M.: Novel feature extraction, selection and fusion for effective malware family classification. CoRR abs/1511.04317 (2015)
2. Anderson, H.S., Kharkar, A., Filar, B., Evans, D., Roth, P.: Learning to evade static PE machine learning malware models via reinforcement learning. CoRR abs/1801.08917 (2018), http://arxiv.org/abs/1801.08917
3. Gibert, D., Bejar, J., Mateu, C., Planes, J., Solis, D., Vicens, R.: Convolutional neural networks for classification of malware assembly code. In: International Conference of the Catalan Association for Artificial Intelligence, pp. 221–226, October 2017. https://doi.org/10.3233/978-1-61499-806-8-221, http://www.ebooks.iospress.com/volumearticle/47742
4. Gibert, D., Mateu, C., Planes, J., Vicens, R.: Classification of malware by using structural entropy on convolutional neural networks. In: Proceedings of the Innovative Applications of Artificial Intelligence Conference (IAAI 2018). Association for the Advancement of Artificial Intelligence (2018)
5. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS 2010). Society for Artificial Intelligence and Statistics (2010)
6. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. CoRR abs/1512.03385 (2015). http://arxiv.org/abs/1512.03385
7. Jain, S., Meena, Y.K.: Byte level n–gram analysis for malware detection. In: Venugopal, K.R., Patnaik, L.M. (eds.) ICIP 2011. CCIS, vol. 157, pp. 51–59. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22786-8_6
8. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Secur. Anal. **5**, 40–45 (2007)
9. Narayanan, B.N., Djaneye-Boundjou, O., Kebede, T.M.: Performance analysis of machine learning and pattern recognition algorithms for malware classification. In: 2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS), pp. 338–342. IEEE (2016)

10. Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M.: Microsoft Malware Classification Challenge. ArXiv e-prints, February 2018)
11. Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G.: Opcode sequences as representation of executables for data-mining-based unknown malware detection. Inf. Sci. **231**, 64–82 (2013). https://doi.org/10.1016/j.ins.2011.08.020. data Mining for Information Security
12. Yu, F., Koltun, V.: Multi-scale context aggregation by dilated convolutions. CoRR abs/1511.07122 (2015). http://arxiv.org/abs/1511.07122

4.3.5 Limitations of Unimodal Approaches

So far, the research presented in this thesis has focused on deep learning approaches that take raw data as input and extract discriminant features through one or more convolutional layers that are later used for classification purposes. These approaches that take as input a single source of information or type of features are known as unimodal approaches. However, malware detection and classification is a research problem characterized as multimodal, as it includes multiple modalities of information. Multimodal machine learning aims to build models that can process and relate information from multiple modalities [7]. Modalities are, essentially, channels of information. These data from multiple sources are semantically correlated, and sometimes provide complementary information to each other, thus reflecting patterns that are not visible when working with individual modalities on their own. Thus, by only taking as input the raw bytes or opcodes sequence a great deal of useful information for classification is overlooked, such as structural information of the Portable Executable (PE) file, the import address table (IAT) which is used as a lookup table when the application is calling a function from a different module, etc. In fact, multimodal methods based on the combination of various hand-crafted features [3, 79] remain unbeaten in terms of classification performance and have been the way to go for detecting and classifying malware. These approaches extract multiple categories of features from the hexadecimal representation of malware's binary content and its assembly language source code, including, but not limited to:

Bytes and opcodes n-gram counts. M. Ahmadi et al. [3] extracted unigram features from both the hexadecimal representation and its disassembly counterpart, while Y. Zhang et al. [79] extracted the unigrams of a subset of about 280 opcodes based on their frequency used in malicious applications [10] and unigrams and bigrams from the bytes file. For more information about n-grams the reader is referred to Section 4.2.

Application Programming Interface (API) frequency. The frequency of use of Application Programming Interfaces (API) and their function calls are regarded as very characterizing features. Literature has demonstrated that API calls can be analyzed to model the program behavior. API functions and system calls are related with services provided by operating systems. They support various key operations such as networks, security, system services, file management, and so on. In addition, they include various functions to utilize system resources, such as memory, file system, network or graphics. There is no other way for software to access system resources that are managed by operating systems without using API functions or system calls and thus, API function calls can provide key information to represent the behavior of the software. For instance, Y. Zhang et al. [79] measured the frequency of all the APIs imported while M. Ahmadi et al. [3] only measured the frequency of 794 frequent APIs commonly used by malicious binaries.

Image-based features. An original way to represent a malware sample is to visualize the byte code by interpreting each byte as the gray-level of one pixel in an image [59]. From this representation some features can be extracted that describe the textures in an image such as the Haralick features and the

Local Binary Pattern features. For more information about the classification of malware represented as gray-scale images the reader is referred to Section 6.

Special symbols. The high frequency of the following set of symbols ,-,+,*,],[,?,@, is typical of code that has been designed to evade detection by utilizing indirect calls or dynamic library loading. On the one hand, indirect calls are those in which the address of the callee is taken from the memory or register. Although the implementation of the calls depends both on the architecture and the compiler, indirect calls may reveal some information on data location obfuscation [57]. On the other hand, dynamic library loading is used to load a library into memory at run-time and accesses its functions based on their address to prevent static analyzers from capturing the name of the imported functions.

Section characteristics. As described in Section 4.1.2, a PE file consists of some predefined sections like .text, .data, .bss, .rdata, .edata, .idata, .rsrc, .tls, and .reloc. Due to the usage of evasion techniques like packing, the predefined sections can be modified, reordered and new sections can be created. Consequently, it is common in methods in the literature to extract sets of features that capture the characteristics of these sections. For instance, the features extracted by M. Ahmadi et al. [3] are listed in Table 4.5.

4.4 Fusing Multiple Modalities of Information

Although combining different modalities or types of information for improving performance seems an intuitively appealing task, it is very challenging to combine the varying levels of noise and conflict between modalities. Multimodal approaches can be categorized into three groups, considering how the multiple modalities are combined.

- Input-level or early fusion. Early fusion methods create a joint representation of the unimodal features extracted separately from multiple modalities. The simplest way to combine these unimodal feature vectors is to concatenate them to obtain a fused representation. Cf. Figure 4.10. Next, a single model is trained to learn the correlation and interactions between the features of each modality. The final outcome of the model can be written as

  $$p = h\left([v_1, v_2, ..., v_m]\right)$$

  where h denotes the single model, $[v_1, v_2, ..., v_m]$ represents the concatenation of the feature vectors, and m is the number of distinct unimodal feature vectors.

- Decision-level or late fusion. In contrast to early fusion, late fusion methods train one model per modality and fuse the learned decision values with a fusion mechanism such as averaging, voting, a learned model, etc. Cf. Figure 4.11. The main advantage of late fusion is that it allows using different models on different modalities, thus being more flexible. In addition, as the predictions for each modality are made separately, it is easier to handle missing modalities.

Table 4.5: List of features extracted from the sections of a PE file.

Feature description
The total number of lines in .bss section
The total number of lines in .data section
The total number of lines in .edata section
The total number of lines in .idata section
The total number of lines in .rdata section
The total number of lines in .rsrc section
The total number of lines in .text section
The total number of lines in .tls section
The total number of lines in .reloc section
The total number of sections
The total number of unknown sections
The total number of lines in unknown sections
The proportion of known sections to all sections
The proportion of unknown sections to all sections
The proportion of the amount of unknown sections to the whole file
The proportion of .text section to the whole file
The proportion of .data section to the whole file
The proportion of .bss section to the whole file
The proportion of .rdata section to the whole file
The proportion of .edata section to the whole file
The proportion of .idata section to the whole file
The proportion of .rsrc section to the whole file
The proportion of .tls section to the whole file
The proportion of .reloc section to the whole file

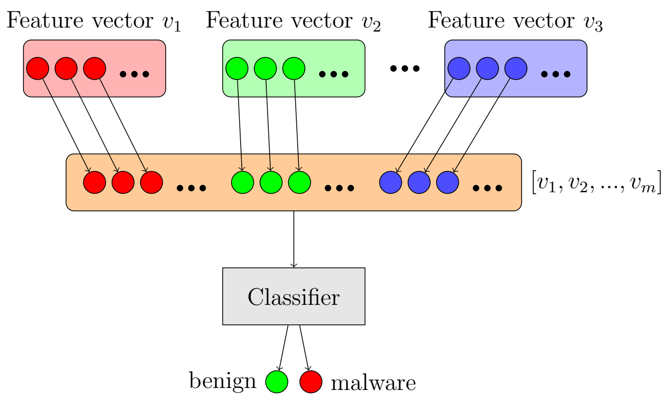


Figure 4.10: Early fusion strategy.

Supposing that model h_i is the decision value on modality i, the final prediction is

$$p = F\left(h_1(v_1), h_2(v_2), ...h_m(v_m)\right)$$

where F denotes the type of fusion strategy.

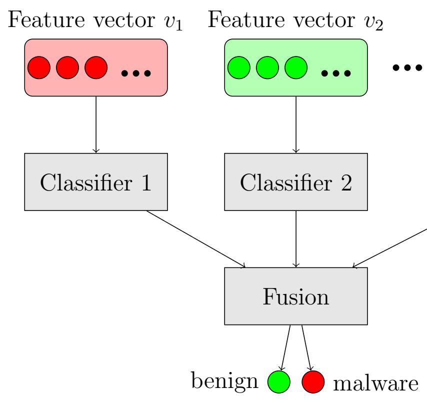


Figure 4.11: Late fusion strategy.

- Intermediate fusion. Intermediate fusion methods construct a shared representation by merging the intermediate features obtained by separate machine learning models. Afterwards, these intermediate features are concatenated and then a machine learning model is trained to capture the interactions between modalities. Cf. Figure 4.12.

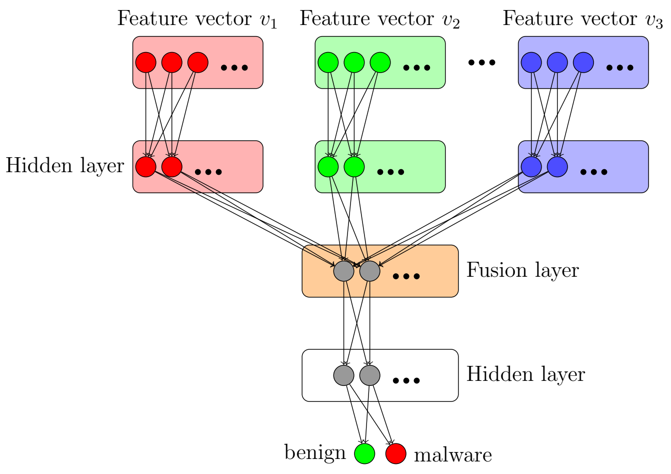


Figure 4.12: Intermediate fusion strategy.

For the task of malware classification, state-of-the-art classification accuracy is reported by approaches that perform early fusion to combine the different types of features [3, 79]. Both approaches implemented a variant of the forward stepwise selection algorithm to avoid using features irrelevant to the model, where instead of considering one feature at a time, they considered all the subset of features belonging to a feature category or type at a time. Thus, at each step, the feature set that produces the minimum value of the multi-class logarithmic loss is added to the model. The process stops when adding more features does not decrease the value of the logarithmic loss. Afterwards, the joint feature vector is passed as input to a classification algorithm that generates the corresponding classification output. On the one hand, M. Ahmadi et al. [3] employed bagging to boost the prediction of their XGBoost classifier [13]. On the other hand, Y. Zhang et al. [79] combined XGBoost [13] and Extra-Trees [23] to improve generalization of the final classifier and avoid overfitting.

However, neither gradient boosting classifiers nor Extra-Trees can handle raw data as input, e.g. feature engineering has to be performed before training the models. Thus, new ways to combine unimodal end-to-end learning classifiers have been studied during the elaboration of this thesis. Section 4.4.1 presents Orthrus [29], an end-to-end learning architecture that performs automatic feature learning and classification from two modalities of data: (1) the byte sequence representing the malware's binary content and (2) the assembly language source code of malware. Section 7 describes the European Patent Application EESR EP19382656, which presents a computer-implemented method, system and computer program to identify a malicious file that combines different types of analysis, processes and procedures that allow a malicious file to be detected and classified. The method comprises: (a) performing a static analysis of a potentially malicious file to obtain a set of features that provide an abstract view of the malicious file; (b) performing a static machine learning classification process using as inputs said set of features, to obtain a preliminary classification output; and (c) performing a fuzzy inference procedure based on fuzzy rules using as input said set of features and said preliminary classification output, to generate an enhanced classification output that identifies a potentially malicious file. Finally, Section 8 presents HYDRA [28], a wide and deep learning framework for malware classification that combines both hand-crafted features and end-to-end components.

4.4.1 Orthrus: A Bimodal Learning Architecture for Malware Classification

A major shortfall of end-to-end learning methods is their inability to consider multiple sources of information when performing classification, leading them to perform poorly in comparison to multimodal approaches. This research article attempts to address the aforementioned shortfall by introducing Orthrus [29], a bimodal approach to categorize malware into families based on deep learning. Orthrus combines two modalities of data: (1) the byte sequence representing the malware's binary content, and (2) the assembly language instructions extracted from the assembly language source code of malware, and performs automatic feature learning and classification with a convolutional neural network. The automatic feature learning process is carried through convolutional layers that extract n-gram like features from both the raw byte sequence and assembly language instructions corresponding to a given malicious file. Afterwards, the most discriminant features learnt by the filters are combined using an intermediate fusion strategy to produce a final decision, that is whether the given executable belongs to one family or another. The overall architecture is presented in Figure 4.4.1. For a complete description of the experimentation and for each of the layers the reader is referred to the original publication [29].

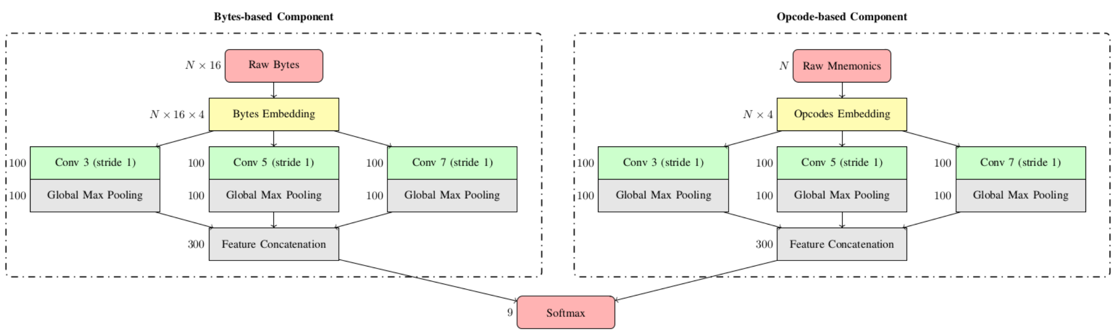


Figure 4.13: Bimodal architecture. The letters and the figures on the left side of the layers represent their sizes.

Orthrus: A Bimodal Learning Architecture for Malware Classification

Daniel Gibert
Dept. of Computer Science
University of Lleida
daniel.gibert@diei.udl.cat

Carles Mateu
Dept. of Computer Science
University of Lleida
carlesm@diei.udl.cat

Jordi Planes
Dept. of Computer Science
University of Lleida
jordi.planes@diei.udl.cat

***Abstract*—Malware detection and classification is a challenging problem and an active area of research. Traditional machine learning methods depend almost entirely on the ability to extract a set of discriminative features into which characterize malware. However, this feature engineering process is very time consuming. On the contrary, deep learning methods replace manual feature engineering by a system that performs both feature extraction and classification from raw data at once. Despite that, a major shortfall of these methods is their inhability to consider multiple disparate sources of information when performing classification, leading them to perform poorly when compared to multimodal approaches. In this work, we introduce Orthrus, a new bimodal approach to categorize malware into families based on deep learning. Orthrus combines two modalities of data: (1) the byte sequence representing the malware's binary content, and (2) the assembly language instructions extracted from the assembly language source code of malware, and performs automatic feature learning and classification with a convolutional neural network. The idea is to benefit from multiple feature types to reflect malware's characteristics. The experiments carried on the Microsoft Malware Classification Challenge dataset show that our proposed solution achieves higher classification performance than deep learning approaches in the literature and n-gram based methods.**

***Index Terms*—Malware Classification, Convolutional Neural Networks, Deep Learning, Multimodal Learning**

I. Introduction

The detection of malware, malignant computer software designed to infiltrate or damage a computer system without consent of the owner, is an important and challenging problem in cybersecurity. The global malware industry is estimated to be worth millions and grows year after year, with an underground services market which provides malicious software, cybercapabilities, and products to criminals, gangs, and even nation states. Recently, we have seen malware campaigns affecting our daily lives, influencing major elections, and crippling businesses overnight. The most notorious cyberespionage campaign affected the Democratic National Committee computer network and ended up with the release of private and confidential information from party members. In addition, the awareness of the danger of cyber threats increased due to the harm posed by major cyberattacks like WannaCry and Petya campaigns, among others, which held computer systems from all over the globe to ransom.

To limit the impact of cyberattacks it is necessary to improve computer systems' defenses. One essential layer is endpoint protection, specially anti-malware scanners, which is the last layer of defense against malware by preventing, detecting, and removing malicious software. Traditional anti-virus engines use a signature-based approach, where a signature is a set of manually defined rules that can identify a concrete piece of malware or a group with similar characteristics. However, this rules are generally specific, sensitive to small changes, and cannot usually recognize new malware. In consequence, the need for new methods to detect unknown malware is appealing for signature-less machine learning approaches due to their ability to summarize complex relationships and later decision making.

Traditional machine learning solutions perform feature engineering to manually extract a set of features that provide an abstract representation of malware. These features can be obtained from the static and dynamic analysis of malware. On the one hand, static analysis consists of examining the code or structure of a computer program without executing it. On the other hand, dynamic analysis monitors the execution of the program on the system. Indistinctly of the type of analysis, feature-based approaches depend almost entirely on the set of discriminative features used to represent malware. Contrarily, deep learning approaches obviate the need for manual feature engineering by automating the feature learning and classification procedure. Deep learning shifts the burden of feature engineering to an underlying system, typically consisting of a neural network with multiple layers, that jointly perform both feature learning and classification. For instance, E. Raff et al. [1] and D. Gibert et al. [2] trained a neural model by feeding it, as input, a sequence of bytes and a sequence of opcodes (machine language instruction), respectively. Nonetheless, both approaches lack the information from multiple sources of information that is combined before classification in traditional machine learning approaches. Thus, deep learning approaches for malware detection tend to perform poorly in comparison with multimodal approaches.

The primary contribution of this work is the development of the first, to our knowledge, bimodal deep learning architecture for malware classification. Orthrus automatically learns

This research has been partially funded by the Spanish MICINN Projects TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, and is supported by the University of Lleida.

features from two sources of information, (1) the hexadecimal representation of malware's binary content, and (2) the assembly language instructions representing the assembly language source code of malware. The idea is to learn from multiple sources of information to maximize the benefits of multiple features types to reflect the characteristics of malware and, to compensate for the weaknesses inherent in unimodal representations. The generalization performance of our bimodal learning approach has been evaluated on the dataset provided by Microsoft for the Big Data Innovators Gathering Anti-Malware Prediction Challenge [3]. Furthermore, we present a comparison with deep learning methods in the literature. Experiments show that our model successfully takes advantage of both modalities of information to perform significantly better than unimodal deep learning methods.

The rest of the paper is organized as follows. Firstly, we introduce the state-of-the-art approaches to address the problem of malware detection and classification. Afterwards, we describe the bimodal architecture followed by the results of the experimentation. Lastly, we summarize the concluding remarks of our research and proposes some future lines of research.

II. Related Work

Traditional machine learning solutions rely on a set of hand-designed features that provide an abstract representation of the program that is later used for classification. The most common features are byte and opcode n-grams [4], [5]. Byte n-grams are extracted from the hexadecimal representation of malware's binary content whereas opcode n-grams are extracted from the assembly language source code of malware.

To detect the presence of compressed and encrypted segments hidden beneath the executable, security researchers use entropy analysis, as compressed and encrypted segments tend to have higher entropy in comparison with native code [6]. However, simple entropy statistics is not enough to detect sophisticated malware, as packed and encrypted code is often concealed in a way that pass through entropy filters. Thus, researchers started analyzing the structural entropy of executables [7]. The structural entropy consists of a stream of entropy values, where each value describes the amount of entropy over a chunk of code in a specific location of the executable.

A distinct way to represent an executable is to visualize its byte code as a grayscale image [8], where every byte is interpreted as one pixel in the image. Afterwards, features describing the texture of the grayscale image can be extracted such as GIST [8], Haralick [9], Local Binary Patterns [9] and PCA features [10].

In addition, the usage of system functions and libraries is a good indicator of the behavior of malware as they offer information about services and resources provided by the OS [11].

The need for manual feature engineering can be obviated by automated feature learning. Deep learning replaces the feature engineering process by an underlying system which typically consists of a neural network with multiple layers, that performs both feature learning and classification. With deep learning, one can start with raw data as features will be automatically created by the neural network when it learns. The main distinction between deep learning approaches for malware detection and classification lean on what they used as raw data.

D. Gibert et al. [2] and N. McLaughlin et al. [12] feed convolutional networks with the opcode sequences extracted from disassembled Portable Executables (PEs) and Android APKs, to classify malicious software targeting the Windows and the Android operative systems, respectively. The shallow layers of the convolutional networks allow to extract N-gram like features without consuming the exploding amount of computational resources required to extract N-grams for a long N. Alternatively, D. Gibert et al. [13] take advantage of the hierarchical structure of Portable Executable files to build a hierarchical convolutional network that extracts features at the mnemonics-level and at the function-level.

On the contrary, E. Raff et al. [1] presented a detection system trained on raw byte sequences. In their work, each byte of the input sequence is embedded into a fixed length feature vector to avoid representing each byte by its value, as it would imply that certain byte values are closer to each other than other byte values, which is false, as the byte value depends on its context. Afterwards, they combined convolution layer with global max-pooling to obtain the activations regardless of the location of the detected features. This shallow architecture applied filters of width equals to 500 bytes followed by an stride of 500, which allowed to identify interpretable subregions of the binary, mostly from the PE header. Furthermore, M. Krl et al. [14] presented a deeper architecture consisting of 11 layers: the embedding layer, four convolutional and two pooling layers, followed by 4 fully-connected layers.

As the length of the bytes sequence might be up to various million time steps, other works preprocessed the sequence to reduce its size and compress its information. D. Gibert et al. [15] feed a convolutional neural network with the structural entropy representation of malware. Hence, the size of the sequence was diminished from millions to thousands or hundreds, depending on the chunk size. Alternatively, D. Gibert et al. [16] generated an encoded representation of contiguous, non-overlapping chunks using a denoising autoencoder. Afterwards, a residual network extracts features from the compressed sequence and performs classification. Q. Le et al. [17] scaled the file byte code to a fixed target size using a generic image scaling algorithm. After scaling, a malware sample corresponds to one sequence of 10000 values. For classification purposes, they applied recurrent neural network layers on top of the convolutional layers.

D. Gibert et al. [18] takes advantage of the representation of malware as a grayscale image [8] to build a convolutional neural network classifier that automatically extracts discriminant features from the image. Moreover, R. Khan et al. [19] analyzed the performance of the ResNet and GoogleNet architectures for the task at hand.

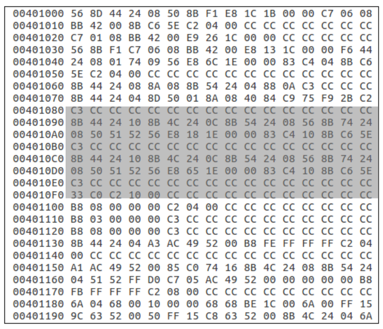
A further way to represent malware is as an ordered sequence of API functions invoked during its execution. To capture the long-term dependencies in the API function traces, B. Athiwaratkun et al. [20] examined recurrent neural network architectures. In the first stage, a LSTM or GRU constructs the features associated to a particular API trace and later, a single fully-connected layer or logistic regression with sofmax perform classification.In addition, B. Kolosnjaji [21] constructed a neural network classifier based on convolutional and recurrent layers that combines convolution of n-grams with sequential modeling provided by the recurrent layers.

III. Classification of Malware using a Bimodal Architecture

The main focus of this research is the classification into families of malware targeting the Windows operating system (OS). The most common executable file extension for Windows systems is the Portable Executable (PE) file format. In particular, this file format is used for executables, object code, DLLs, FON Font files, and others in 32-bit and 64-bit versions of the Windows OS. To this end, the method has been evaluated on the Microsoft Malware Classification Challenge dataset [3].

A. N-gram approaches

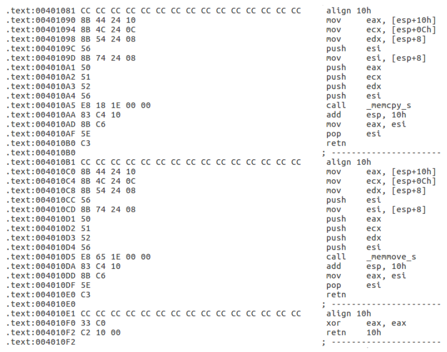
Static machine learning solutions for malware detection and classification extract features from either the hexadecimal representation of malware's binary content or its assembly language source code counterpart. The hexadecimal representation is a simple way to represent the binary's content of a PE file. Using this representation the binary content is represented as a sequence of bytes (base-16 number representation with digits $[0-9]$ and $[A-F]$). See Figure 1 for an hex view of a PE file. The main advantage of representing malware as a



```
00401000 56 8D 44 24 08 50 8B F1 E8 1C 1B 00 00 C7 06 08
00401010 BB 42 00 8B C6 5E C2 04 00 CC CC CC CC CC CC CC
00401020 C7 01 08 BB 42 00 E9 26 1C 00 00 CC CC CC CC CC
00401030 56 8B F1 C7 06 08 BB 42 00 E8 13 1C 00 00 F6 44
00401040 24 08 01 74 09 56 E8 6C 1E 00 00 83 C4 04 8B C6
00401050 5E C2 04 00 CC CC CC CC CC CC CC CC CC CC CC CC
00401060 8B 44 24 08 8A 08 8B 54 24 04 88 0A C3 CC CC CC
00401070 8B 44 24 04 8D 50 01 8A 08 40 84 C9 75 F9 2B C2
00401080 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401090 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010A0 08 50 51 52 56 E8 18 1E 00 00 83 C4 10 8B C6 5E
004010B0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010C0 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010D0 08 50 51 52 56 E8 65 1E 00 00 83 C4 10 8B C6 5E
004010E0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010F0 33 C0 C2 10 00 CC CC CC CC CC CC CC CC CC CC CC
00401100 B8 08 00 00 00 C2 04 00 CC CC CC CC CC CC CC CC
00401110 B8 03 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401120 B8 08 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401130 8B 44 24 04 A3 AC 49 52 00 B8 FE FF FF FF C2 04
00401140 00 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401150 A1 AC 49 52 00 85 C0 74 16 8B 4C 24 08 8B 54 24
00401160 04 51 52 FF D0 C7 05 AC 49 52 00 00 00 00 00 B8
00401170 FB FF FF FF C2 08 00 CC CC CC CC CC CC CC CC CC
00401180 6A 04 68 00 10 00 00 68 68 BE 1C 00 6A 00 FF 15
00401190 9C 63 52 00 50 FF 15 C8 63 52 00 8B 4C 24 04 6A
```

Fig. 1. Hexadecimal view of a PE file.

sequence of bytes is that it is OS resilient, i.e., it could be used to represent malware indistinctively of the target OS and hardware. Alternatively, the assembly language source code contains the symbolic machine code of the executable as well as metadata information such as rudimentary function calls, memory allocation and variable information. See Figure 2 for the assembly view of the grayed area in Figure 1.



```
.text:00401081 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:00401090 8B 44 24 10                                      mov     eax, [esp+10h]
.text:00401094 8B 4C 24 0C                                      mov     ecx, [esp+0Ch]
.text:00401098 8B 54 24 08                                      mov     edx, [esp+8]
.text:0040109C 56                                               push    esi
.text:0040109D 8B 74 24 08                                      mov     esi, [esp+8]
.text:004010A1 50                                               push    eax
.text:004010A2 51                                               push    ecx
.text:004010A3 52                                               push    edx
.text:004010A4 56                                               push    esi
.text:004010A5 E8 18 1E 00 00                                   call    _memcpy_s
.text:004010AA 83 C4 10                                         add     esp, 10h
.text:004010AD 8B C6                                            mov     eax, esi
.text:004010AF 5E                                               pop     esi
.text:004010B0 C3                                               retn
.text:004010B0                                                 ; ---------------------
.text:004010B1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010C0 8B 44 24 10                                      mov     eax, [esp+10h]
.text:004010C4 8B 4C 24 0C                                      mov     ecx, [esp+0Ch]
.text:004010C8 8B 54 24 08                                      mov     edx, [esp+8]
.text:004010CC 56                                               push    esi
.text:004010CD 8B 74 24 08                                      mov     esi, [esp+8]
.text:004010D1 50                                               push    eax
.text:004010D2 51                                               push    ecx
.text:004010D3 52                                               push    edx
.text:004010D4 56                                               push    esi
.text:004010D5 E8 65 1E 00 00                                   call    _memmove_s
.text:004010DA 83 C4 10                                         add     esp, 10h
.text:004010DD 8B C6                                            mov     eax, esi
.text:004010DF 5E                                               pop     esi
.text:004010E0 C3                                               retn
.text:004010E0                                                 ; ---------------------
.text:004010E1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010F0 33 C0                                            xor     eax, eax
.text:004010F2 C2 10 00                                         retn    10h
.text:004010F2                                                 ; ---------------------
```

Fig. 2. Assembly view of the grayed part in Figure 1. The first column represents the address, the second column the byte sequence and the third column the mnemonics sequence.

The most common type of features are n-grams. An n-gram is a contiguous sequence of n items from a given sequence of text. N-grams can be extracted from the bytes sequence representing malware's binary content and from the instruction statements extracted from the assembly language source code. By treating a file as a sequence of bytes, byte n-grams are extracted by looking at the unique combination of every n consecutive bytes as an individual feature. On the other hand, n-grams from the assembly language source code refer to the unique combination of every n consecutive opcodes, e.g. the instructions `ADD`, `MUL`, `POP`.

N-gram based methods construct a feature vector representation of malware where each element in the vector indicates the number of appearances of a particular n-gram in the instruction sequence. Thus, the length of the feature vector depends on the number of unique n-grams, which increases exponentially with n. As an example, considering the extraction of bytes n-grams with $n = 3$, the number of possible n-grams is $256^3 = 16,777,216$. Although malware n-grams tend to follow a Zipfian distribution [22], the resulting feature vector is still too large to keep in memory, and even if it is not, you still have to optimize a function with too many input variables, a.k.a. the curse of dimensionality. N-gram based approaches in the literature have reduced this high dimensional input space using feature selection techniques [5], [23] or the hashing trick [24], [25]. On the one hand, feature selection techniques select a subset of relevant features from the initial input space. On the other hand, feature hashing, a.k.a. the hashing trick, is a method for handling sparse, high-dimensional features by using a hash function to determine the feature's location in a vector of lower dimension. It can be seen as a random projection of the the input space $A \in \mathbb{R}^n$ to a low dimensional space $B \in \mathbb{R}^m$, where $m \ll n$. In our case, an array of size N that counts the number of times each n-gram occurred, and a hash function map each n-gram to a location in a lower dimensional array, which will be later used for training a classification algorithm.

In spite of the technique, both feature selection and feature hashing require to exhaustively enumerate a large number of n-grams during training. To overcome this limitation, D. Gibert et al. [2] explored the application of convolutional networks to malware classification by the assembly language language instructions as a text to be analyzed. This approach has the advantage that the features are automatically inferred from raw data and hence, it removes the feature extraction and selection steps. Similarly, E. Raff et al. [1] and M. Krl et al. [14] presented convolutional neural network architectures to detect malware from raw byte sequences. The main drawback of the aforementioned deep learning approaches is that they focus on only one source of information, either the opcode or the bytes sequence representation, and malware authors can exploit this information to easily bypass detectors [26]. As a result, the most accurate Machine Learning systems for malware detection and classification are still those that are able to extract and combine subsets of features from various sources of information [27].

B. Network Architecture

To overcome the current limitations of deep learning systems in this paper we present Orthrus, a baseline learning system to categorize malware into families that involves various modalities of data. The main idea is to learn from various sources of information to maximize the benefits of multiple feature types to reflect the characteristics of malware. To obviate manual feature engineering, a neural network is used to perform both feature learning and classification. As a result, the network receives as input (1) the sequence of hexadecimal values representing malware's binary content and (2) the sequence of assembly language instructions from the assembly language source code. The automatic feature learning process is carried through a convolutional layer that extracts N-gram like features from both input sequences. Afterwards, the most discriminative features learnt by the filters are combined to produce a final decision, whether the given executable belongs to one family or another. The process of merging intermediate features from the modalities of information is known as intermediate fusion. The overall architecture is presented in Figure 3. It comprises the following layers:

- Bytes input layer. Instead of taking as input an executable represented as a bytes sequence, bytes were grouped into subsequences representing the bytes content of its assembly language source code counterpart. For instance, taking the assembly view in Figure 2 as example, bytes were grouped as: [8B, 44, 24, 10], [8B, 4C, 24, 0C], [8B, 54, 24, 08], [56], [8B, 74, 24, 08], and so on. The maximum sequence length is 16. All subsequences with lesser length were filled with PAD tokens. Initially we considered to use one of the aforementioned architectures [1], [14] but they performed poorly due to the size of the filters and the limited number of samples regarding some families. See Figures 7 and 8. Thus, by grouping the bytes into subsequences and by reducing the size of the kernels we facilitated the learning of simpler and discriminant features.
- Mnemonics input layer. This layer takes as input an executable represented as a sequence of mnemonics. A mnemonic is simply the name of the assembly language instruction. In other words, the parameters of the instruction are removed. For example, the instruction *mov eax, [esp + 10h]* is reduced to *mov*. The maximum number of mnemonics per executable is determined by N, which is set to 10000.
- Embedding layers. As the network cannot be fed with just text strings, each token (either byte or mnemonic) is represented as a low-dimensional vector of real values, also known as word embedding, of size E. In our experiments E has been set to 4. We tried various values [4, 8, 16, 32] for the embedding size and we saw that increasing E does not lead to an increase in accuracy. In addition, increasing the embedding size also increases the memory requirements and in the case of the hexadecimal sequence, it is prohibitive in terms of resources and computational time.
- Bytes convolutional layer. This layer is responsible for convolving various filters over the bytes input and learn filters that activate when a particular feature is found. The size of each filter is $h \times 16 \times E$ where $h \in \{3, 5, 7\}$. Thus, filters are applied to encompass 3, 5 and 7 subsequences at once.
- Mnemonics convolutional layer. This layer convolves various filters over the mnemonics sequence to extract N-gram like features from it. The size of each filter is $h \times E$, where $h \in \{3, 5, 7\}$. As a result, filters are applied to sequences of 3 to 7 mnemonics. The aim behind having filters of various sizes is to allow the network to detect discriminant subsequences that have variations in size.
- Global max-pooling layer. Global max-pooling is applied to extract the maximum activation of each of the feature map activations passed from the convolutional layer.
- Softmax layer. Lastly, the softmax layer combines the features learned and applies the softmax function to output the probability distribution over malware families.

In our experiments we observed that taking both modalities of information as input is suboptimal, since it leads to overfitting one subset of features belonging to one modality and underfitting the features belonging to the other. To address this issue we separately pretrain each subnetwork and optimize their hyperparameters for each subtask. Afterwards, the learned weights of each subnetwork are used to initialize the bimodal network and thus, the knowledge learned by each model is transferred into the bimodal network to save training time and help the network converge faster. Furthermore, to make the network less sensitive to a particular modality it is applied modality dropout, which randomly drops one data modality during training. In addition, dropout has been applied in the softmax layer with a dropout rate equal to 0.5. The nonlinearity function adopted is the Exponential Linear Unit (ELU) [28].

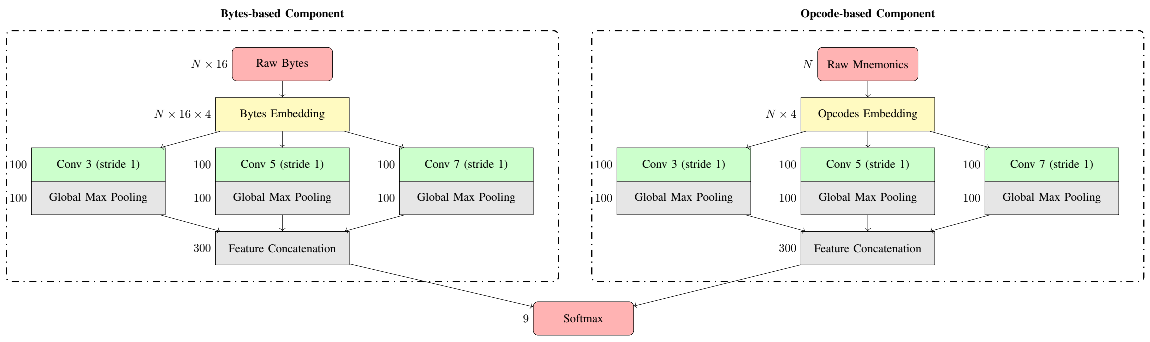


Fig. 3. Bimodal architecture. The letters and the figures at the left side of the layers represent their sizes.

IV. Evaluation

The method has been evaluated on the Microsoft Malware Classification Challenge dataset [3], a standard benchmark for research.

A. Microsoft Malware Classification Challenge

In comparison with other relevant tasks such as image classification, speech recognition, text classification, etc, much of the previous work on malware detection use data not available to public. In consequence, it is not possible to meaningfully compare performance across works as different datasets use different labeling procedures. To simplify comparison and reproducibility we decided to evaluate the performance of our approach on the Microsoft Malware Classification Challenge dataset [3], a standard benchmark for malware research. The dataset is publicly available on Kaggle[1]. It contains the hexadecimal representation of the malware's binary content and its disassembly counterpart. The set of samples represent 9 different families. Cf. Table I. One particularity of the dataset is that the distribution of samples per family is not balanced, i.e., there are some classes with a considerably greater number of samples in comparison with others. Additionally, the average number of bytes and opcodes differs greatly for each class. See Figures 4 and 5. You can observe that those classes with greater number of opcodes do not necessarily coincide with those with the greater number of bytes. This is because the bytes representation includes information of several PE sections, e.g. `.data`, `.edata`, `.idata`.

TABLE I
Class distribution in the Microsoft dataset

Family	#Instances	Type
Ramnit	1541	Worm
Lollipop	2478	Adware
Kelihos_ver3	2942	Backdoor
Vundo	475	Trojan
Simda	42	Backdoor
Tracur	751	TrojanDownloader
Kelihos_ver1	398	Backdoor
Obfuscator.ACY	1228	Obfuscated malware
Gatak	1013	Backdoor

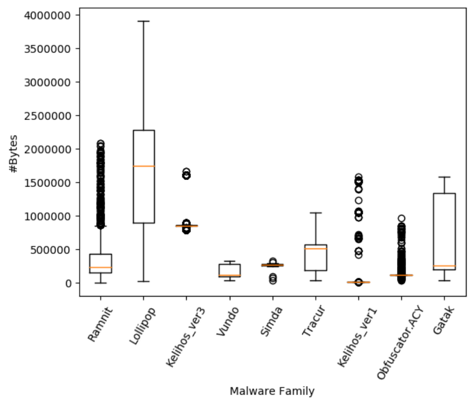


Fig. 4. Distribution of bytes per class.

B. Experimental Setup

The experiments were run on a computer with the following hardware specifications: Intel i7-7700K, 32 GB RAM, 2xNvidia GTX 1080Ti. This allowed us to take advantage of the multi-GPU setup during training to distribute some parts of the model to different GPU instances. That is, each subcomponent of the network was distributed on a different GPU instance.

The generalization performance of our approach has been estimated using 10-fold cross validation. However, instead of evaluating the model with accuracy alone, we selected the best model according to the F1-score. This is because accuracy can be a misleading measure in datasets were there exist a large class imbalance. For instance, a model can correctly predict the value of the majority class for all predictions and achieve a high classification accuracy while making mistakes on the minority and critical classes. In our case, a model can achieve a very high accuracy on the Microsoft dataset by correctly classifying the majority classes and misclassifying samples belong to the `Simda` family. The F1-score metric penalizes

[1]https://www.kaggle.com/c/malware-classification/

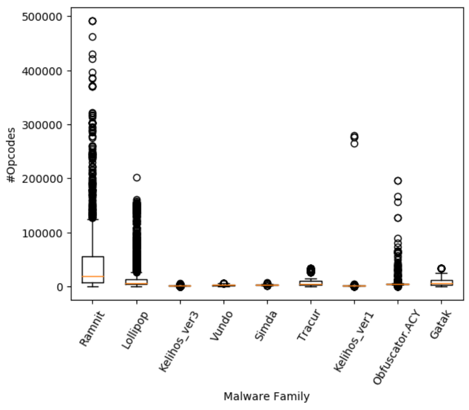
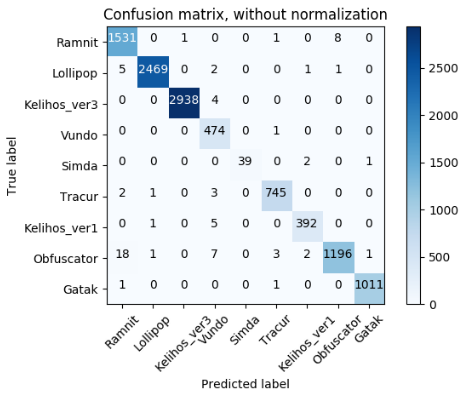


Fig. 5. Distribution of opcodes per class.

this kind of behavior and best meet the requirements of the dataset. Alternatively, we unsuccessfully tried the balanced cross-entropy loss. The results obtained were slightly worse.

C. Comparison with the State-of-the-Art

The 10-fold cross validation confusion matrix is presented in Figure 6. Notice that the percentage of errors in the minority classes does not differ from the number of errors on the majority classes. All classes are classified with more than 97% of accuracy with the exception of the `Simda` family which failed to classify 3 of the 42 samples during 10-fold cross validation. On the other hand, the major contributor to misclassifications is the `Obfuscator.ACY` family, which according to Microsoft, is malware that uses a combination of obfuscation techniques such as encryption, compression, anti-debugging, anti-emulation, etc, to hide its purpose, and thus, are way harsher to classify correctly.



	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator	Gatak
Ramnit	1531	0	1	0	0	1	0	8	0
Lollipop	5	2469	0	2	0	0	1	1	0
Kelihos_ver3	0	0	2938	4	0	0	0	0	0
Vundo	0	0	0	474	0	1	0	0	0
Simda	0	0	0	0	39	0	2	0	1
Tracur	2	1	0	3	0	745	0	0	0
Kelihos_ver1	0	1	0	5	0	0	392	0	0
Obfuscator	18	1	0	7	0	3	2	1196	1
Gatak	1	0	0	0	0	1	0	0	1011

Fig. 6. Orthrus confusion matrix

To evaluate the performance of our bimodal approach, we compared our model with state-of-the-art methods in the literature that have evaluated their model on the Microsoft Malware Classification Challenge dataset. The results are shown in Table II. Existing deep learning approaches for malware classification can be categorized into various groups depending on their corresponding input. With the exception of M. Mays et al. [30], these approaches take as input a single modality of information and perform feature extraction and classification altogether. D. Gibert et al. [18] and J. Kim et al. [29] take as input the grayscale representation of the malware's binary content. D. Gibert et al. [15] represents the content of a malicious program as an entropy stream, where each value describes the amount of entropy of a small chunk of code in a specific location of the file. E. Raff et al. [1] and M. Krl et al. [14] treat each byte as a unit in a sequence and thus, presented architectures to process raw byte sequence of over a few million steps. On the contrary, D. Gibert et al. [16] and Q. Le et al. [17] preprocessed the byte sequence and reduced the size of the input with autoencoders and data compression techniques, respectively. Furthermore, N. McLaughlin et al. [12] and D. Gibert et al. [13] extract N-gram like features from the assembly language instructions of the assembly language content using one or various convolutional layers. Lastly, M. Mays et al. [30] learn two distinct models, one taking as input a grayscale representation of the malware's binary content and the second taking as input a feature vector indicating the presence of particular opcode N-grams. Afterwards, an ensemble classifier returns the final prediction. In addition, we implemented various n-gram classification systems using the hashing trick as baselines. The classification algorithms implemented are logistic regression (LR) and feed-forward neural networks (NN) with one or two hidden layers. The number of hidden neurons is [256] and [256,128] for the neural networks with one and two hidden layers, respectively. The non-linearity applied is the ReLU function. Furthemore, dropout was applied between layers. With the exception of the unigram models, the 2-gram and 3-gram based classification systems apply the hashing trick to map every n-gram into a lower dimensional vector of size 500. The hashing trick has been indispensable to reduce the high dimensionality of the input space. Cf. Table III. As it can be observed in Table II, the bimodal approach outperforms by some margin the existing deep learning methods. Notice that each subnetwork achieves higher accuracy and F1-score than those methods that take as input either the grayscale image representation of malware, its structural entropy or the raw byte sequence. Moreover, the intermediate fusion of features from both the opcode and byte sequences achieve better performance than opcode-based methods. This is because there are some malware instances in the dataset that have been obfuscated with compression and encryption techniques and have very few instructions or none. Thus, the features from the byte sequence provide helpful information and boost the classifier. Additionally, the bytes subnetwork overcome methods [1], [14] for various reasons. First, they have higher complexity (more layers, bigger filter sizes) which make their architectures not suitable for small-size datasets. Second, the 2-dimensional representation of the

TABLE II
STATE-OF-THE-ART COMPARISON OF DEEP LEARNING METHODS FOR MALWARE CLASSIFICATION.

Method	Input	Accuracy	F1-score
LR	Byte 1-Gram	0.8785	0.7549
NN 1H	Byte 1-Gram	0.9718	0.9503
LR	Opcode 1-Gram	0.9911	0.9867
NN 1H	Opcode 1-Gram	0.9932	0.9833
NN 2H	Opcode 1-Gram	0.9865	0.9764
LR	Opcode 2-Gram	0.9729	0.9518
NN 1H	Opcode 2-Gram	0.9857	0.9761
NN 2H	Opcode 2-Gram	0.9871	0.9782
LR	Opcode 3-Gram	0.9545	0.9075
NN 1H	Opcode 3-Gram	0.9758	0.9530
NN 2H	Opcode 3-Gram	0.9650	0.9415
D. Gibert et al. [18]	Grayscale images	0.9750	0.9400
J. Kim et al. [29]	Grayscale images	0.9639	–
D. Gibert et al. [15]	Structural Entropy	0.9828	0.9314
E. Raff et al. [1]	Bytes sequence	0.9641	0.8902
M. Krl et al. [14]	Bytes sequence	0.9756	0.9071
Q. Le et al. [17]	Bytes sequence	0.9820	0.9605
D. Gibert et al. [16]	Bytes sequence	0.9828	0.9636
N. McLaughlin et al. [12]	Opcodes sequence	0.9903	–
D. Gibert et al. [13]	Opcodes sequence	0.9913	0.9830
M. Mays et al. [30]	Grayscale images + Opcode N-grams	0.9770	–
Mnemonics subnetwork	Opcodes sequence	0.9893	0.9802
Bytes subnetwork	Bytes sequence	0.9885	0.9774
Bimodal network	Opcodes+Bytes sequences	**0.9924**	**0.9872**

TABLE III
NUMBER OF UNIQUE N-GRAMS IN THE MICROSOFT MALWARE CLASSIFICATION CHALLENGE DATASET.

	Bytes	Opcodes
1-gram	256	400
2-gram	65536	21036
3-gram	16777216	197442

byte sequence presented in this work allows to group some of the bytes per default, and provides some insights about their function to the network.

V. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, this research is the first application of multimodal deep learning for PE malware classification that uses intermediate fusion to merge features from various modalities of information. The multimodal approach combines two sources of information through a simple architecture, (1) the byte sequence representing malware's binary content and (2) the mnemonic sequence representing malware's assembly language source code. This architecture extracts n-gram like features from both input sequences to build a robust classifier than existing deep learning approaches. Experiments demonstrate that our model takes advantage of both modalities of information to perform significantly better than state-of-the-art methods on a standard benchmark, the Microsoft Malware Classification Challenge dataset.

A future line of research might be to explore more modalities of data to build a stronger malware classifier. More specifically, research on the combination of wide and deep models [31] to combine the strength of both approaches, memorization (wide models) and generalization (deep models).

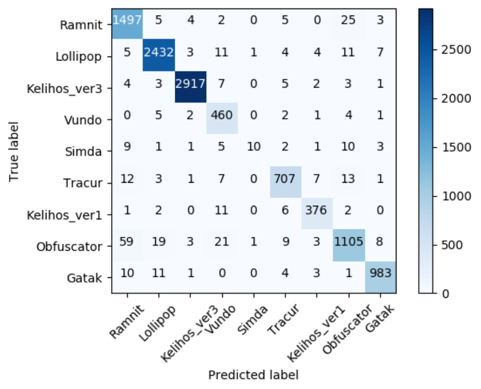


Fig. 7. E. Raff et al. [1] confusion matrix.

REFERENCES

[1] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole EXE," in *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018.*, 2018, pp. 268–276. [Online]. Available: https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422

[2] D. Gibert, J. Béjar, C. Mateu, J. Planes, D. Solis, and R. Vicens, "Convolutional neural networks for classification of malware assembly code," in *Recent Advances in Artificial Intelligence Research and Development - Proceedings of the 20th International Conference of the Catalan Association for Artificial Intelligence, Deltebre, Terres de l'Ebre, Spain, October 25-27, 2017*, 2017, pp. 221–226. [Online]. Available: https://doi.org/10.3233/978-1-61499-806-8-221

[3] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft Malware Classification Challenge," *ArXiv e-prints*, Feb. 2018.

[4] S. Jain and Y. K. Meena, "Byte level n–gram analysis for malware detection," in *Computer Networks and Intelligent Computing*, K. R.

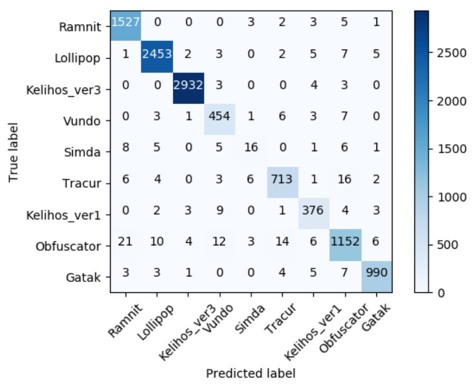


Fig. 8. M. Krl et al. [14] confusion matrix.

Venugopal and L. M. Patnaik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 51–59.

[5] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, pp. 64–82, 2013, data Mining for Information Security. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025511004336

[6] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security Privacy*, vol. 5, no. 2, pp. 40–45, March 2007.

[7] I. Sorokin, "Comparing files using structural entropy," *Journal in Computer Virology*, vol. 7, no. 4, p. 259, Jun 2011.

[8] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: Visualization and automatic classification," in *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, ser. VizSec '11. New York, NY, USA: ACM, 2011, pp. 4:1–4:7.

[9] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16. New York, NY, USA: ACM, 2016, pp. 183–194.

[10] B. N. Narayanan, O. Djaneye-Boundjou, and T. M. Kebede, "Performance analysis of machine learning and pattern recognition algorithms for malware classification," in *2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*, July 2016, pp. 338–342.

[11] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware detection based on mining api calls," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 1020–1025. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774303

[12] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn, "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: ACM, 2017, pp. 301–308.

[13] D. Gibert, C. Mateu, and J. Planes, "A hierarchical convolutional neural network for malware classification," in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019.

[14] M. Krl, O. vec, M. Blek, and O. Jaek, "Deep convolutional malware classifiers can learn from raw executables and labels only," in *Workshop in the Sixth International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=HkHrmM1PM

[15] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Classification of malware by using structural entropy on convolutional neural networks," in *The Thirtieth AAAI Conference on Innovative Applications of Artificial Intelligence (IAAI-18)*, 2018. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16133

[16] D. Gibert, C. Mateu, and J. Planes, "An end-to-end deep learning architecture for classification ofmalware's binary content," in *Artificial Neural Networks and Machine Learning – ICANN 2018*, V. Kůrková, Y. Manolopoulos, B. Hammer, L. Iliadis, and I. Maglogiannis, Eds. Cham: Springer International Publishing, 2018, pp. 383–391.

[17] Q. Le, O. Boydell, B. M. Namee, and M. Scanlon, "Deep learning at the shallow end: Malware classification for non-domain experts," *Digital Investigation*, vol. 26, pp. S118 – S126, 2018.

[18] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Using convolutional neural networks for classification of malware represented as images," *Journal of Computer Virology and Hacking Techniques*, Aug 2018. [Online]. Available: https://doi.org/10.1007/s11416-018-0323-0

[19] R. U. Khan, X. Zhang, and R. Kumar, "Analysis of resnet and googlenet models for malware detection," *Journal of Computer Virology and Hacking Techniques*, Aug 2018.

[20] B. Athiwaratkun and J. W. Stokes, "Malware classification with lstm and gru language models and a character-level cnn," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2017, pp. 2482–2486.

[21] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *AI 2016: Advances in Artificial Intelligence*, B. H. Kang and Q. Bai, Eds. Cham: Springer International Publishing, 2016, pp. 137–149.

[22] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas, "An investigation of byte n-gram features for malware classification," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 1–20, Feb 2018. [Online]. Available: https://doi.org/10.1007/s11416-016-0283-1

[23] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, and Y. Elovici, "Unknown malcode detection via text categorization and the imbalance problem," in *2008 IEEE International Conference on Intelligence and Security Informatics*, June 2008, pp. 156–161.

[24] E. Raff and C. Nicholas, "Hash-grams: Faster n-gram features for classification and malware detection," in *Proceedings of the ACM Symposium on Document Engineering 2018*, ser. DocEng '18. New York, NY, USA: ACM, 2018, pp. 22:1–22:4. [Online]. Available: http://doi.acm.org/10.1145/3209280.3229085

[25] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "Mutantx-s: Scalable malware clustering based on static features," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 187–198. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/hu

[26] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," *CoRR*, vol. abs/1803.04173, 2018. [Online]. Available: http://arxiv.org/abs/1803.04173

[27] M. Ahmadi, G. Giacinto, D. Ulyanov, S. Semenov, and M. Trofimov, "Novel feature extraction, selection and fusion for effective malware family classification," *CoRR*, vol. abs/1511.04317, 2015. [Online]. Available: http://arxiv.org/abs/1511.04317

[28] D. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *CoRR*, vol. abs/1511.07289, 2015. [Online]. Available: http://arxiv.org/abs/1511.07289

[29] J.-Y. Kim, S.-J. Bu, and S.-B. Cho, "Malware detection using deep transferred generative adversarial networks," in *Neural Information Processing*, D. Liu, S. Xie, Y. Li, D. Zhao, and E.-S. M. El-Alfy, Eds. Cham: Springer International Publishing, 2017, pp. 556–564.

[30] M. Mays, N. Drabinsky, and S. Brandle, "Feature selection for malware classification," in *Proceedings of the 28th Modern Artificial Intelligence and Cognitive Science Conference 2017, Fort Wayne, IN, USA, April 28-29, 2017.*, 2017, pp. 165–170.

[31] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah, "Wide & deep learning for recommender systems," in *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, ser. DLRS 2016. New York, NY, USA: ACM, 2016, pp. 7–10. [Online]. Available: http://doi.acm.org/10.1145/2988450.2988454

Chapter 5

The Rise of Machine Learning for Detection and Classification of Malware: Research Developments, Trends and Challenges

This article has been published in the Journal of Networks and Computer Applications (SJR: 1.389), belonging to the First Quartile (Q1) as classified by Scimago Journal Rank. See Table 5.1.

Table 5.1: Journal metrics corresponding to the Journal of Networks and Computer Applications for the year 2019.

Journal Metric	Value
Citescore	13.8
Impact Factor	5.570
SNIP	3.154
SJR	1.389

The article *"The Rise of Machine Learning for Detection and Classification of Malware: Research Developments, Trends and Challenges"* [30] provides a systematic and detailed overview of machine learning techniques to tackle the problem of malware detection and classification and in particular, deep learning techniques. The main contributions of this article to the state-of-the-art are the following: (1) it provides a complete description of the methods and features in a traditional machine learning workflow for malware detection and classification; (2) it explores the challenges and limitations of traditional machine learning; (3) it analyzes recent trends and developments in the field with special emphasis on deep learning approaches; (4) it presents the research issues and unsolved challenges of the state-of-the-art techniques; and (5) it discusses new directions of research. This survey aims to help researchers to have an understanding of the malware detection field and of the new developments and directions of research explored by the scientific community to tackle the problem.

Journal of Network and Computer Applications 153 (2020) 102526

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

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

Review

The rise of machine learning for detection and classification of malware: Research developments, trends and challenges

Daniel Gibert [*], Carles Mateu, Jordi Planes

University of Lleida, Jaume II, 69, Lleida, Spain

ARTICLE INFO

Keywords:
Malware detection
Feature engineering
Machine learning
Deep learning
Multimodal learning

ABSTRACT

The struggle between security analysts and malware developers is a never-ending battle with the complexity of malware changing as quickly as innovation grows. Current state-of-the-art research focus on the development and application of machine learning techniques for malware detection due to its ability to keep pace with malware evolution. This survey aims at providing a systematic and detailed overview of machine learning techniques for malware detection and in particular, deep learning techniques. The main contributions of the paper are: (1) it provides a complete description of the methods and features in a traditional machine learning workflow for malware detection and classification, (2) it explores the challenges and limitations of traditional machine learning and (3) it analyzes recent trends and developments in the field with special emphasis on deep learning approaches. Furthermore, (4) it presents the research issues and unsolved challenges of the state-of-the-art techniques and (5) it discusses the new directions of research. The survey helps researchers to have an understanding of the malware detection field and of the new developments and directions of research explored by the scientific community to tackle the problem.

1. Introduction

A brief look at the history of malicious software reminds us that the presence of malware threats has been with us since the dawn of computing. The earliest documented virus appeared during the 1970s. It was known as the Creeper Worm and was an experimental self-replicating program that copied itself to remote systems and displayed the message: "I'm the creeper, catch me if you can". Later, in the early 80s, appeared Elk Cloner, a boot-sector virus that targeted Apply II computers. From these simple beginnings, a massive industry was born and, since then, the fight against malware has never stopped. By the looks of it, this fight turned out to be a never-ending and cyclical arms race: as security analysts and researchers improve their defenses, malware developers continue to innovate, find new infection vectors and enhance their obfuscation techniques. Malware threats continue to expand vertically (i.e. numbers and volumes) and horizontally (i.e. types and functionality) due to the opportunities provided by technological advances. Internet, social networks, smartphones, IoT devices and so on, make it possible for the creation of smart and sophisticated malware. In recent years, ransomware and cryptomining malware emerged as the most prolific types, with Cerber and Locky holding computers all over the globe for ransom while Cryptoloot used the victim's computing power to mine for crypto without their knowledge. Even though malware targeting computer systems still predominates in the ecosystem, mobile and IoT malware is on the rise. According to Symantec (Corporation, 2018), mobile malware variants increased by 54% in 2017 while IoT attacks had a 600% increase, with the Mirai botnet and its variants serving as the vehicle for some of the most potent DDoS attacks in history (Kolias et al., 2017).

To keep up with malware, security analysts and researchers need to constantly improve their cyber-defenses. One essential element is endpoint protection. Endpoint protection provides a suite of security programs including, but not limited to, firewall, URL filtering, email protection, anti-spam and sandboxing. Specifically, anti-malware software provides the last layer of defense. AV engines are responsible for preventing, detecting and removing malicious software installed on the endpoint device. Traditionally, AV solutions relied on signature-based and heuristic-based methods. A signature is an algorithm or hash that uniquely identifies a specific malware while heuristics are a set of rules determined by experts after analyzing the behavior of malware. However, both approaches require the malware to be analyzed prior to the definition of these rules and heuristics. The goal of malware anal-

[*] Corresponding author.
E-mail addresses: daniel.gibert@diei.udl.cat (D. Gibert), carlesm@diei.udl.cat (C. Mateu), jplanes@diei.udl.cat (J. Planes).

https://doi.org/10.1016/j.jnca.2019.102526
Received 1 July 2019; Received in revised form 29 November 2019; Accepted 26 December 2019
Available online 2 January 2020

ysis is to provide information about the characteristics, purpose and behavior of a given piece of software. There are two types of analysis: (1) static analysis and (2) dynamic analysis. On the one hand, static analysis involves examining an executable without execution. On the other hand, dynamic analysis involves examining the behavior of the executable by running it. Both types of analysis have their advantages and limitations and they complement each other. Static analysis is faster but, if malware is successfully concealed using code obfuscation techniques, it could evade detection. Contrarily, code obfuscation techniques and polymorphic malware hardly evades dynamic analysis as it monitors and analyzes the runtime execution of a program. Nevertheless, traditional malware detection and malware analysis are unable to keep pace with new attacks and variants. Organizations are facing the daunting challenge of dealing with millions of attacks a day. In addition, organizations are also experiencing a shortage of cybersecurity skills and talent (on Cybersecurity for the 44th Presidency et al., 2010). The identified issues present a unique opportunity for machine learning to significantly impact and change the cybersecurity landscape due to its ability to handle large volumes of data (Fraley et al., 2017).

During the last decade, machine learning has triggered a radical shift in many sectors, including cybersecurity. There is a general belief among cybersecurity experts that AI-powered antimalware tools will help detect modern malware attacks and improve scanning engines. Evidence of this belief is the number of studies published in the last few years on malware detection techniques that leverage machine learning. According to Google Scholar,[1] the number of research papers published in 2018 is 7720, a 95% increase with respect to 2015 and a 476% increase with respect to 2010. This increase in the number of studies is the result of various factors, including, but not limited to, the increase in public labeled feeds of malware, the increase in computational power as the same time as its reduction in price, and the evolution of the machine learning field, which achieved breakthrough success on a wide range of tasks such as computer vision and speech recognition. Traditional machine learning approaches can be categorized into two primary groups, static and dynamic approaches, depending on the type of analysis. The main difference between them is that static approaches extract features from the static analysis of malware, while dynamic approaches extract features from the dynamic analysis. A third group, defined as hybrid approaches, might be considered. Hybrid approaches combine aspects of both static and dynamic analysis. Furthermore, neural networks have outshone in learning features from raw inputs in various fields. Recent trends in machine learning for cybersecurity are replicating the success of neural networks in the malware domain. For instance, Raff et al. (2018a) and Krčál et al. (2018) proposed building a convolutional neural network to determine the maliciousness of PE executables from the raw bytes of the file itself. The motivation behind neural network approaches is to build detection systems that do not rely on the experts' knowledge of the domain to define discriminative features.

Given the growing impact of AI-powered tools to detect malware, a new literature review is needed considering the recent research studies and exploring the details of traditional static and dynamic approaches. There is some research discussing malware detection methods but we consider it is incomplete. (The reader is referred to Section 2). To complement the papers surveyed and mitigate some flaws in the literature, this paper presents a systematic review on traditional and state-of-the-art machine-learning-powered techniques for malware detection and classification, with special emphasis on the type of information (features) extracted from Portable Executable files. This paper provides the basic background in malware analysis, and a brief description of the process and tools to dissect malware. For a more complete description we refer the reader to Ligh et al. (2010); Sikorski and Honig (2012); Monnappa (2018). This review is intended to support security analysts, who may be interested in applying machine learning to automate part of the malware analysis process, to have a general understanding of the methods currently in use and of the new trends. This paper categorizes methods in three main groups: (i) static methods, (2) dynamic methods and (3) hybrid methods. Furthermore, it provides a detailed description of neural-based methods for detecting and classifying malware, categorized according to how the input is preprocessed before feeding the neural network, as well as a brief description of multimodal learning approaches. This paper closes by discussing the research issues and challenges faced by researchers in the field, including the availability of open and public benchmarks to evaluate the performance of methods, the problem of concept drift in the malware domain, incremental learning, adversarial learning, and the problem of class imbalance.

This survey is organized as follows. Section 2 provides a summary of the surveyed research in the literature. Section 3 describes the background of malware analysis. Section 4 provides a systematic description of static and dynamic methods for malware detection and outlines the most discriminant features for the task at hand. Section 5 presents a detailed overview of the neural-based methods. Section 6 introduces the multimodal and hybrid approaches. In Section 7, a comprehensive analysis of new challenges and the issues of malware detection are discussed. Finally, Section 8 summarizes the concluding remarks of this survey.

2. Related work

This section provides a summary of the surveyed research in the literature and discusses some of its defects. Table 1 sums up the main contributions of the surveys in the literature. We follow by presenting a brief description for each survey, and their flaws that we try to mitigate in our work.

Shabtai et al. (2009) provide a taxonomy for malware detection using machine learning algorithms by reporting some feature types and feature selection techniques used in the literature. They mainly focus on the feature selection techniques (Gain ratio, Fisher score, document frequency, and hierarchical feature selection) and classification algorithms (Artificial Neural Networks, Bayesian Networks, Naïve Bayes, K-Nearest Neighbor, etc). In addition, they review how ensemble algorithms can be used to combine a set of classifiers. Bazrafshan et al. (2013) identify three main methods for detecting malicious software: (1) signature-based methods, (2) heuristic-based methods and behavior-based methods. In addition, they investigate some features for malware detection and discuss concealment techniques used by malware to evade detection. Nonetheless, the aforementioned research does not consider either dynamic or hybrid approaches. Souri et al. (2018) present a survey of malware detection approaches divided into two categories: (1) signature-based methods and (2) behavior-based methods. However, the survey does not provide either a review of the most recent deep learning approaches or a taxonomy of the types of features used in data mining techniques for malware detection and classification. Ucci et al. (2019) categorize methods according to: (i) what is the target task they try to solve, (ii) what are the feature types extracted from Portable Executable files (PEs), and (iii) what machine learning algorithms they use. Although the survey provides a complete description of the feature taxonomy, it does not outline new research trends, especially deep learning and multimodal approaches. Ye et al. (2017) cover traditional machine learning approaches for malware detection, that consists of feature extraction, feature selection and classification steps. However, important features such as the entropy or structural entropy of a file, and some dynamic features such as network activity, opcode and API traces, are missing. In addition, deep learning methods or multimodal approaches for malware detection, which have been hot topics for the last few years, are not covered. Lastly, Razak et al. (2016) provide a bibliometric analysis of malware. It analyzes the publications by country, institution or authors related to malware. Nonetheless, the paper does not provide a description of the features employed by malware

[1] https://scholar.google.es/.

Table 1
List of contributions by the surveyed papers. A ✓tick denotes the information that a survey tries to cover but it does not necessarily provides a thoroughly description of the topic.

Paper	Feature Taxonomy	Static Methods	Dynamic Methods	Hybrid Methods	Multimodal Learning Methods	Deep Learning Methods	Issues and Challenges
Shabtai et al. (2009)	✓	✓	×	×	×	×	×
Bazrafshan et al. (2013)	✓	✓	×	×	×	×	×
Souri et al. (2018)	×	✓	✓	×	×	×	×
Ucci et al. (2019)	✓	✓	✓	×	×	×	✓
Ye et al. (2017)	✓	✓	✓	✓	×	×	✓
Razak et al. (2016)	✓	✓	✓	×	×	×	✓
The present survey	✓	✓	✓	✓	✓	✓	✓

detectors and does not consider the state-of-the-art in the field.

Given the aforementioned limitations in the surveyed papers, the present research presents a systematic review on traditional and state-of-the-art machine learning techniques for malware detection and classification. This paper categorizes traditional methods into two groups: (1) static methods and (2) dynamic methods, categorizing the methods by the type of information or features extracted from Portable Executable files. It extends the surveyed papers by exploring various ways of combining different modalities or types of information, and analyzes state-of-the-art deep learning approaches, which are grouped according to the nature of the raw data fed into the systems. The paper closes with a discussion of the research issues and challenges faced by researches including, but not limited to, the problem of concept drift, adversarial learning and the problem of class imbalance.

3. Background

This section presents an overview of the types of analysis, techniques and tools for dissecting malware targeting the Windows operating system, by far the most used OS worldwide. First, we describe the Portable Executable file format. Then, we provide a description of the fundamental approaches for malware analysis and we give a list of the most common tools utilized for the examination of malicious software. Lastly, we introduce the taxonomy of malware and a brief overview of its evolution.

3.1. The Portable Executable file format

The Portable Executable (PE) format is a file format for executables, object code, DLLs, FON Font files and others used in 32-bit and 64-bit versions of the Windows operating system. The PE32 format stands for Portable Executables of 32-bit while PE32 + stands for Portable Executables of 64-bit format.

Portable Executables encapsulate the information necessary for a Windows operating system to manage the executable code. This includes dynamic library references for linking, API export and import tables, resource management data and threat-local storage data. A PE file consists of a number of headers and sections that tell the dynamic linker how to map the file into memory. See Fig. 1. The PE Header contains information about the executable such as the number of sections, the size of the "PE Optional Header", characteristics of the file, etc.[2] It also contains the import address table (IAT), which is a lookup table used by the application when calling a function in a different module. In addition, a Portable Executable file has various sections that contain the code and data of the executable including, but not limited to, the following:

- The. data section. This section is used to declare initialized data or constants that do not change at runtime.

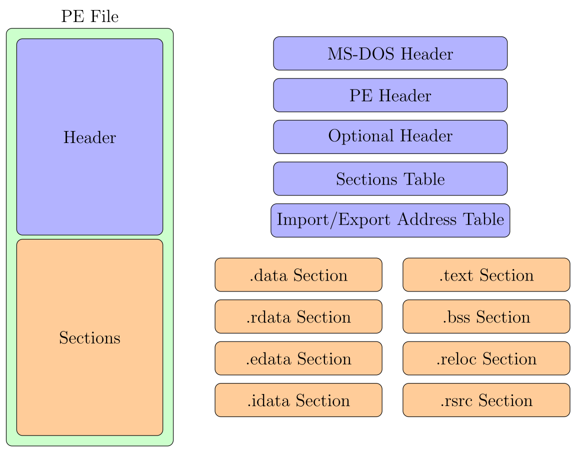


Fig. 1. Portable executable file format.

- The. bss section. This section is used for declaring variables and contains uninitialized data.
- The. text section. This section keeps the actual code of the program.
- The. rsrc section. This section contains all the resources of the program.
- The. rdata section. This section holds the debug directory which stores the type, size and location of various types of debug information stored in the file.
- The. idata section. This section contains information about functions and data that the program imports from DLLs.
- The. edata section. This section contains the list of the functions and data that the PE file exports for other programs.
- The. reloc section. This section holds a table of base relocations. A base relocation is an adjustment to an instruction or initialized variable value that is needed if the loader could not load the file where the linker assumed it would.

More information on the PE file format can be found in the documentation provided by Microsoft.[3]

3.2. Taxonomy of malware

Malicious software, also known as malware, is any kind of software that is specifically designed to disrupt, damage or gain unauthorized access to a computer system or network. Depending on the purposes and proliferation systems, malware can be divided into various, not mutually exclusive categories.

[2] https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files#PE_Header.

[3] https://docs.microsoft.com/en-_us/windows/win32/debug/pe-_format.

- Adware. Malware designed to automatically generate online advertisements. This type of malware generates revenue for its developer by displaying advertisements on the user interface or the screen.
- Backdoor. Computer software that is designed to bypass a system's security mechanism and install itself on a computer to allow the attacker to access it.
- Bot. Software created to automatically perform specific operations such as DDoS attacks or distribute other malware. Bots are part of a botnet, a network of interconnected devices, which are controlled using command and control (C&C) software.
- Downloader. A downloader program's purpose is to download and install additional malicious programs.
- Launcher. A launcher is a computer program designed to stealthily launch other malicious programs.
- Ransomware. Malicious software that restricts user access to the computer system by encrypting the files or locking down the system while demanding a ransom for its release.
- Rootkit. Malware designed to conceal the existence of other malicious programs.
- Spyware. Computer software that spies and collects sensitive information without permission from a victim's computer. Examples include key-loggers, password gravers and sniffers.
- Trojan. A Trojan is a type of malicious software that disguises itself as legitimate software to trick users into downloading and installing malware on their systems.
- Virus. Malicious software that can propagate itself from device to device.
- Worm. A type of virus that exploits vulnerabilities of the operating system to spread. The major difference between worms and viruses is the ability of worms to independently self-replicate and spread while viruses depend on human activity.

3.3. *Malware analysis*

The process of dissecting malware to understand how it works, determine its functionality, origin and potential impact is called malware analysis. With the millions of new malicious programs in the wild, and the mutated versions of previously detected programs, total malware encountered by security analysts has been growing over the past years.[4] Consequently, malware analysis is critical to any business and infrastructure that responds to security incidents.

There are two fundamental approaches to malware analysis: (1) static analysis and (2) dynamic analysis. On the one hand, static analysis involves examining the malware without running it. On the other hand, dynamic analysis involves running the malware. An in-depth description of both approaches is provided in Sections 3.3.1 and 3.3.2.

3.3.1. *Static analysis*

Static analysis consists of examining the code or structure of the executable file without executing it. This kind of analysis can confirm whether a file is malicious, provide information about is functionality and can also be used to produce a simple set of signatures. For instance, the most common method used to uniquely identify a malicious program is hashing. That is, a hashing program produces a unique hash, a sort of fingerprint, that identifies the program. The two most popular hash functions are the Message-Digest Algorithm 5 (MD5) and the Secure Hash Algorithm 1 (SHA-1). The most common static analysis approaches are:

- Finding sequences of characters or strings. Searching through the strings of a program is the simplest way to obtain hints about its functionality. Strings extracted from the binary can contain references to filepaths of files modified or accessed by the executable, URLs to which the program accesses, domain names, IP addresses, attack commands, names of Windows dynamic link libraries (DLLs) loaded, registry keys, and so on. The utility tool *Strings*[5] can be used to search ASCII or Unicode strings ignoring context and formatting in an executable.
- Gathering the linked libraries and functions of an executable, as well as the metadata about the file included in the headers. These data provide information about code libraries and functionalities common to many programs, that programmers link so that they do not need to re-implement a certain functionality. The names of this Windows functions can give us an idea of what the executable does. The utility *Dependency Walker*[6] is a free program for Microsoft Windows used to list the imported and exported functions of a PE file.
- Analyze PE file headers and sections. The PE file headers provide more information than just imports. They contain metadata about the file itself, such as the actual sections of the file. One way to retrieve this information is with the *PEView* tool.[7]
- Searching for packed/encrypted code. Malware writers usually use packing and encryption to make their files more difficult to analyze. Software programs that have been packed or encrypted usually contain very few strings and higher entropy compared to legitimate programs. One way to detect packed files is with the *PEiD* program[8]
- Disassembling the program, i.e. translating machine code into assembly language. This reverse-engineering process loads the executable into a disassembler to discover what the program does. The most relevant software programs for disassembling PE executables are *IDA Pro*,[9] *Radare2*[10] and *Ghidra*.[11]

3.3.2. *Dynamic analysis*

Dynamic analysis involves executing the program and monitoring its behavior on the system. This is typically performed when static analysis has reached a dead end, either due to obfuscation or on having exhausted the available static analysis techniques. Unlike static analysis, it traces the real actions executed by the program. However, the analysis must be run in a safe environment to not expose the system to unnecessary risks, where the system is both the machine running the analysis tool and the rest of the machines on the network. To this end, dedicated physical or virtual machines are set up.

Physical machines must be set up on air-gapped networks, that is isolated networks where machines are disconnected from the Internet or any other network, to prevent malware from spreading. The main downside of physical machines is this scenario with no Internet connection, as many malicious programs depend on Internet connection for updates, command and control and other features.

The second option is to set up virtual machines to perform dynamic analysis. A virtual machine emulates a computer system and provides the functionality of a physical computer. The OS running in the virtual machine is kept isolated from the host OS and thus, malware running on a virtual machine cannot harm the host OS. *VMware Workstation*[12] and *Oracle VM VirtualBox*[13] are some of the virtual machine solutions available to analysts. In addition, there are several all-in-one software products based on sandbox technology that can be used to perform basic dynamic analysis. The most well-known is the *Cuckoo Sandbox*,[14] an

[4] https://www.av-_test.org/en/statistics/malware/.

[5] https://docs.microsoft.com/en-_us/sysinternals/downloads/strings.

[6] http://www.dependencywalker.com/.

[7] http://wjradburn.com/software/PEview.zip.

[8] https://peid.waxoo.com/.

[9] https://www.hex-_rays.com/products/ida/.

[10] https://rada.re/r/.

[11] https://github.com/NationalSecurityAgency/ghidra.

[12] https://www.vmware.com/products/workstation-_pro.html.

[13] https://www.virtualbox.org/.

[14] https://cuckoosandbox.org/.

open source automated malware analysis system. This modular sandbox provides capabilities to trace API calls, analyze network traffic or perform memory analysis. Alternatively, there is a wide list of utilities for dynamically analyze malware and perform advanced and specific monitoring of some functionalities. *Process Monitor*,[15] or procmon, is a tool for Windows that monitors certain registry, file system, network, process and thread activity. *Process Explorer*[16] show the information about which handles and DLL processes are opened or loaded into the operating system. *Regshot*[17] is a registry compare utility that allows snapshots of registries to be taken and compared. *NetCat*[18] is a networking utility that can be used to monitor data transmission over a network. *Wireshark*[19] is an open source sniffer that allows packets to be captured and network traffic to be intercepted and logged. Another indispensable software utility are debuggers. A debugger is used to examine the execution of another program. They provide a dynamic view of a program as it runs. The primary debugger of choice for malware analysts is *OllyDbg*[20] , an ×86 debugger that is free and has many plugins to extend its capabilities.

The risks of using virtualization and sandboxing for malware analysis is that some malware can detect when it is running in a virtual machine or a sandbox and subsequently, they will execute differently than when in a physical machine to make the job of malware analysts harder. In addition, even if you take all possible precautions, some risk is always present when analyzing malware. From time to time, vulnerabilities have been found in the virtualization tools that allow an attacker to exploit some of its features such as the share folders feature.

3.4. Malware evolution

The diversity, sophistication and availability of malicious software pose enormous challenges for securing networks and computer systems from attacks. Malware is constantly evolving and forces security analysts and researchers to keep pace by improving their cyberdefenses. The proliferation of malware increased due to the use of polymorphic and metamorphic techniques used to evade detection and hide its true purpose. Polymorphic malware uses a polymorphic engine to mutate the code while keeping the original functionality intact. Packing and encryption are the two most common ways to hide code. Packers hide the real code of a program through one or more layers of compression. Then, at runtime the unpacking routines restore the original code in memory and execute it. Crypters encrypt and manipulate malware or part of its code, to make it harder for researchers to analyze the program. A crypter contains a stub used to encrypt and decrypt malicious code. Metamorphic malware rewrites its code to an equivalent whenever it is propagated. Malware authors may use multiple transformation techniques including, but not limited to, register renaming, code permutation, code expansion, code shrinking and garbage code insertion. The combination of the aforementioned techniques resulted in rapidly growing malware volumes, making forensic investigations of malware cases time-consuming, costly and more difficult.

Traditional antivirus solutions that relied on signature-based and heuristic/behavioral methods present some problems. A signature is a unique feature or set of features that uniquely distinguishes an executable, like a fingerprint. However, signature-based methods are unable to detect unknown malware variants. To tackle these challenges, security analysts proposed behavior-based detection, which analyzes the file's characteristics and behavior to determine if it is indeed malware, though the scanning and analysis can take some time. To overcome the prior pitfalls of traditional antivirus engines and keep pace with new attacks and variants, researchers started adopting machine learning to complement their solutions, as machine learning is well suited for processing large volumes of data.

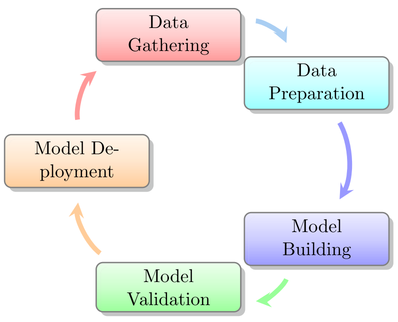


Fig. 2. Machine learning workflow.

4. Traditional machine learning approaches

Over the past decade there has been an increase in the research and deployment of machine learning solutions to tackle the tasks of malware detection and classification. The success and consolidation of machine learning approaches would not have been possible without the confluence of three recent developments:

1. The first development is the increase in labeled feeds of malware meaning that, for the first time, labeled malware is available not only to the security community but also to the research community. The size of these feeds ranges from limited high-quality samples, like the ones provided by Microsoft (Ronen et al., 2018) for the Big Data Innovators Gathering Anti-Malware Prediction Challenge, to huge volumes of malware, such as theZoo (Yuval Nativ, 2015) or VirusShare (2011).
2. The second development is that computational power has increased rapidly and at the same time has become cheaper and closer to the budget of most researchers. Consequently, it allowed researchers to speed-up in the iterative training process and to fit larger and more complex models to the ever increasing data.
3. Third, the machine learning field has evolved at an increased pace during the last decades, achieving breakthrough success in terms of accuracy and scalability on a wide range of tasks, such as computer vision, speech recognition and natural language processing.

In machine learning, a workflow is an iterative process that involves gathering available data, cleaning and preparing the data, building models, validating and deploying into production. See Fig. 2. Instead of dealing with raw malware, the data preparation process of traditional machine learning approaches involves preprocessing the executable to extract a set of features that provide an abstract view of the software. Afterwards the features are used to train a model to solve the task at hand. Because of the variety of malware functionalities, it is important not only to detect malicious software, but also to differentiate between different kinds of malware in order to provide a better understanding of their capabilities. The main difference between machine learning solutions for detection or classification of malware is the output returned by the system implemented. On the one hand, a malware detection system outputs a single value $y = f(x)$, in the range from 0 to 1, which indicates the maliciousness of the executable. On the other hand, a classification system outputs the probability of a given executable belonging to each output class or family, $y \in \mathbb{R}^N$, where N indicates the number of different families.

[15] https://docs.microsoft.com/en-_us/sysinternals/downloads/procmon.
[16] https://docs.microsoft.com/en-_us/sysinternals/downloads/process-_explorer.
[17] https://sourceforge.net/p/regshot/wiki/Home/.
[18] http://netcat.sourceforge.net/.
[19] https://www.wireshark.org/.
[20] http://www.ollydbg.de/.

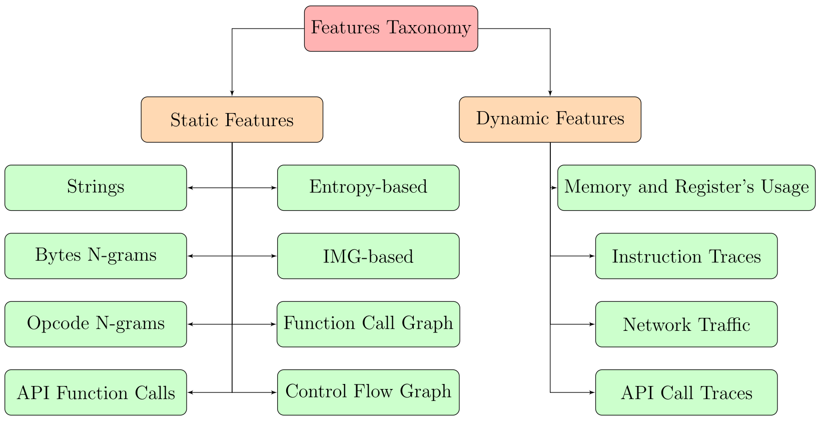


Fig. 3. Taxonomy of features used by traditional M.L. approaches.

A taxonomy of the features is provided in Fig. 3. Accordingly, the types of features can be divided into two groups, like the types of malware analysis approaches: (1) static features and (2) dynamic features. Each feature type individually below.

4.1. *Static features*

Static features are extracted from a piece of program without involving its execution. In Windows Portable Executable files, static features are basically derived from two sources of information, the binary content of the executable or the assembly language source file obtained after decompiling and disassembling the binary executable. On the other hand, in Android applications these features are extracted by disassembling the APK. To extract the assembly language source code of some given software you can use the disassembler tool of your choice. For Windows, you might use IDA Pro or Radare2. Tables 2 and 3 provide a summary of the static methods reviewed. Below you will find a description of each static feature type presented in Fig. 3.

4.1.1. *String analysis*

String analysis refers to the extraction of every printable string within an executable or program. A string refers to a sequence of characters. Searching for strings is the simplest way to obtain clues about the functionality of a program. The information that may be found in these strings can be, for instance, URLs that the program connects to, file locations or filepaths of files accessed/modified by the program, names of the menus of the application, etc. The utility called "Strings" can be used to search an executable for ASCII and Unicode strings, ignoring context and formatting.

Although there are studies using string analysis to detect malware (Konopisky, 2018; Lee et al., 2011), string analysis is commonly employed together with other static or dynamic techniques to reduce its pitfalls. (Ye et al., 2008a). developed a malware detection system based on interpretable strings extracted from both API execution calls and semantic strings reflecting an attacker's intent and goal. The system was composed of a parser to extract interpretable strings for each PE file and a SVM ensemble with bagging to construct the detector. The performance of the system was evaluated on a dataset collected by Kingsoft anti-virus lab.

4.1.2. *Bytes and opcode N-Grams*

The most common type of features for malware detection and classification is n-grams. An n-gram is a contiguous sequence of n items from a given sequence of text. N-grams can be extracted from the bytes sequences representing the malware's binary content and from the assembly language source code. By treating a file as a sequence of bytes, byte n-grams are extracted by looking at the unique combination of every n consecutive bytes as an individual feature. On the other hand, the sequence of assembly language instructions can also be extracted from the assembly language source code. In this case, only the mnemonic of the instruction, i.e. "ADD", "MUL", "PUSH", etc., is retained. Thus, opcode or mnemonic n-grams refer to the unique combination of every n consecutive opcodes as an individual feature.

Moskovitch et al. (2008) presented a method for classifying malware based on text categorization techniques. First, they extracted all n-grams from the training data, with n ranging from 3 to 6. Second, they selected the top 5500 features according to their Document Frequency (DF) score, to which the Fisher Score feature selection technique was later applied. Afterwards, using the resulting features as input they trained various algorithms such as an Artificial Neural Network (ANN), a Support Vector Machine (SVM), Naïve Bayes (NB) and Decision Trees (DT).

Jain and Meena (2011) proposed a method to extract bytes n-gram features, with n ranging from 1 to 8, from known malicious samples to assist in classification of unknown executables. As the number of unique n-grams is extremely large, they used a technique called class-wise document frequency to reduce the feature space. Finally, different N-gram models were prepared using various classifiers like Naïve Bayes, Instance-based Learner, Decision Trees, Adaboost and Random Forests.

Fuyong et al. (2017) proposed a method that calculates the information gain of each bytes n-gram in the training samples and selected K n-grams with the maximum information gain as features. Afterwards, they calculated the averages of each attribute of the feature vectors from the malware and benign samples separately. Lastly, a new piece of software was assigned to one of the two categories according to the similarity between the feature vector of the unknown sample and the average vectors of the two categories.

Santos et al. (2013) proposed a technique for malware detection based on the frequency of appearance of opcode sequences and its relevance. Each program was represented as a vector of features where each feature corresponds to a distinct 1-g or 2-g. To reduce the number of 2-g features, they applied Information Gain to select the top 1000 features. Their approach was validated on 17000 malicious and 1000 benign programs, and results show that the higher accuracy was achieved by a Support Vector Machine classifier with Pearson VII as kernel.

Shabtai et al. (2012) proposed a framework for detecting malware based on opcode n-gram features with n ranging from 1 to 6. They performed a wide set of experiments to: (1) identify the best term representation, whether it is the Term Frequency (TF) or Term Frequency-Inverse Document Frequency, (2) determine the n-gram size, (3) find

Table 2
A side-by-side comparison of the algorithms and feature types of the reviewed static-based methods. Algorithms: Support Vector Machine (SVM), Random Forests (RF), Inference Trees (IT), Recursive Bipartition (RB), Naive Bayes (NB), Artificial Neural Networks (ANN), Decision Trees (DT), Instance-based Learner (IL), K-Nearest Neighbor (K-NN), Logistic Regression (LR), Gradient Boosting (GB), Sequential Minimal Optimization (SMO), Decision Stump (DS), Random Tree (RT), Voted Perceptron (VT).

Paper	Feature Type	Feature Selection, Reduction	Classification Algorithm
Ye et al. (2008a)	Strings	–	SVM ensemble with bagging
Moskovitch et al. (2008)	3, 4, 5, 6-g (bytes)	Fisher Score	ANN, SVM, NB, DT
Jain and Meena (2011)	n-grams (bytes)	Classwise Document Frequency	NB, IL, DT, AdaBoost, RF
Fuyong et al. (2017)	3-g (bytes)	Information Gain	1-NN
Santos et al. (2013)	1,2-g (opcodes)	Information Gain	SVM with Pearson's VII Kernel
Shabtai et al. (2012)	1,2,3,4,5,6-g (opcodes	Term Frequency (TF) and TF-Inverse Document Frequency (TF-IDF)	DT, ANN, LR, RF, BDT; NB, BNB
Hu et al. (2013)	n-grams (opcodes)	hashing trick	agglomerative hierarchical clustering
Yuxin et al. (2019)	n-grams (opcodes)	–	DBN, SVM, K-NN, DT
Sami et al. (2010)	API calls	Fisher Score + Clospan Algorithm	RF
Ye et al. (2008b)	API calls	–	Rule-based Classification System
Ahmadi et al. (2016)	API calls	–	GB
Sorokin and Jun (2011)	Structural Entropy	Discrete Wavelet Transform	Sequence Similarity + Wagner-Fischer Dynamic Programming
Baysa et al. (2013)	Structural Entropy	Discrite Wavelet Transform	Sequence Similarity + Levenhstein distance
Wojnowicz et al. (2016)	Structural Entropy	Haar Discrete Wavelet Transform	Suspiciously Structured Entropic Change Score (SSECS) + LR
Gibert et al. (2018b)	Structural Entropy	Haar Discrete Wavelet Transform	K-NN + Levenhstein distance
Nataraj et al. (2011)	Gray Scale IMG	GIST features	K-NN
Ahmadi et al. (2016)	Gray Scale IMG	Haralick & Local Binary Pattern features	GB
Kancherla et al. (2013)	Gray Scale IMG	Intensity-based, Wavelete-based and Gabor-based features	SVM
Kinable et al. (2011)	Function Call Graph	–	DBSCAN, K-medoids
Hassen and Chan (2017)	Function Call Graph	In-house vector representation algorithm	RF, meta-classifier
Eskandari and Hashemi (2011)	CFG	–	RF, SMO, DS, K-Star, NB, RT
Faruki et al. (2012)	CFG	–	RF, SMO, J-48 DT, NB, VP

Table 3
A side-by-side comparison of the dataset characteristics of the reviewed of the static methods.

Paper	Source	Total Size	Task
Ye et al. (2008a)	Kingsoft lab	39838	Detection
Moskovitch et al. (2008)	VXHeavens, Windows XP	30423	Detection
Jain and Meena (2011)	VXHeavens, Windows XP	2138	Detection
Fuyong et al. (2017)	Open Malware Benchmark, Windows XP, Windows 8	2540	Detection
Santos et al. (2013)	VXHeavens, Windows OS	18000	Detection
Shabtai et al. (2012)	VXHeavens, Windows XP	30423	Detection
Hu et al. (2013)	VXHeavens	132234	Classification
Yuxin et al. (2019)	–	9200	Detection
Sami et al. (2010)	–	34820	Detection
Ye et al. (2008b)	Kingsoft Corporation	29580	Detection
Ahmadi et al. (2016)	Microsoft Malware Classification Challenge	21741	Classification
Sorokin and Jun (2011)	–	–	–
Baysa et al. (2013)	–	–	–
Wojnowicz et al. (2016)	Cylance repository	699121	Detection
Gibert et al. (2018b)	Microsoft Malware Classification Challenge	21741	Classification
Nataraj et al. (2011)	–	9458	Classification
Ahmadi et al. (2016)	Microsoft Malware Classification Challenge	21741	Classification
Kancherla et al. (2013)	Offensive Computing, Windows XP, Windows Vista, Windows 7, Windows NP	27000	Detection
Kinable et al. (2011)	–	1919	Classification
Hassen and Chan (2017)	Microsoft Malware Classification Challenge	21741,	Classification
Eskandari and Hashemi (2011)	APA malware research center	4445	Detection
Faruki et al. (2012)	–	6234	Detection

the optimal K top n-grams and feature selection method, and (4) evaluate the performance of various machine learning algorithms.

Hu et al. (2013) presented MutantX-S, a clustering approach based on opcode N-gram features extracted from the assembly language source code of malware obtained after a disassemble process. MutantX-S improves the scalability on handling very large numbers of malware with high-dimensional features by applying a hashing trick and a close-to-linear clustering algorithm. Instead of working on the large volumes of data, the algorithm performed agglomerative hierarchical clustering only on prototypes.

Alternatively, Yuxin et al., (2019) used a Deep Belief Network (DBN) as an autoencoder to reduce the dimensions of the input feature vectors. As a result, after learning is completed, the last hidden layer of the DBN outputs a new representation or encoding of the N-gram vectors passed as input. By training the DBN with unlabeled data, their classification accuracy outperformed that of the K-Nearest Neighbor, Support Vector Machines and Decision Tree algorithms.

Despite their success at detecting malware, n-gram approaches have some issues that is worth mentioning. First, it is impractical and computationally prohibitive to exhaustively enumerate all n-grams. Estimating model parameters when the number of features is larger than the number of samples might lead to the curse of dimensionality. As a result, feature selection and reduction techniques must be employed. Second, researchers (Raff et al., 2018b) have concluded that byte n-grams appear to be learning mostly from string content in an executable, in particular items from the PE header. As there are millions of potential n-grams (for a larger n), feature selection techniques tend to select as features those that occur frequently enough. This encourages the selection of low entropy features consisting mostly of strings and padding. Third, regardless of what n-grams are learned, we must obtain an exact match when classifying a new sample. Consequently, any minor change will make the feature not occur and, thus, not impact our model. Thus, this lack of generalization is a potential source of over-fitting.

4.1.3. *API function calls*

Application Programming Interfaces (API) and their function calls are regarded as very discriminative features. Literature has shown that API functions invocation might be used to model the program's behavior. Essentially, API functions and system calls are related to services provided by the operating systems such as networking, security, file management, and so on. As there is no other way for software to access the system resources without using API functions, the invocation of particular API functions provides key information to represent the behavior of malware.

Sami et al. (2010) proposed a three-step framework to classify PE files based on API calls usage. First, they analyzed the Portable Executable files and extracted the list of imported API calls. Second, they reduced the feature vector using the Clospan algorithm (Yan et al., 2003). Lastly, the subset of features was used to learn a model using Random Forest.

Ye et al. (2008b) proposed a rule-based system for malware classification. The system consists of three major components: (1) the PE parser, (2) the OOA (Objective-Oriented Association) rule generator and (3) the malware detection module. The PE parser is responsible for parsing the executable and extracting the static execution calls of the corresponding API functions. Then, these calls are used as signatures of the PE files and stored in a signature database. Afterwards, an OOA algorithm is applied to generate class association rules which are stored in the rule database. Lastly, the feature calls and the rules are passed to the malware detection module to determine whether a file is benign or malicious.

Ahmadi et al. (2016) used the frequency of a subset of 794 API function calls, extracted from an analysis on almost 500 K malware samples, to build a multimodal system to classify malware into families. A complete description of their research is provided in Section 6

4.1.4. *Entropy*

Malware authors often employ a variety of obfuscation techniques to hide the malicious purpose of the executables. The two most commonly used are compression and encryption, which are used to conceal malicious segments from static analysis. Consequently, it is of great interest for the information security industry to be able to detect the presence of encrypted or compressed segments of code within executable files. To this end, entropy analysis has been employed because files with segments of code that have been compressed or encrypted tend to have higher entropy than native code. In the context of information theory, the entropy of a bytes sequence reflects its statistical variation. In particular, zero entropy would mean that the same character has been repeated over the analyzed segment. This behavior can be observed in a "padded" chunk of code. On the contrary, a high entropy value would indicate that the chunk consists entirely of distinct values. For instance, Lyda et al. (2007) analyzed a corpus of files consisting of plain text files, native, compressed and encrypted executables, and observed that the average entropy of the executables was 5.09, 6.80 and 7.17, respectively.

As a result, previous research has used a high mean entropy to detect the presence of encryption and compression. However, when the malicious code is concealed in a sophisticated manner it might be hard to detect through such simple entropy statistics. A common approach to reduce the entropy of a file is to pad "nop" instructions. Nevertheless, files with encrypted, compressed, native or padded segments tend to have distinct and unique entropy levels. Thus, researchers (Sorokin and Jun, 2011) started analyzing what is known as the structural entropy of a file, the representation of the malware's byte sequence as a stream of entropy values, where each value indicates the amount of entropy over a small chunk of code in a specific location (see Fig. 4). In particular, Sorokin and Jun (2011) compared the similarity between the structural entropy of an unknown file with that of the training dataset to detect malware.

Baysa et al. (2013) extended the previous work to detect metamorphic malware. They applied wavelet analysis to determine the areas where there are significant changes in the entropy values. Afterwards, they compared the similarity between two files using the Levenshtein distance. Therefore, given an unknown piece of software, it would be classified as the class corresponding to the most similar sample in the training set.

Wojnowicz et al. (2016) developed a method to automatically quantify the extent to which variations in a file's structural entropy make it suspicious. This score is calculated through a two-step process: (1) they computed the wavelet-based energy spectrum of the executable's structural entropy; and (2) they fit various logistic regression models over j-th resolution levels to produce a set of beta coefficients to weight the strength of each resolution energy on the file's probability of being malicious.

4.1.5. *Malware representation as a gray scale image*

An interesting approach for malware visualization was first introduced by

Nataraj et al. (2011) who visualized the malware's binary content as a gray scale image. This is achieved by interpreting every byte as one pixel in an image, where values range from 0 to 255 (0:black, 255:white). Afterwards, the resulting array is reorganized as a 2-D array.

Fig. 5 presents samples from two malware families represented as gray scale images. You can observe that the image representation of samples of a given family is quite similar while distinct from that belonging to a different family. This visual similarity is the result of reusing code to create new binaries. Thus, if old samples are re-used to implement new binaries, the resulting ones would be similar. In most cases, by representing an executable as a gray scale image it would be possible to detect small variations between samples belonging to the same family.

This visual similarity has been exploited by various authors for detecting and classifying malware. In particular, Nataraj et al. (2011) extracted GIST features from the gray scale representation of malware's binary content. Finally, a new executable is classified under one family or another using the K-Nearest Neighbor algorithm (K-NN) with the Euclidean distance as metric. Ahmadi et al. (2016) extracted Haralick and Local Binary Pattern features for classifying malware using boosting tree classifiers.

Kancherla et al. (2013) extracted three sets of features: (1) Intensity-

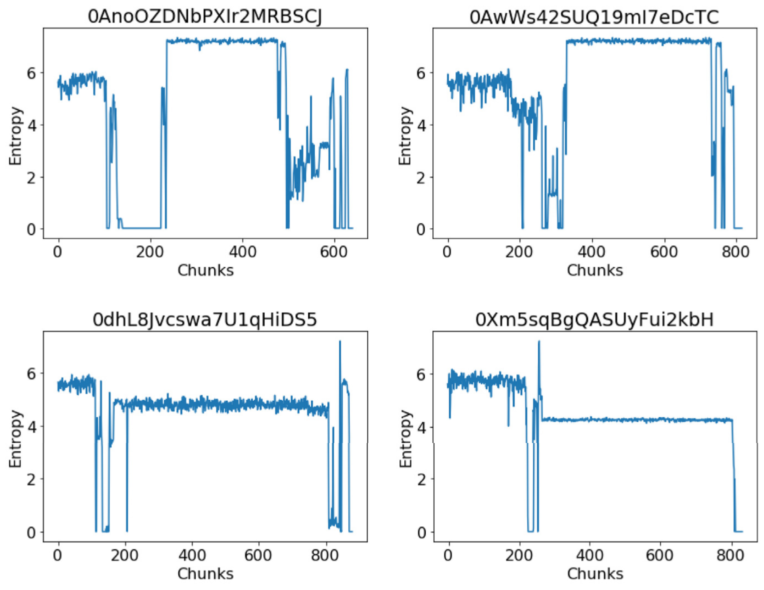


Fig. 4. Structural entropy representation of samples belonging to the Ramnit and Gatak families.

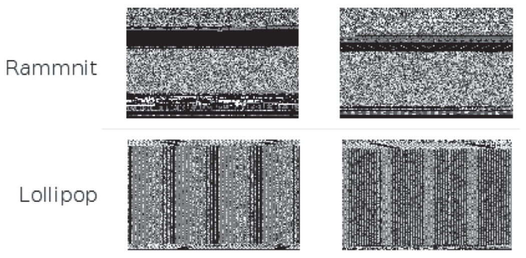


Fig. 5. Gray scale representation of the binary content of malware samples belonging to the Ramnit and the Lollipop families.

based, (2) Wavelet-based and (3) Gabor-based features. In particular, they extracted the average intensity, variance, mode, skewness, kurtosis and the number of pixels with intensity value 0 and 255. Regarding the wavelet-based features, they applied level 3 wavelet decomposition using the Daubechies wavelet, also known as db4, and they obtained one set of approximate coefficients and three sets of detailed coefficients. The features extracted from each of these coefficients were the mean, variance, maximum and minimum values. Finally, to extract the Gabor-based features they applied the Gabor filter (convolution of an input with Gabor function) to the image. The performance of support vector machines as learning algorithm was evaluated on a dataset of 15000 malicious and 12000 benign samples where 70% were used for training and 30% for testing.

The gray scale image representation of software has some drawbacks directly related to how images are generated. Primarily, binaries are not 2-D images and by transforming them as such you introduce unnecessary priors. First, to construct an image you need to select an image width which adds a new hyper-parameter to tune. Notice that selecting the width consequently determines the height on the image depending of the size of the binary. Second, it imposes non-existing spatial correlations between pixels in different rows, which might not be true.

Additionally, like the majority of static features, it suffers from code obfuscation techniques. In particular, techniques like encryption and compression might completely change the bytes structure of a binary program and, thus, methods based on this kind of representation would fail to correctly classify its class. This can be observed in the gray scale representation of samples belonging to the Autorun. K and Yuner. A families from the MalImg dataset (Nataraj et al., 2011), which are almost equal due to both having being compressed with the UPX packer.

4.1.6. Function call graphs

A Function Call Graph (FCG) is a directed graph whose vertices represent the functions of which a software program is composed, and the edges symbolize function calls. A vertex is represented by either one of the following two types of functions:

1. Local functions, implemented by the programmer to perform specific tasks.
2. External functions: provided by the O.S. or system and external libraries.

One particularity of the graph is that only local functions can invoke external functions, not the other way around. Function call graphs are generated from the static analysis of the disassembly file. To extract the FCG of Windows PE executables, IDA Pro or Radare2 can be used.

Kinable et al. (2011) presented an approach to cluster malware based on the structural similarities between function call graphs. They investigated the performance of the k-medoids and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithms for malware clusterization. The comparison among call graphs was computed with pairwise graph similarity scores via graph matching. The experiments were performed on a dataset comprising 194,675 samples from 1050 different malware families.

Hassen and Chan (2017) proposed a method to extract a vector representation of the function call graph based on function clustering. The first module of the system extracts the FCG and labels the vertices with external functions with the function names. The original names of the

Table 4
A side-by-side comparison of the algorithms and input data of the reviewed dynamic-based methods. Algorithms: Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM).

Paper	Input	Classification Algorithm	Feature Selection, Reduction Techniques
Ghiasi et al. (2012)	Register's Usage	Matching based on Registers Values Set Analysis	–
Ghiasi et al. (2015)	Register's Usage	Matching based on Jaccard's similarity distance on dynamic VSA representations	Prototype Extraction
Carlin et al. (2017a)	Instruction Traces	RF, Hidden Markov models	Opcode counts
O'kane et al. (2016)	Instruction Traces	SVM	PCA
Anderson et al. (2011)	Instruction Traces	SVM	–
Storlie et al. (2014)	Instruction Traces	flexible spline logistic regression	–
Bekerman et al. (2015)	Network Traffic	Naïve Bayes, J48 DT, RF	Correlation Feature Selection Algorithm
Zhao et al. (2015)	DNS and Network Traffic	Reputation Engine	–
Kheir (2013)	Network Traffic (HTTP traffic)	High-level clustering, fine clustering and Incremental K-means clustering	.
Boukhtouta et al. (2016)	Network Traffic	Boosted J48, J48, NB, Boosted NB, SVM, HMMs	–
Perdisci and Wenke Lee (2015)	HTTPs Traffic	Coarse-grain Clustering, Fine-grain Clustering, Cluster-merging	–
Galal et al. (2016)	API Call Traces	DT, RF, SVM	Hand-crafted Heuristics
Ding et al. (2013)	API Call Traces	Object Oriented Association Mining	Document Frequency, Information Gain
Salehi et al. (2017)	API Call Traces	RF, J48 DT, Bayesian Logistics Regression, Sequential Minimal Optimization	Fisher Score, SVM based on Recursive Feature Elimination
Rieck et al. (2011)	API Call Traces	Hierarchical Clustering	–
Uppal et al. (2014)	API Call Traces	NB, RF, DT, SVM	odds ratio

internal functions are not preserved and instead, each internal function is represented as the sequence of instructions that the function implements. Afterwards, the resulting FCG is passed to the next module to cluster the local functions and relabel them with their cluster-id. Finally, the graph representation is converted into a feature vector using function clustering based on the Minhash signatures of the functions.

4.1.7. Control Flow Graph

A Control Flow Graph (CFG) is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths. A basic block is a linear sequence of program instructions having an entry point (the first instruction executed) and an exit point (the last instruction executed). A CFG is a representation of all the paths that can be traversed during a program's execution.

Eskandari and Hashemi (2011) presented an approach to detect metamorphic malware through their Control Flow Graphs. The system consists of three components. First, the PE file is disassembled. Second, a preprocessing algorithm is applied to assembly files to generate a CFG including the API calls. Then, the resulting sparse graph is converted to a vector representation. Third, the system labels the CFG using a classification algorithm. The performance of the system was evaluated on 2140 and 2305 benign and malicious PE executables, respectively, and the best results were achieved by a Random Forest classifier, with 97% accuracy.

Faruki et al. (2012) proposed an approach to generate API calls n-grams to detect malicious code from malware's CFG. In their work, the abstraction of the executable is represented by the API calls made. These API calls are later converted into feature vectors using n-gram analysis with n ranging from 1 to 4. Afterwards, classification is performed with various algorithms, including Random Forest, Sequential Mining Optimization, J-48 Decision Tree, Naïve Bayes and Voted Perceptron. The best results were achieved by the Random Forest classifier with the API 4-g feature vector as input.

4.2. Dynamic features

Dynamic features are those extracted from the execution of malware at runtime. Dynamic analysis involves monitoring malware (and observing the real sequence of instructions executed or the sequence of API functions triggered) as it runs or examining the system after the malware has executed. It reveals process creation, file and registry manipulation and modifications of memory values, registers and variables. Tables 4 and 5 compile the dynamic approaches reviewed. A description of the most common information and features extracted through dynamic analysis is provided below. Approaches are grouped in four groups depending on their input data. Section 4.2.1 presents methods that extract features from malware's memory, registers and CPU usage. Section 4.2.2 includes approaches that extract features from the runtime traces of executables. Section 4.2.3 summarizes methods that extract features from the network activity of malware. Finally, Section 4.2.4 presents methods that process the API call traces of malware.

4.2.1. Memory and Register's usage

The behavior of a computer program can be represented by the values of the memory contents at runtime. In other words, values stored in different registers while a computer program is running can distinguish benign from malicious programs.

Ghiasi et al. (2012) proposed a method based on similarities of malware behaviors. First, they monitored the runtime behavior of malware and stored the register values for each API call hooked, before and after the API was invoked. Subsequently, they traced the distribution and changes of register values and created a vector for each of the values of the EAX, EBX, EDX, EDI, ESI and EBP registers. In the matching phase, a similarity score was computed between a new file and the whole training files. Then, the new file was set to the label of the file in the training set that had the highest similarity score.

Ghiasi et al. (2015) proposed a method to find similarities of runtime behaviors based on the assumption that binary behaviors affect register values differently. In their work, the runtime behavior is recorded and some API calls from common DLLs are hooked. The system analyzes memory contents and register values to build a similarity score between two files. When a new file is entered into the system, the highest similarity score between this file and the prototypes is calculated. Prototypes are small sets of files that are representative samples of the whole dataset, which provide an acceptable approximation in pair-wise distance analysis. Afterwards, the two files are similar if they achieve the minimum threshold of similarity.

Table 5
A side-by-side comparison of the dataset characteristics of the reviewed dynamic methods.

Paper	Source	Total Size	Task
Ghiasi et al. (2012)	–	1211	Detection
Ghiasi et al. (2015)	–	1240	Detection
Carlin et al. (2017a)	VirusShare	1000000	Detection
O'kane et al. (2016)	–	750	Detection
Anderson et al. (2011)	Windows XP	2230	Detection
Storlie et al. (2014)	Offensive Computing repository	21988	Detection
Bekerman et al. (2015)	Vering, Emerging Threats	50720 (records)	Classification
Zhao et al. (2015)	Alexa's TOP 1000 sites	4000000 (records)	Detection
Kheir (2013)	AV Company	100000	Detection
Boukhtouta et al. (2016)	–	–	Detection, Classification
Perdisci and Wenke Lee (2015)	–	–	Detection, Classification
Galal et al. (2016)	VirusSign, Windows7	4000	Detection
Ding et al. (2013)	–	8170	Detection
Salehi et al. (2017) Windows XP	Sami et al. (2010)	4368	Detection
Rieck et al. (2011)	Sunblet Software	33698	Classification
Uppal et al. (2014)	VXHeavens	270	Detection

4.2.2. Instruction traces

A dynamic instruction trace is a sequence of processor instructions called during the execution of a program. Contrary to the static instruction trace, dynamic traces are ordered as they are executed while static traces are ordered as they appear in the binary file. Dynamic traces are a more robust measure of the program's behavior, since code packers and encrypters can obfuscate and hinder the code instructions from static analysis.

Carlin et al. (2017a) presented an approach that performs dynamic analysis on virtual machines to extract program runtime traces from both benign and malicious executables. They analyzed the sequence of opcodes executed to detect malware by testing two algorithms: (1) a Random Forest classifier to classify all count-based data and (2) a Hidden Markov model to classify data based on temporal relations in the opcode sequences. Carlin etal., 2017b, instead of building a classification system based on opcode counts, performed n-gram analysis, where $n = 1 \ldots 3$, to enhance the feature set. Their approach detected malware with 99.01% accuracy using sequences of up to 32 K opcodes.

O'kane et al. (2016) analyzed malicious runtime traces to determine (1) the optimal set of opcodes necessary to build a robust indicator of maliciousness in software, and to determine (2) the optimal duration of the program's execution to accurately classify benign and malicious software. The proposed approach used a Support Vector Machine on the opcode density histograms extracted during the program's execution to detect malware.

Anderson et al. (2011) introduced a malware detection method based on the analysis of graphs constructed using the instruction traces collected from the execution of the target executable. These graphs represent Markov chains, where the vertices are the instructions and the transition probabilities were estimated by the data contained in the trace. A combination of graph kernels, including the Gaussian kernel and the spectral kernel, was used to calculate the similarity matrix between the instruction trace graphs. Finally, the resulting similarity matrix is fed to a support vector machine to perform classification.

Storlie et al. (2014) presented a malware detection system based on the analysis of dynamically collected instruction traces. Instruction traces were collected from the execution of malware in a sandbox environment, a modified version of the Ether malware analysis framework (Dinaburg et al., 2008). Each instruction trace was represented with a Markov chain structure in which each transition matrix P has rows modeled as Dirichlet vector. Afterwards, the maliciousness of the program was determined using a flexible spline logistic regression model.

4.2.3. Network traffic

Detecting malicious traffic on a network can uniquely provide specific insights into the behavior of malicious programs. As soon as malware infects a host machine, it may establish communication with an external server to obtain the commands to execute on the victim or to download updates, other malware or to leak private and sensitive information of the user/device. As a result, the monitoring of network traffic entering and exiting the network, the traffic within the network and the host activity, provide helpful information to detect malicious behavior. Approaches in the literature extract events at several abstraction levels, from raw packets to network flows, detailed protocol decoding such as HTTP and DNS requests, to host-based events and metadata such as IP addresses, ports and packet counts.

Bekerman et al. (2015) presented a system for detecting malware by analyzing network traffic. In their work, they extracted 972 behavioral features from analyzing the network traffic on the Internet, Transport and Application layers. Afterwards, a subset of the features was selected using the Correlation Feature Selection Algorithm (Hall, 1999). Then, the resulting features were used to test three different classification algorithms, including Naïve Bayes, Decision Tree (J48) and Random Forest.

Zhao et al. (2015) proposed a system to detect APT malware infections based on both malicious DNS and traffic analysis. The system consists of two main components: (1) the malicious DNS detector and (2) the network traffic analyzer. On the one hand, the malicious DNS detector extracts 14 features indicative of APT malware and C&C domains. On the other hand, the network traffic analyzer combines a signature-based system and an anomalous-based system which detect infections based on the accuracy of the rules from the VRT Rule sets (Snort, 2015) of Snort, and anomalies occurring on the Protocol and Application level, respectively. Afterwards, a J48 Decision Tree classifies the threat.

Kheir (2013) presented a systematic approach to build detection signatures based on user agent anomalies within malware HTTP traffic. First, they extracted user agent header fields within HTTP traffic. Then, they performed an initial high-level clustering step to group user agents which are likely to have similar patterns. Afterwards, they applied a second clustering step to each group of user agent to group together those agents that can be described with a common set of signatures. Lastly, incremental K-means clustering was applied to regroup user agents that share similar pattern sequences in the same clusters. Then, the token-subsequence algorithm further extracted these shared patterns and built lists of token sequences that were translated into signatures that applied at either the network or the application layer using web proxies.

Boukhtouta et al. (2016) proposed a malware detection and classification system based on DPI and flow packed headers. Their approach executed malware in a sandbox for 3 min to generate representative malicious traffic. Then, bidirectional flow features were extracted from the traffic such as the number of forward and backward packets, the maximum and minimum inter-arrival times for forward packets, the packet size, etc. The resulting features were provided as input to the following classification algorithms: Boosted J48, J48, Naïve Bayes, Boosted Naïve Bayes and SVMs, which detected whether or not the traffic was malicious. Once the traffic had been defined as malicious, Hidden Markov Models created non-deterministic models that profiled malware families using unidirectional flows represented as a set of 45 features including the total number of packets, the median, mean and first-quartile of inter-arrival times, etc.

Perdisci and Wenke Lee (2015) proposed a method to perform behavioral clustering of malware based on the HTTPs traffic obtained from monitoring the executables in a controlled environment. The method recorded the sequences of HTTP requests performed by malware and used this information to cluster malware using at least one of the following clustering algorithms: coarse-grain clustering, fine-grain clustering and cluster-merging. Finally, network signatures were extracted for each cluster and used to identify infected computers.

4.2.4. API call traces

Software programmers use the Windows API to access basic resources available to a Windows system including, but not limited to, file systems, devices, processes, threads and error handling, and also to access functions beyond the kernel such as the Windows registry, start/stop/create a Windows service, manage user accounts and so on. Consequently, the Windows API call traces have been used in the literature to capture the behavior of malicious applications.

Galal et al. (2016) presented an approach to process raw information gathered by API call hooking to produce a set of actions representing the malicious behaviors of malware. An action was a representative semantic feature inferred from the sequences of API calls using a set of heuristic functions. Afterwards, the viability of actions was assessed by various classification algorithms such as Decision Trees, Random Forests and Support Vector Machines.

Ding et al. (2013) proposed an API (Application Programming Interface)-based association mining method for malware detection. To increase the detection speed of the objective-oriented association (OOA) mining, they improved the rule quality, changed the criteria for API selection to remove APIs that cannot become frequent items, find association rules with the strongest discriminant power, among others. These strategies improved the running speed of their approach by 32% and 15% of the time cost for data mining and classification, respectively.

Salehi et al. (2017) proposed a dynamic method to detect malicious activity in Android APKs based on the arguments and return values of API calls. They developed an "in-house" tool consisting of a virtual machine, a hooking tool and a logging system, which was used to analyze the binary files and monitor their behavior. Their approach is based on the hypothesis that API names alone may not represent intent of the operations that the function performs. For this reason, the feature set modeling malicious and benign behaviors was constructed using the API calls, their input arguments and return values. Afterwards, the feature set was reduced through a two-stage process. In the first stage, the Fisher score was applied to select the most discriminative features. In the second stage, Support Vector Machine based on Recursive Feature Elimination reduced the feature set even more. Then, the generated feature set was used as input to the classification algorithms.

Rieck et al. (2011) developed a framework for the automatic analysis of malware behavior using clustering techniques. The framework automatically identifies novel classes of malware with similar behavior and assigns unknown malware to these discovered classes. Malware is monitored in a sandbox and the API calls are inspected at runtime. Each execution of a binary is represented as a sequential report of MIST instructions. This information is embedded in a vector space using q-grams. Afterwards, the embedded reports are clustered using prototypes. Hierarchical clustering was employed to determine groups of malware behavior. For classification, the algorithm determines the nearest prototype of the training data.

Uppal et al. (2014) presented a malware identification approach based on features from the API sequences. The method monitors the execution of a binary to keep track of the API calls invoked. Then, API call grams are generated and the odds ratio of each gram is calculated. This odds ratio is used to rank the features and select the leading n features to form the feature vector. For classification, various algorithms were proposed including Naïve Bayes, Random Forest, Decision Tree and Support Vector Machine. The evaluation of their approach was performed on a dataset on 270 binaries obtained from VXHeavens.

5. Deep learning approaches

The above traditional machine learning approaches (see Section 4) rely mainly on manually designed features based on expert knowledge of the domain. These solutions provide an abstract view of malware that a machine learning classifier, e.g. Neural Network, Decision Tree, Support Vector Machine, etc, uses to make a decision. Feature engineering and feature extraction are key, time-consuming processes of the machine learning workflow. Following recent trends in computer vision and natural language processing fields, the development of M.L. solutions for malware detection has started heading towards deep learning architectures. These solutions have replaced the aforementioned feature engineering process of the M.L. workflow with a fully trainable system beginning from raw input to the final output of recognized objects.

Deep learning approaches for tackling the problem of malware detection and classification can be classified into various groups depending on how the input is preprocessed before feeding the learning algorithm. Tables 6 and 7 present a summary of recent developments. A detailed description of the distinct groups and methods is provided below.

5.1. Feature vector representation

The methods corresponding to this category perform feature engineering to extract a set of features which provide an abstract representation of an executable. Then, the resulting feature vector is fed as input to a feed-forward Neural Network. Notice that the feature vectors extracted by the methods presented in Sections 4.1 and 4.2 can also be used to train feed-forward networks.

Saxe et al. (2015) introduced a malware detection system, powered by a deep neural network, consisting of three main components: (1) the feature extraction component extracts 4 different types of features, byte/entropy histogram features, PE import features, String 2D histogram features, and PE metadata features; (2) the second component consists of the deep neural network classifier; and (3) the third component is the score calibrator, which calibrates the final score. Their system was evaluated on a dataset of 431926 executables retrieved from the Invencea database and achieved a detection rate of 95%.

Huang and Stokes (2016) proposed a multi-task deep learning architecture for malware detection and classification. They extracted a combined feature set consisting of null-terminated tokens, API event plus parameter value, and API trigrams from static and dynamic analysis. Due to the high dimensionality of the input space, mutual information was performed to generate features that best characterize each class. Afterwards, the resulting feature vector was reduced to 50000 features using random projections. Finally, a deep feed-forward Neural Network was trained using the projected feature vector.

Dahl et al. (2013) investigated a malware classification architecture which projects a high-dimensional feature vector to a much lower dimension using random projections. More specifically, random projections reduced the dimensionality of the feature vector from 179000 to 4000 features. Afterwards, a Neural Network classifier learned a non-linear model to classify malware. The system was evaluated on a dataset of 2.6 million labeled samples and achieved an error rate of 0.49%.

Table 6
A side-by-side comparison of the algorithms and input data of the reviewed deep learning methods. Algorithms: Convolutional Neural Network (CNN), Residual Network (ResNet), Autoencoder (AE), Recurrent Neural Network (RNN), Long Short-Term Memory Network (LSTM), Gated Redurrent Unit Network (GRU), Neural Network (NN).

Paper	Feature Type	Classification Algorithm
Saxe et al. (2015)	byte/entropy histogram features	Feed-forward network
	PE import features	
	String 2D histogram feature,	
	PE metadata features,	
Huang and Stokes (2016)	Sequence of API calls events	Feed-forward network
	Sequence of NULL-terminated objects	
	API 3-g	
Dahl et al. (2013)	NULL-terminated patterns	Feed-forward network
	API 3-g	
	API 1-g	
Gibert et al. (2018c)	gray-scale image	CNN
Rezende et al. (2017)	gray-scale image	ResNet-50
Raff et al. (2018a)	bytes sequence	CNN
Krčál et al. (2018)	bytes sequence	CNN
Gibert et al. (2018a)	bytes sequence	Denoising AE + ResNet
Davis et al. (2017)	bytes sequence	CNN + RNN
Gibert et al. (2018b)	structural entropy	CNN
Athiwaratkun et al. (2017)	API call sequence	LSTM, GRU
Kolosnjaji et al. (2016)	API call sequence	CRNN
Gibert et al. (2017)	mnemonics sequence	Shallow CNN
Gibert et al. (2019)	mnemonics sequence	Hierarchical CNN
Prasse et al. (2017)	HTTP traffic	LSTM
AL-Hawawreh et al. (2018)	Network behavior	AE + NN

Table 7
A side-by-side comparison of the dataset characteristics of the reviewed deep learning methods.

Paper	Source	Total Size	Task
Saxe et al. (2015)	Invencea's private malware database	431.926	Detection
Huang and Stokes (2016)	In-house dataset	6.500.000	Detection, Classification
Dahl et al. (2013)	In-house dataset	2.600.000	Detection, Classification
Gibert et al. (2018c)	DB A: MalIMG	DB A: 9339	Classification
	DB B: Microsoft Malware Classification Challenge	DB B: 21741	
Rezende et al. (2017)	MalIMG dataset	9339	Classification
Raff et al. (2018a)	In-house dataset	2.011.786	Detection
Krčál et al. (2018)	AVAST's repository	20.000.000	Detection
Gibert et al. (2018a)	Microsoft Malware Classification Challenge	21.741	Classification
Davis et al. (2017)	–	–	Detection, Classification
Gibert et al. (2018b)	Microsoft Malware Classification Challenge	21.741	Classification
Athiwaratkun et al. (2017)	In-house dataset	75.000	Detection
Kolosnjaji et al. (2016)	VirusShare	–	Detection
	Maltrieve private collection		
Gibert et al. (2017)	Microsoft Malware Classification Challenge	21741	Classification
Gibert et al. (2019)	Microsoft Malware Classification Challenge	21741	Classification
Prasse et al. (2017)	In-house datasets	DB A: 44.348.879	Detection
		DB B: 129.005.149	
AL-Hawawreh et al. (2018)	DB A: KDD Cup 99	DB A: 148.517	Detection
	DB B: UNSW-NB15	DB B: 257.673	

5.2. IMG-based representation

Deep learning IMG-based approaches take as input the gray scale image representation of malware's binary content already described in Section 4.1.5. Instead of relying on hand-engineered feature extractors to gather relevant information about the gray scale image, they feed the images into a Convolutional Neural Network architecture that perform both feature learning and classification.

Gibert et al. (2018c) proposed a Convolutional Neural Network architecture composed of three convolutional blocks followed by one fully-connected and the output layer. Each convolutional block consisted of a convolutional operation, the ReLU activation, max-pooling and normalization. The convolutional layers acted as detection filters for the presence of specific features or patterns in the data and the subsequently fully-connected layers combine the learned features and determine a specific target output. Their approach was evaluated on the Microsoft Malware Classification Challenge (Ronen et al., 2018) against hand-crafted feature extractors (Nataraj et al., 2011; Kancherla et al., 2013; Ahmadi et al., 2016) and results demonstrate the superior performance of a deep learning architecture for classifying malware represented as gray scale images. Similarly, Rezende et al. (2017) proposed to use the ResNet-50 architecture with pretrained weights to classify malware images obtained from the MalImg dataset (Nataraj et al., 2011).

5.3. API call traces

Section 4.1.3 presented approaches that used as input a feature vector where each position of the vector indicated whether a particular API function was invoked by the program. However, this kind of feature representation does not take into account the order in which the API functions had been invoked. Alternatively, one can collect the ordered sequence of API functions invoked and use this information to build classifiers that capture the dependencies in the API function traces.

Athiwaratkun et al. (2017) examined recurrent neural network architectures to better capture long-term dependencies in API call traces. They experimented with the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) as language models. Their proposed method is composed of two stages. In the first stage, LSTM or GRU are used to construct the features associated with a particular API call trace. In the second stage, these features are classified with either a single fully-connected layer or Logistic Regression with softmax. In addition, they also proposed a character-level convolutional neural network (Zhang et al., 2015). This network takes as input a sequence of 1014 characters maximum length where each character is an event. Sequences with fewer than 1014 characters were padded in the end with end-of-sequence tokens. The character-level network presented consists of 9 layers, 6 convolutional and 3 fully-connected.

Kolosnjaji et al. (2016) investigated the utilization of neural networks to improve the classification of newly retrieved malware samples into a predefined set of malware families. They analyzed two types of neural network layers for modeling system call sequences: convolutional and recurrent layers. They constructed a Neural Network based on convolutional and recurrent layers that combines the convolution of n-grams with full sequential modeling. The input of the network is the API call sequences of malware without API calls repeated more than two times in a row. Each API call was encoded using one-hot encoding to find a unique vector for every API call. The convolutional part of the network consists of a convolutional layer followed by pooling with the convolution acting as feature extractor. The outputs of the convolutional part are connected to the recurrent part which models sequential dependencies in the kernel API traces. To extract the features of highest importance from the LSTM output, mean-pooling was used. Furthermore, they applied dropout to prevent overfitting and a softmax layer to output the class probabilities.

5.4. Instruction traces

Similarly, a program can be modeled as a sequence of instructions executed by the processor.

These sequences of instructions can be obtained from both static and dynamic analysis. On the one hand, it might be possible to obtain them by disassembling the binary executable and processing the resulting disassembled file. On the other hand, the executable can be monitored during runtime and extract the complete sequence of instructions executed on the system. These sequences of instructions can be used to train an end-to-end system to jointly learn the appropriate features and perform classification without having to explicitly enumerate millions of n-grams during training.

Gibert et al. (2017) proposed a Neural Network architecture with an embedding layer, one convolutional layer followed by a max-pooling and an output layer. The convolutional layer could intrinsically learn to detect n-gram-like signatures by learning to detect subsequences of opcodes that are indicative of malware. In addition, depending on the size of the kernel, the convolutional layer allows detecting very long n-gram-like signatures which would be impractical if explicit enumeration of all n-grams were required. This is achieved by defining filters of various sizes. For instance, in their work the convolutional layer contained 64 filters of size $h\,x\,k$ for every $h \in \{2,3,4,5,6,7\}$, where k refers to the size of the embedding vector. Then, the maximum value was taken as the feature corresponding to the filter by applying the max-pooling operator over the feature map (also known as global max-pooling). This permits to extract n-gram-like signatures with n ranging from 2 to 7. Finally, the softmax layer outputs the probability distribution over the classes.

Alternatively, Gibert et al. (2019) proposed a Hierarchical Convolutional Neural Network (HCNN) to deal with the hierarchical structure of PE executables. In their work, instead of representing malware as a sequence of instructions, they grouped instructions in the same function to keep the hierarchical structure of a computer program. In consequence, the assembly language instructions were split into functions, where each function was represented by a sequence of mnemonics. In consequence, the hierarchical convolutional neural network captured features at the mnemonic-level and at the function-level.

5.5. Bytes-based representation

The simplest way to represent a computer program is as a sequence of bytes. In other words, each byte is treated as a unit in an input sequence. The main advantage of this representation is that it could be used to represent malware indistinctly of the O.S. and hardware because it is not affected by the file format of the executable, whether it is a Portable Executable (PE) file, or an Executable and Linkable Format (ELF) file, etc. However, representing an executable as a sequence of bytes presents considerable challenges not found in other domains. First, by treating each byte as a unit in a sequence, the size of the resulting byte sequences could consist of several million time steps, making it among the most challenging sequence classification problems. Second, the meaning of any particular byte depends on its context and could encode any type of information such as binary code, human-readable text, images, sound, etc. Third, binary files exhibit various levels of spatial correlation. Adjacent machine instructions tend to be correlated spatially, but, due to jumps and function calls, this correlation might not always hold, as they transfer the control of the program into other addresses in memory and the execution continues from there. Consequently, these discontinuities are maintained on the binary file and in its hexadecimal representation. Therefore, when designing a model to detect malware from a sequence of bytes, (1) its ability to scale well with sequence length and (2) its ability to consider both local and global context while examining an entire file must be taken into account.

Raff et al. (2018a) proposed a Convolutional Neural Network architecture to capture such high level location invariance. They combined the convolutional activations with a global max-pooling before the fully connected layer to allow the model to produce its activations regardless of the location of the detected features in the bytes sequence. Rather than performing convolutions on the raw byte values, they used an embedding layer to map each byte to a fixed length feature vector.

Krčál et al. (2018) explored a deeper architecture composed of an embedding layer followed by four convolutions with strides separated by a max-pooling layer between the second and third convolutional layers, followed by global average pooling and four fully connected layers. They evaluated their model against the MalConv architecture and observed that they slightly increased the performance of the MalConv in their dataset from 94.6% to 96.0% of accuracy. In addition, they enriched the feature vector obtained after the global pooling with hand-crafted features to build a stronger classifier.

As part of an analysis of the likelihood that a given input includes malicious code, an executable can be divided into chunks of code. Afterwards, the information at each chunk can be encoded or codified as a single value. Thus, the resulting output would be a time series $m = \{m_1, m_2, \ldots, m_n\}$, where m_i is the corresponding codification of the i-th chunk and n is the number of chunks into which a binary has been divided. Gibert et al. (2018b) proposed a method for classifying malware represented as a stream of entropy values using Convolutional Neural Networks. Thus, they calculated the entropy of each chunk of code. Afterwards, they applied the single-level discrete wavelet transform to the entropy time series to compress the signal and reduce the noise. The wavelet transformation generated two time series, the approximation coefficients and the details coefficients. Then, both time series were fed into a Convolutional Neural Network that performs feature learning on both time series and classifies a given malware sample into its corresponding family.

Gibert et al. (2018a) encoded the information stored by each chunk using Denoising Autoencoders (DAE). In their work, they first divided a binary file into contiguous, non-overlapping chunks of fixed size. Afterwards, a denoising autoencoder takes as input every chunk of bytes and projects it into a single value that captures the main factors of variation in the data. The resulting time series is then fed into a Dilated Residual Network which learns descriptive patterns from the encoding of the bytes sequence and assigns a label indicating the family to which the malware belongs.

In Cylance's patent (Davis et al., 2017), instead of codifying the information at a particular chunk into a single value, they implemented a computer method to detect malicious code that comprises three phases:(1) the examination of a sequence of chunks with a Convolutional Neural Network, (2) an analysis of at least some of the chunks using a Recurrent Neural Network (RNN), and (3) determining the likelihood that the input includes malicious code, based on at least some of the chunks analyzed using the RNN.

5.5.1. *Network traffic*

The methods that fall under this category are those that aim to classify network traffic. More specifically, they try to detect malicious traffic by identifying the type and quantity of traffic flowing through a network.

Prasse et al. (2017) proposed a framework to detect malware on client computers based on the analysis of HTTP traffic. They extracted various features from the sequences of flows sent or received by client computers and domain-name features. Then, an LSTM classifier takes sequences of flows as input and learns to determine whether or not the flows originate from malicious applications.

AL-Hawawreh et al. (2018) proposed an anomaly detection technique for detecting intrusions in Internet Industrial Control Systems (IICSs) based on deep learning models. The system includes an unsupervised learning phase, where a Deep Autoencoder learns normal network behaviors, and a supervised learning phase, where a Deep Neural Network uses the estimated parameters of the Autoencoder to fine-tune its parameters and classify incoming network observations.

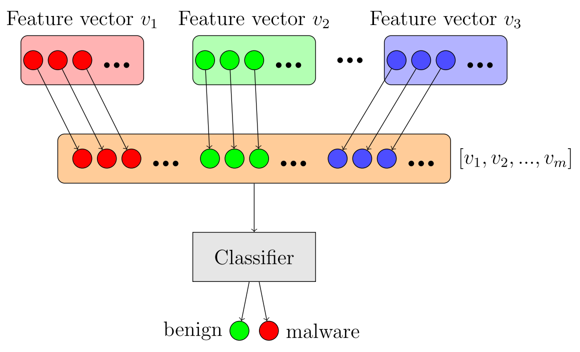


Fig. 6. Early fusion strategy.

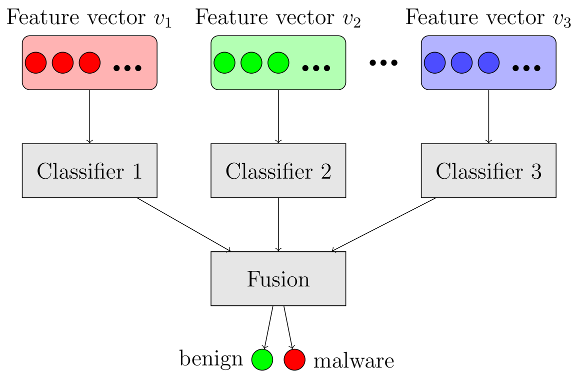


Fig. 7. Late fusion strategy.

6. Multimodal approaches

So far, we have presented approaches that largely rely on one type of feature or modality of data to detect and classify malware. However, malware detection is a research problem characterized as multimodal as it includes multiple modalities of data. Multimodal learning is the field that studies how to be able to interpret such multimodal signals together. Though combining different modalities or types of information for improving performance seems an intuitively appealing task, it is very challenging to combine the varying levels of noise and conflict between modalities. Multimodal approaches can be categorized into three groups considering how the multiple modalities are combined.

- Input-level or early fusion. Early fusion methods create a joint representation of the unimodal features extracted separately from multiple modalities. The simplest way to combine these unimodal feature vectors is to concatenate them to obtain a fused representation. Cf. Fig. 6. Next, a single model is trained to learn the correlation and interactions between the features of each modality. The final outcome of the model can be written as

  $$p = h\left(\left[v_1, v_2, \ldots, v_m\right]\right)$$

  where h denotes the single model, $\left[v_1, v_2, \ldots, v_m\right]$ represents the concatenation of the feature vectors, and m is the number of distinct unimodal feature vectors.
- Decision-level or late fusion. In contrast to early fusion, late fusion methods train one model per modality and fuses the learned decision values with a fusion mechanism such as averaging, voting, a learned model, etc. Cf. Fig. 7. The main advantage of late fusion is that it allows using different models on different modalities, thus being more flexible. In addition, as the predictions for each modality are made separately, it is easier to handle missing modalities. Supposing that model h_i is the decision value on modality i, the final prediction is

  $$p = F\left(h_1(v_1), h_2(v_2), \ldots h_m(v_m)\right)$$

  where F denotes the type of fusion strategy.
- Intermediate fusion. Intermediate fusion methods construct a shared representation by merging the intermediate features obtained by separate machine learning models. Afterwards, these intermediate features are concatenated and then a machine learning model is trained to capture the interactions between modalities. Cf. Fig. 8.

In addition, features extracted from both types of analysis, static and dynamic, can be combined to build more robust classifiers. Approaches that combine static analysis and dynamic analysis are known as hybrid approaches.

On the one hand, static analysis aims at finding malicious characteristics of an executable, app or program without actually running it. Static analysis is faster but suffers from code obfuscation. That is, malicious characteristics can be concealed using different obfuscation techniques (You et al., 2010) or by polymorphic and metamorphic malware (Moser et al., 2007). On the other hand, this obfuscation technique fails at dynamic analysis as it monitors and analyses the runtime behavior of a program during its execution in a controlled environment. But there are some limitations to dynamic analysis. The monitoring process is time consuming and the environment where the program is run must be secured as not to infect the platform. In addition, the controlled environment might be different from the real runtime environment and the malware may behave differently, causing an inexact behavior logging. Moreover, some actions of the program are only triggered if certain conditions are satisfied and may not be detected/activated in a controlled

Table 8
A side-by-side comparison of the features and fusion strategies of the reviewed multimodal and hybrid methods. Algorithms: Support Vector Machine (SVM), Naive Bayes (NB), Decision Tree (DT), Random Forest (RF), Convolutional Neural Network (CNN), Neural Network (NN), K-Nearest Neighbor (K-NN), Passive-Aggressive I (PA-I), Passive-Aggressive II (PA-II), Confidence Weighted Learning (CW), Adaptive Regularization of Weight (AROW), Normal Herd (NHERD), Logistic Regression (LR).

Paper	Fusion Strategy	Static Analysis	Dynamic Analysis	Classification Algorithm
Ahmadi et al. (2016)	Early and late fusion	Bytes 1-g, metadata features, entropy statistics, Haralick and Local Binary Pattern features, ASCII strings, symbol frequencies, opcode 1-g, register's usage, API function calls, section sizes, frequency of keywords	–	Ensemble of Gradient Boosting Trees
Microsoft Challenge winner's solution	Early and late fusion	Opcode 2, 3, 4-g, segment counts, asm pixel intensity, byte 4-g, single byte frequency, function names, derived assembly features	–	Ensemble of Gradient Boosting Trees
Kolosnjaji et al. (2017)	Intermediate fusion	Instruction traces, PE header features, imported functions and DLL files	–	CNN + Feedforward NN
Bayer et al. (2009)	Early fusion	–	API call traces and network traffic	Approximate Nearest Neighbor
Mohaisen and Alrawi (2013)	Early fusion	–	files created, modified or deleted, registry keys created, modified or deleted, destination IP addresses, hosts, TCP and UDP connections, requests and DNS records	SVM, LR, DT and K-NN
Dhammi and Singh (2015)	Early fusion	–	File details, signatures, hosts involved, affected files, registry keys, mutexes, section details, imports and strings	LMT, NB, SVM, Rider and K-NN
Pektaş and Acarman (2017)	Early fusion	–	File system, network and registry features, API call N-grams	PA-I, PA-II, CW, AROW, NHERD
Mohaisen et al. (2015)	Early fusion	–	File system, memory, network and registry based features	SVM, DT, LR, K-NN
Islam et al. (2013)	Early fusion	Function length frequency, string information	API function calls	SVM, RF, DT, Instance-based
Han et al. (2019a)	Early fusion based on semantic blocks	Static API sequences	Dynamic API sequences	K-NN, DT, RF, Extreme Gradient Boosting
Han et al. (2019b)	Early fusion	PE sections size, API sequence, DLL information	IP, port, DNS and domain request, file manipulation operations, registry modification operations	K-NN, DT, RF, Extreme Gradient Boosting
Kumar et al. (2019)	Early fusion	PE file metadata	Network data, system calls, process and registry features	RF, DT, XGBoost, NN, K-NN
Rhode et al. (2019)	Early fusion	Machine metrics	API calls	NN, RF, SVM

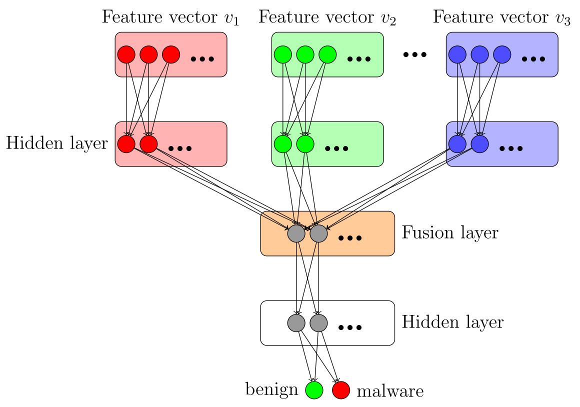


Fig. 8. Intermediate fusion strategy.

environment. Considering the advantages and disadvantages of static and dynamic malware detection, a natural improvement and line of research is to focus on hybrid schemes that combine elements of both.

Notice that hybrid approaches also include various modalities of data and could be included under the same category. The main difference between hybrid and multimodal approaches is that hybrid approaches combine features from both static and dynamic analysis while multimodal approaches do not have to.

A summary of the main characteristics of the multimodal and hybrid approaches for malware detection and classification is presented in Tables 8 and 9. A description of each of them is provided below.

Ahmadi et al. (2016) proposed a system that uses different malware features to effectively classify malware samples according to their corresponding family. For each malware sample, they extract a set of content-based and statistical features that reflect the structure of PE files. Then, these features are combined by stacking the feature categories into a single feature vector using a variation of the forward stepwise selection technique. Instead of gradually increasing the feature set by adding features to the model, one by one, they considered all the subset of features belonging to a category. The classification algorithm of their choice was a parallel implementation of the Gradient Boosting Tree classifier, XGBoost. Additionally, they used bagging to boost the classifier stability and accuracy. Their approach was evaluated on the Microsoft Malware Classification Challenge dataset (Ronen et al., 2018) and it achieved accuracy comparable to the winner of the competition[21]

Table 9
A side-by-side comparison of the dataset characteristics of the reviewed multimodal approaches.

Paper	Source	Total Size	Task
Ahmadi et al. (2016)	Microsoft Malware Classification Challenge	21741	Classification
Microsoft Challenge winner's solution	Microsoft Malware Classification Challenge	21741	Classification
Kolosnjaji et al. (2017)	–	22757	Classification
Bayer et al. (2009)	ANUBIS	2658	Classification
Mohaisen and Alrawi (2013)	–	3980	Detection
Dhammi and Singh (2015)	–	1270	Detection
Pektaş and Acarman (2017)	VirusShare	17900	Classification
Mohaisen et al. (2015)	–	115157	Classification
Islam et al. (2013)	–	2939	Detection
Han et al. (2019a)	VirusShare, Windows 7	6471	Classification
Han et al. (2019b)	VirusShare, Windows 7	4250	Classification
Kumar et al. (2019)	MalShare, VirusShare	120000	Classification
Rhode et al. (2019)	VirusShare, Commercial data	6809	Detection

but without requiring the same computational resources. On the other hand, the winning team relied on a large set of well-known features including, but not limited to, byte N-grams and opcode N-grams, which require large computational resources both during the training and the testing phases.

Kolosnjaji et al. (2017) proposed a neural network architecture that consists of convolutional and feed-forward subnetworks. The convolutional subnetwork learns features from sequences of disassembled malicious binaries. Conversely, the feed-forward network takes as input a set of features extracted from the metadata contained in the PE Header and the list of imported functions and their DLL files. Then, the final neural network-based classifier combines the feedforward and convolutional neural network architectures along with their corresponding features into a single network. This network generates the final classification output after aggregating the features learned by both subnetworks.

Bayer et al. (2009) built behavioral profiles of malware based on the system calls, their dependencies and network activities. This generalized representation serves as input to a clustering algorithm that groups malware samples that exhibit similar behavior. Clustering malware is a multi-step process. The first step is the automated analysis of the executables performed by an extended version of ANUBIS.[22] The second step is the extraction of the behavioral profile. Lastly, in the third step samples that exhibit similar behavior are grouped in the same cluster using an approximate, probabilistic approach based on locality sensitive hashing (Indyk and Motwani, 1998).

Mohaisen and Alrawi (2013) proposed a behavior based approach for identifying malware belonging to the Zeus family. The Zeus banking trojan is a form of malware that targets the Windows OS and is often used to steal money and credentials from the infected victim. For classification purposes, a set of 65 unique and robust features are extracted including files created, modified or deleted, registry keys created, modified or deleted, destination IP addresses, ports, TCP and UDP connections, requests, DNS records, etc. Then, the resulting feature vector is used to evaluate the performance of various M.L. algorithms such as SVM, LR, DT and K-NN.

Dhammi and Singh (2015) proposed a malware detection system based on the dynamic analysis of malware using the Cuckoo sandbox. Their approach extracted various features from the malware execution such as file details, signatures, hosts involved, affected files, registry keys, mutexes, section details, imports and strings. All the features obtained from Cuckoo are mapped into an Attribute Relation File Format (ARFF) file, and later, the resulting ARRF file is fed into WEKA (Hall et al., 2009) for classification.

Pektaş and Acarman (2017) presented a malware classification system based on runtime behavior by applying online machine learning. The system entails three stages. The first stage consists of monitoring the behavior of the file in sandbox environments; VirMon and Cuckoo. During the second stage, feature extraction is applied to build a feature vector consisting of features based on the file system, network and registry activities and API call N-grams. Finally, the third stage performs classification using online learning algorithms.

Mohaisen et al. (2015) presented AMAL, an automated behavior-based malware analysis system that provides tools to collect behavioral features that characterize malware based on the usage of the file system, memory, network and registry. Then, the resulting feature vector is used to perform classification with Support Vector Machine, Decision Tree, Logistic Regression and K-Nearest Neighbor algorithms.

Islam et al. (2013) presented a method integrating static and dynamic features into a single classification system. For each executable file, they extracted and converted to vector representations both function length frequency and printable string information. After running the executables and logging the Windows API calls, they extracted API features comprising API function names and parameters. Then, all feature vectors are combined into a single vector for each executable. Next, the resulting vector is used as input to four base classifiers: Support Vector Machine, Random Forest, Decision Tree and Instance-based.

Han et al. (2019a) built a malware detection framework based on the correlation and fusion of static and dynamic API call sequences. In their work, they explored the difference and relation between static and dynamic API call sequences by defining a number of types of malicious behaviors. After correlation and fusion, a hybrid feature vector space is established for detection and classification. To evaluate the effectiveness of their approach, they trained four classifiers to detect/classify malware including K-Nearest Neighbor, Decision Tree, Random Forest and Extreme Gradient Boosting.

Han et al. (2019b) presented MalInsight, a malware detection framework based on programs profiling of: (1) their basic structure, (2) their low-level behavior, and (3) their high-level behavior. These three aspects reflect structural features; the primary operations interacting with the OS, files, the registry and the network. The resulting feature set is used to train various machine learning classifiers: K-Nearest Neighbor, Decision Tree, Random Forests and Extreme Gradient Boosting. These classifiers were evaluated on a dataset consisting of 4250 samples obtained from VirusShare and from the Windows 7 Pro operative system. Results show accuracy of 97.21% in detecting unknown malware.

Kumar et al. (2019) used a combination of static and dynamic approaches to classify malware into types in the initial 4 s of its execution using a Random Forest classifier. Stopping the process early before the analysis is fully executed is known as early-stage detection. From static analysis they extracted information from the PE header such as file header, optional header and section header. They also extracted information from the section table and sections such as the number of sections, their size, the section virtual address, etc. From dynamic anal-

[21] http://blog.kaggle.com/2015/05/26/microsoft-_malware-_winners-_interview-_1st-_place-_no-_to-_overfitting/.
[22] http://anubis.iseclab.org.

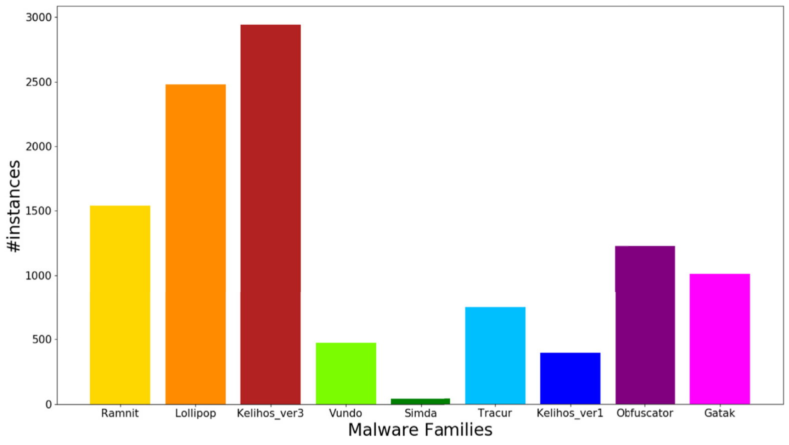


Fig. 9. Class distribution in the Microsoft Malware Classification Challenge dataset.

ysis they extracted features based on critical resources such as network data, system calls, process and registry. Afterwards, the feature set is reduced using the Information Gain algorithm. Finally, the resulting feature vector is used for classification purposes by training a Random Forest, Decision Tree, XGBoost, Neural Network and K-NN classifiers.

Rhode et al. (2019) collected two types of features: (1) API calls and (2) machine metrics. The Cuckoo sandbox was used to collect the API call features while Psutil library was used to collect the machine metrics. Machine metrics include user CPU usage, system CPU usage, memory use, swap use, bytes received and transmitted, number of packets received and transmitted, total number of processes, maximum process ID and time in seconds since execution began. Then, the two feature vectors are fused into a single vector that is used to detect malware using a Neural Network, Random Forest or Support Vector Machine as classifiers.

7. Research issues and challenges

This section presents some of the issues and challenges faced by security researchers. It is structured as follows: Section 7.1 presents the class imbalance problems. Section 7.2 reviews the availability of public benchmarks of malware for research. Finally, Section 7.3 discusses the problem of concept drift and presents various adversarial learning techniques to fool machine learning detectors.

7.1. Class imbalance

Obtaining good training data is one of the most challenging aspects of any machine learning problem. Machine learning classifiers are only as good as the data used to train them, and reliable labeled data is especially important for the task of malware detection, where the process of labeling a file can be a very time-consuming process.

Additionally, there are various disciplines including fraud detection, malware detection, malware classification, medical diagnosis, etc, where it is common to have a disproportional number of samples per class. For instance, the number of benign samples might not be proportionally equal to the number of malicious samples, or the number of samples belonging to one family might far exceed the number of samples from other families. This is known as the class imbalance problem (Japkowicz and Stephen, 2002; Guo et al., 2008).

By way of an example, let's look at the distribution of classes of the Microsoft dataset in Fig. 9. Families Kelihos_ver3, Lollipop and Ramnit have 2942, 2478, 1541 samples, respectively. On the other hand, families Simda and Kelihos_ver1 have 42 and 398 samples, respectively. This kind of distribution, where one class much larger than the other(s) can lead to a model that predicts the value of the majority classes for all predictions and still achieve high classification accuracy while lacking predictive power.

In other words, the classifier might be biased towards the majority classes and achieve very poor classification rates on the minority classes. It might happen that the classifier predicts everything as the major class and ends up ignoring the minor classes. This is called the accuracy paradox. In these cases, accuracy is a misleading measure. It may be desirable to select a less accurate model but with greater predictive power. For problems like this, additional measures are required to evaluate a classifier such as precision 1, recall 2 and the F1 score 3. Alternatively, the Receiver Operating Characteristic (ROC) curve graphically illustrates the discriminative ability of a binary classifier.

Precision (P) is the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p).

$$P = \frac{T_p}{T_p + F_p}. \tag{1}$$

Recall (R) is the number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n).

$$R = \frac{T_p}{T_p + F_n}. \tag{2}$$

The F1 score is the weighted average of precision, defined as following:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}. \tag{3}$$

Finally, the ROC curve is created by plotting the True Positive Rate (TPR) or recall against the False Positive Rate. The FPR is also known as the probability of false alarm and can be calculated as (1-Specificity) where Specificity is equal to $\frac{TN}{TN+FP}$. The higher the AUC, the better the model is at predicting the correct label of classes.

7.2. Open and public benchmarks

The task of malware detection and classification has not received the same attention in the research community as other applications, where rich benchmark datasets exist. These include digit classification, image labelling, speech recognition, etc. This situation has been exacerbated by legal restrictions. Even though malware binaries are shared generously through web sites such as VirusShare and VX Heaven, benign binaries are often protected by copyright laws that prevent sharing. Nevertheless, both benign and malicious binaries may be obtained in volume for internal use only through services such as VirusTotal, but subsequent sharing is prohibited. In addition, unlike other domains where data may be labeled very quickly and in many cases by a non-expert, determining whether a file is malicious or benign can be a time-consuming process, even for security experts. Furthermore, services like VirusTotal specifically restrict the public sharing of vendor antimalware labels.

The aforementioned issues render it impossible to meaningfully compare accuracy numbers across works, as different datasets are used with different labeling procedures. At the present time, the only standard benchmark available to the research community regarding Windows Portable Executables is the one provided by Microsoft (Ronen et al., 2018) for the Big Data Innovators Gathering Anti-Malware Prediction Challenge. The dataset is hosted on Kaggle and includes almost half a terabyte of malware consisting of around 20 K malicious samples from nine families. Each sample is comprised of two files: (1) the hexadecimal representation of the malware's binary content and (2) their corresponding disassembled file. Unfortunately, the byte code does not include the headers and thus, it is not possible to analyze dynamically the executables or to reproduce the disassembly process. In consequence, researchers are constrained to using only the provided byte code and disassembly files (generated with the IDA Pro disassembler).

7.3. Concept drift

In the machine learning literature, the term "concept drift" has been used to describe the problem of the changing underlying relationships in the data. Supervised learning is the machine learning task of learning a function that maps an input to an output based on a set of input-output samples. Technically speaking, it is the problem of approximating a mapping function (f) given input data (x) to predict an output value (y), $y = f(x)$. Traditional machine learning applications such as digit classification, text categorization or speech recognition, assume that training data is sampled from a stationary population. In other words, they assume that the mapping learning from historical data will be valid for new data in the future and that the relationships between input and output do not change over time. This is not true for the problem of malware detection and classification.

Software applications, including malware, naturally evolve over time due to changes resulting from adding features, fixing bugs, porting to new environments and platforms (Lehman, 1996). These changes are expected to be introduced relatively infrequently. Additionally, successive versions of the software are expected to be highly similar to previous versions, with few exceptions such as when the code base undergoes significant refactoring and there are changes in the compilers or libraries linked to the software. Moreover, the similarity between previous and future versions is expected to degrade slowly over time. In consequence, the prediction quality decays over time as malware evolves and new variants and families appear (Jordaney et al., 2017). Thus, in order to build high-quality models for malware detection and classification, it is important to identify when the model shows signs of degradation and thereby it fails to recognize new malware. Existing solutions (Kantchelian et al., 2013; Gama et al., 2014) aim at periodically retrain the model with the hope that it will automatically adapt to changes in malware over time. The process of retraining the model can be done from scratch, partially and incrementally, were incremental retraining refers to the process of retraining a given model with new labeled malware samples and all previous training samples without forgetting the knowledge obtained from prior datasets.

7.4. Adversarial learning

Malware is pushed to evolve in order to survive and operate. That is, malicious software has to constantly evolve to avoid detection by anti-malware engines. In consequence, malware writers are well-motivated to intentionally seek evasion by employing a wide range of obfuscation techniques (You et al., 2010; OKane et al., 2011).

To put it in the machine learning context, an attacker's aim is to fool the machine learning detector by camouflaging a piece of malware in feature space by inducing a feature representation highly correlated to benign behavior. The ability of the attacker to bypass machine learning solutions is related to their knowledge about features and machine learning models to target. For instance, consider a machine learning approach that relies on the program's invocations of API functions or the DLLs dynamically loaded by the executable. An attacker might use this information to conceal the usage of any suspicious API function by packing the executable and leaving only the stub of the import table or perhaps even no import table at all. These modifications to the feature space can be manually performed or not.

Adversarial machine learning (Huang et al., 2011) is a technique employed to attempt to fool machine learning by automatically crafting adversarial examples. That is, samples with small, intentional feature perturbations that cause a machine learning model to make an incorrect prediction. Machine learning-based detectors are vulnerable to adversarial examples, and the application of machine learning to the cybersecurity domain does not constitute an exception. For a detailed overview of the evolution of adversarial machnine learning over the past decade we refer to Biggio and Roli (2018). They reviewed the work done in the context of various applications, including computer security and its notion of arms race and proposed a comprehensive threat model that accounts for the presence of the attacker during the system design. Recent classifiers proposed for malware detection, have indeed shown to be easily fooled by well-crafted adversarial manipulations (Demetrio et al., 2019; Chen et al., 2017; Huang et al., 2018; Suciu et al., 2018; Maiorca et al., 2019). Chen et al. (2017) explored adversarial machine learning to attack a malware detector based on the input of Windows Application Programming Interface (API) calls extracted from the PE files.

Suciu et al. (2018) analyzed various append-based strategies to generate adversarial examples to conceal malware and bypass the MalConv (Raff et al., 2018a) model.

Furthermore, Demetrio et al. (2019) proposed a novel attack algorithm to generate adversarial malware binaries which only change a few tens of bytes of the file header. Their algorithm was evaluated against MalConv. They found that MalConv learns discriminative features mostly from the characteristics of the file header and used their findings to exploit and bypass the model. Contrarily, Maiorca et al. (2019) explored the types of adversarial attacks that have exploited the vulnerabilities of the components of PDFs to bypass malware detectors, including JavaScript-based attacks, ActionScript-based attacks and file embedding-based attacks.

7.5. Interpretability of the models

The interpretation of machine learning models is a new and open challenge (Shirataki and Yamaguchi, 2017; Gilpin et al., 2018). Most of the models used at the present time are treated as a black box. This black box is given an input X and it produces an output Y through a sequence of operations hardly understandable to a human. This could pose a problem in cybersecurity applications when a false alarm occurs as analysts would like to understand why it happened. The interpretability of the model determines how easily the analysts can manage and assess the quality and correct the operation of a given model. For this reason, cybersecurity analysts have preferred solutions that are more interpretable and understandable such as rule-based and signature-based systems rather than neural-based methods because they are easier to tune and optimize to mitigate and control the effect of false positives and false negatives. However, there is no work in the literature that investigates the interpretability of machine learning models for malware detection and classification.

8. Conclusions

This paper presents a systematic review of malware detection and classification approaches using machine learning. To sum up, a total of 67 research papers for tackling the problem of malware detection and classification on the Windows platform are reviewed. The reviewed papers are compared and analyzed according to various essential factors including the input features, the classification algorithm, the characteristics of the dataset and the objective task. There are four main contributions of our work.

First, we provide a detailed description of the methods and features in a traditional machine learning workflow, from the feature extraction, selection and reduction steps to classification. The traditional approaches are classified into three main categories: (1) static-based and (2) dynamic-based approaches and (3) hybrid approaches. On the one hand, static-based approaches extract features derived from a piece of program without involving its execution. On the other hand, dynamic-based approaches include those approaches that extract features from the execution of malware during runtime. Lastly, hybrid approaches are those that combine static and dynamic analysis to extract features.

Second, it arranges the existing literature on malware detection through deep learning and provides a comparative analysis of the approaches based on the network architecture and its input. Deep learning approaches are grouped considering the type of input of the networks: (1) methods that perform feature engineering to extract a feature vector representing the executable; (2) methods that take the gray scale representation of an executable as input; (3) methods that are fed with the sequence of API function invocations; (4) methods that model a program as a sequence of instructions; (5) methods that represent a computer program as a sequence of bytes; and (6) methods that aim to classify a program from its network traffic.

Third, it introduces new directions of research and present classifiers that rely on more than one type of feature or modality of data to detect malware. It organizes multimodal approaches into three groups, depending on how the different modalities of data are fused: (i) early-fusion methods create a joint representation of the unimodal feature vectors; (ii) late-fusion methods train one model per modality and fuses the output decision values; and (iii) intermediate-fusion methods construct a shared representation by merging the intermediate features obtained by separate models.

Fourth, it discusses the most important research issues and challenges faced by researchers. Special emphasis is placed on the problem of concept drift and the challenges of adversarial learning. Furthermore, it examines the status of the benchmarks used by the scientific community to evaluate the performance of their methods and reviews the problem of class imbalance.

Declaration of competing interest

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

Acknowledgements

This research has been partially funded by the Spanish MICINN Projects TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, and is supported by the University of Lleida. This research article has received a grant (2019 call) from the University of Lleida Language Institute to review the English.

References

Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., Giacinto, G., 2016. Novel feature extraction, selection and fusion for effective malware family classification. CODASPY 16. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 183–194, https://doi.org/10.1145/2857705.2857713.

AL-Hawawreh, M., Moustafa, N., Sitnikova, E., 2018. Identification of malicious activities in industrial internet of things based on deep learning models. Journal of Information Security and Applications. 41, 1–11. http://www.sciencedirect.com/science/article/pii/S2214212617306002.

Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T., Nov 2011. Graph-based malware detection using dynamic analysis. J. Comput. Virol. 7 (4), 247–258, https://doi.org/10.1007/s11416-_011-_0152-_x.

Athiwaratkun, B., Stokes, J.W., March 2017. Malware classification with lstm and gru language models and a character-level cnn. In: 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 2482–2486.

Bayer, U., Comparetti, P.M., Hlauschek, C., Krgel, C., Kirda, E., 2009. Scalable, behavior-based malware clustering. In: NDSS. The Internet Society. p. Section4 http://dblp.uni-_trier.de/db/conf/ndss/ndss2009.html#BayerCHKK09.

Baysa, D., Low, R.M., Stamp, M., Nov 2013. Structural entropy and metamorphic malware. Journal of Computer Virology and Hacking Techniques 9 (4), 179–192, https://doi.org/10.1007/s11416-_013-_0185-_4.

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

Bekerman, D., Shapira, B., Rokach, L., Bar, A., 2015. Unknown malware detection using network traffic classification. 09. In: 2015 IEEE Conference on Communications and Network Security (CNS), pp. 134–142.

Biggio, B., Roli, F., 2018. Wild patterns: ten years after the rise of adversarial machine learning. Pattern Recognit. 84, 317–331. http://www.sciencedirect.com/science/article/pii/S0031320318302565.

Boukhtouta, A., Mokhov, S.A., Lakhdari, N.-E., Debbabi, M., Paquet, J., May 2016. Network malware classification comparison using dpi and flow packet headers. Journal of Computer Virology and Hacking Techniques 12 (2), 69–100, https://doi.org/10.1007/s11416-_015-_0247-_x.

Carlin, D., Cowan, A., O'Kane, P., Sezer, S., 2017a. The effects of traditional anti-virus labels on malware detection using dynamic runtime opcodes. IEEE Access 5, 17742–17752.

Carlin, D., O'Kane, P., Sezer, S., 2017b. Dynamic Analysis of Malware Using Run-Time Opcodes. Springer International Publishing, Cham, https://doi.org/10.1007/978-_3-_319-_59439-_2\T1\textbackslash_4.

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

Corporation, S., 2018. Symantec 2018 Internet Security Threat Report. Tech. rep. Symantec Corporation, https://www.symantec.com/content/dam/symantec/docs/reports/istr-_23-_executive-_summary-_en.pdf.

Dahl, G.E., Stokes, J.W., Deng, L., Yu, D., May 2013. Large-scale malware classification using random projections and neural networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 3422–3426.

Davis, A., Wolff, M., Wojnowicz, M., Soeder, D.A., Zhao, X., 2017. Neural Attention Mechanisms for Malware Analysis. 07. https://patentimages.storage.googleapis.com/4f/e5/74/b62fd3b08788bd/US9705904.pdf.

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., 2019. Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries. CoRR abs/1901.03583. http://arxiv.org/abs/1901.03583.

Dhammi, A., Singh, M., Aug 2015. Behavior analysis of malware using machine learning. In: 2015 Eighth International Conference on Contemporary Computing (IC3), pp. 481–486.

Dinaburg, A., Royal, P., Sharif, M., Lee, W., 2008. Ether: malware analysis via hardware virtualization extensions. CCS 08. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 51–62, https://doi.org/10.1145/1455770.1455779.

Ding, Y., Yuan, X., Tang, K., Xiao, X., Zhang, Y., 2013. A fast malware detection algorithm based on objective-oriented association mining. Comput. Secur. 39, 315–324. http://www.sciencedirect.com/science/article/pii/S0167404813001259.

Eskandari, M., Hashemi, S., 2011. Metamorphic malware detection using control flow graph mining. 06 International Journal of Computer Science and Network Security 11 (12).

Faruki, P., Laxmi, V., Gaur, M.S., Vinod, P., 2012. Mining control flow graph as api call-grams to detect portable executable malware. SIN 12. In: Proceedings of the Fifth International Conference on Security of Information and Networks. ACM, New York, NY, USA, pp. 130–137, https://doi.org/10.1145/2388576.2388594.

Fraley, J.B., Cannady, J., March 2017. The promise of machine learning in cybersecurity. In: SoutheastCon 2017, pp. 1–6.

Fuyong, Z., Tiezhu, Z., July 2017. Malware detection and classification based on n-grams attribute similarity. In: 2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC), vol. 1, pp. 793–796.

Galal, H.S., Mahdy, Y.B., Atiea, M.A., May 2016. Behavior-based features model for malware detection. Journal of Computer Virology and Hacking Techniques 12 (2), 59–67, https://doi.org/10.1007/s11416-_015-_0244-_0.

Gama, J.a., liobait, I., Bifet, A., Pechenizkiy, M., Bouchachia, A., Mar. 2014. A survey on concept drift adaptation. ACM Comput. Surv. 46 (4), https://doi.org/10.1145/2523813 44:144:37.

Ghiasi, M., Sami, A., Salehi, Z., Sep. 2012. Dynamic malware detection using registers values set analysis. In: 2012 9th International ISC Conference on Information Security and Cryptology, pp. 54–59.

Ghiasi, M., Sami, A., Salehi, Z., 2015. Dynamic vsa: a framework for malware detection based on register contents. Eng. Appl. Artif. Intell. 44, 111–122. http://www.

sciencedirect.com/science/article/pii/S0952197615001190.
Gibert, D., Bjar, J., Mateu, C., Planes, J., Solis, D., Vicens, R., 2017. Convolutional neural networks for classification of malware assembly code. In: Recent Advances in Artificial Intelligence Research and Development - Proceedings of the 20th International Conference of the Catalan Association for Artificial Intelligence, Deltebre, Terres de l'Ebre, Spain, October 25-27, 2017, pp. 221–226, https://doi.org/10.3233/978-_1-_61499-_806-_8-_221.
Gibert, D., Mateu, C., Planes, J., 2018a. An end-to-end deep learning architecture for classification of malware's binary content. In: Krkov, V., Manolopoulos, Y., Hammer, B., Iliadis, L., Maglogiannis, I. (Eds.), Artificial Neural Networks and Machine Learning ICANN 2018. Springer International Publishing, Cham, pp. 383–391.
Gibert, D., Mateu, C., Planes, J., Vicens, R., 2018b. Classification of malware by using structural entropy on convolutional neural networks. In: IAAI Conference on Artificial Intelligence, pp. 7759–7764. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16133.
Gibert, D., Mateu, C., Planes, J., Vicens, R., Aug 2018c. Using convolutional neural networks for classification of malware represented as images. Journal of Computer Virology and Hacking Techniques, https://doi.org/10.1007/s11416-_018-_0323-_0.
Gibert, D., Mateu, C., Planes, J., 2019. A hierarchical convolutional neural network for malware classification. In: The International Joint Conference on Neural Networks 2019. IEEE, pp. 1–8.
Gilpin, L.H., Bau, D., Yuan, B.Z., Bajwa, A., Specter, M., Kagal, L., 2018. Explaining Explanations: an Approach to Evaluating Interpretability of Machine Learning. CoRR abs/1806.00069 http://arxiv.org/abs/1806.00069.
Guo, X., Yin, Y., Dong, C., Yang, G., Zhou, G., Oct 2008. On the class imbalance problem. In: 2008 Fourth International Conference on Natural Computation, vol. 4, pp. 192–201.
Hall, M.A., 1999. Correlation-based Feature Selection for Machine Learning. Ph.D. thesis. The University of Waikato.
Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., Nov. 2009. The weka data mining software: an update. SIGKDD Explor. Newsl. 11 (1), 10–18, https://doi.org/10.1145/1656274.1656278.
Han, W., Xue, J., Wang, Y., Huang, L., Kong, Z., Mao, L., 2019a. Maldae: detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. Comput. Secur. 83, 208–233. http://www.sciencedirect.com/science/article/pii/S016740481831246X.
Han, W., Xue, J., Wang, Y., Liu, Z., Kong, Z., 2019b. Malinsight: a systematic profiling based malware detection framework. J. Netw. Comput. Appl. 125, 236–250. http://www.sciencedirect.com/science/article/pii/S1084804518303503.
Hassen, M., Chan, P.K., 2017. Scalable function call graph-based malware classification. CODASPY 17. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 239–248, https://doi.org/10.1145/3029806.3029824.
Hu, X., Shin, K.G., Bhatkar, S., Griffin, K., 2013. Mutantx-s: scalable malware clustering based on static features. In: Presented as Part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13). USENIX, San Jose, CA, pp. 187–198. https://www.usenix.org/conference/atc13/technical-_sessions/presentation/hu.
Huang, W., Stokes, J.W., 2016. Mtnet: a multi-task neural network for dynamic malware classification. In: Caballero, J., Zurutuza, U., Rodrguez, R.J. (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment. Springer International Publishing, Cham, pp. 399–418.
Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I., Tygar, J.D., 2011. Adversarial machine learning. AISec 11. In: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence. ACM, New York, NY, USA, pp. 43–58, https://doi.org/10.1145/2046684.2046692.
Huang, A., Al-Dujaili, A., Hemberg, E., O'Reilly, U., 2018. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. CoRR abs/1801.02950. http://arxiv.org/abs/1801.02950.
Indyk, P., Motwani, R., 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. STOC 98. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing. ACM, New York, NY, USA, pp. 604–613, https://doi.org/10.1145/276698.276876.
Islam, R., Tian, R., Batten, L.M., Versteeg, S., 2013. Classification of malware based on integrated static and dynamic features. J. Netw. Comput. Appl. 36 (2), 646–656. http://www.sciencedirect.com/science/article/pii/S1084804512002214.
Jain, S., Meena, Y.K., 2011. Byte level ngram analysis for malware detection. In: Venugopal, K.R., Patnaik, L.M. (Eds.), Computer Networks and Intelligent Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 51–59.
Japkowicz, N., Stephen, S., 2002. The class imbalance problem: a systematic study. Intell. Data Anal. 429–449.
Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L., Aug. 2017. Transcend: detecting concept drift in malware classification models. In: 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, BC, pp. 625–642. https://www.usenix.org/conference/usenixsecurity17/technical-_sessions/presentation/jordaney.
Kancherla, K., Mukkamala, S., April 2013. Image visualization based malware detection. In: 2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS), pp. 40–44.
Kantchelian, A., Afroz, S., Huang, L., Islam, A.C., Miller, B., Tschantz, M.C., Greenstadt, R., Joseph, A.D., Tygar, J.D., 2013. Approaches to adversarial drift. AISec 13. In: Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security. ACM, New York, NY, USA, pp. 99–110, https://doi.org/10.1145/2517312.2517320.
Kheir, N., 2013. Behavioral classification and detection of malware through http user agent anomalies. Journal of Information Security and Applications 18 (1), 2–13 sETOP2012 and FPS2012 Special Issue, http://www.sciencedirect.com/science/article/pii/S2214212613000331.
Kinable, J., Kostakis, O., Nov 2011. Malware classification based on call graph clustering. J. Comput. Virol. 7 (4), 233–245, https://doi.org/10.1007/s11416-_011-_0151-_y.
Kolias, C., Kambourakis, G., Stavrou, A., Voas, J., 2017. Ddos in the iot: Mirai and other botnets. Computer 50 (7), 80–84.
Kolosnjaji, B., Zarras, A., Webster, G., Eckert, C., 2016. Deep learning for classification of malware system call sequences. In: Kang, B.H., Bai, Q. (Eds.), AI 2016: Advances in Artificial Intelligence. Springer International Publishing, Cham, pp. 137–149.
Kolosnjaji, B., Eraisha, G., Webster, G., Zarras, A., Eckert, C., May 2017. Empowering convolutional networks for malware classification and analysis. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp. 3838–3845.
Konopisky, D., 10 2018. Malware Detection in Application Based on Presence of Computer Generated Strings. https://patentscope.wipo.int/search/en/detail.jsf?docIdWO2018177602&tabPCTBIBLIO&queryString&recNum29&maxRec71152078.
Krl, M., vec, O., Blek, M., Jaek, O., 2018. Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only. https://openreview.net/forum?idHkHrmM1PM.
Kumar, N., Mukhopadhyay, S., Gupta, M., Handa, A., Shukla, K.S, Aug 2019. Malware classification using early stage behavioral analysis. In: 2019 14th Asia Joint Conference on Information Security (AsiaJCIS), pp. 16–23.
Lee, J., Im, C., Jeong, H., 2011. A study of malware detection and classification by comparing extracted strings. ICUIMC 11. In: Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication. ACM, New York, NY, USA, p. 75, https://doi.org/10.1145/1968613.1968704. 175:4.
Lehman, M.M., 1996. Laws of software evolution revisited. In: Montangero, C. (Ed.), Software Process Technology. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 108–124.
Ligh, M., Adair, S., Hartstein, B., Richard, M., 2010. Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code. Wiley Publishing.
Lyda, R., Hamrock, J., March 2007. Using entropy analysis to find encrypted and packed malware. IEEE Security Privacy 5 (2), 40–45.
Maiorca, D., Biggio, B., Giacinto, G., Aug. 2019. Towards adversarial malware detection: lessons learned from pdf-based attacks. ACM Comput. Surv. 52 (4), https://doi.org/10.1145/3332184 78:178:36.
Mohaisen, A., Alrawi, O., 2013. Unveiling zeus: automated classification of malware samples. In: Proceedings of the 22Nd International Conference on World Wide Web. WWW 13 Companion. ACM, New York, NY, USA, pp. 829–832, https://doi.org/10.1145/2487788.2488056.
Mohaisen, A., Alrawi, O., Mohaisen, M., 2015. Amal: high-fidelity, behavior-based automated malware analysis and classification. Comput. Secur. 52, 251–266. http://www.sciencedirect.com/science/article/pii/S0167404815000425.
Monnappa, 2018. Learning Malware Analysis: Explore the Concepts, Tools, and Techniques to Analyze and Investigate Windows Malware. Packt Publishing.
Moser, A., Kruegel, C., Kirda, E., Dec 2007. Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 421–430.
Moskovitch, R., Stopel, D., Feher, C., Nissim, N., Elovici, Y., June 2008. Unknown malcode detection via text categorization and the imbalance problem. In: 2008 IEEE International Conference on Intelligence and Security Informatics, pp. 156–161.
Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S., 2011. Malware images: visualization and automatic classification. VizSec 11. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security. ACM, New York, NY, USA, https://doi.org/10.1145/2016904.2016908. pp. 4:14:7.
OKane, P., Sezer, S., McLaughlin, K., Sept 2011. Obfuscation: the hidden malware. IEEE Security Privacy 9 (5), 41–47.
on Cybersecurity for the 44th Presidency, C. C., Langevin, J., Lewis, J., for Strategic, C., International Studies (Washington, D, 2010. A Human Capital Crisis in Cybersecurity: Technical Proficiency Matters. a White Paper of the CSIS Commission on Cybersecurity for the 44th Presidency. Center for Strategic and International Studies. https://books.google.es/books?idZa-_MnQAACAAJ.
Okane, P., Sezer, S., McLaughlin, K., May, 2016. Detecting obfuscated malware using reduced opcode set and optimised runtime trace. Security Informatics 5 (1), 2, https://doi.org/10.1186/s13388-_016-_0027-_2.
Pekta, A., Acarman, T., 2017. Classification of malware families based on runtime behaviors. Journal of Information Security and Applications. 37, 91–100. http://www.sciencedirect.com/science/article/pii/S2214212617301643.
Perdisci, Roberto, Wenke Lee, G.O., 01 2015. Method and System for Network-Based Detecting of Malware from Behavioral Clustering. https://patentimages.storage.googleapis.com/42/60/cd/37786f1ef6be24/US20150026808A1.pdf.
Prasse, P., Machlica, L., Pevn, T., Havelka, J., Scheffer, T., 2017. Malware detection by analysing encrypted network traffic with neural networks. In: Ceci, M., Hollmn, J., Todorovski, L., Vens, C., Deroski, S. (Eds.), Machine Learning and Knowledge Discovery in Databases. Springer International Publishing, Cham, pp. 73–88.
Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.K., 2018a. Malware detection by eating a whole EXE. In: The Workshops of the the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018, pp. 268–276. https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422.
Raff, E., Zak, R., Cox, R., Sylvester, J., Yacci, P., Ward, R., Tracy, A., McLean, M., Nicholas, C., Feb, 2018b. An investigation of byte n-gram features for malware classification. Journal of Computer Virology and Hacking Techniques 14 (1), 1–20, https://doi.org/10.1007/s11416-_016-_0283-_1.
Razak, M.F.A., Anuar, N.B., Salleh, R., Firdaus, A., 2016. The rise of malware: bibliometric analysis of malware study. J. Netw. Comput. Appl. 75, 58–76. http://

www.sciencedirect.com/science/article/pii/S1084804516301904.
Rezende, E., Ruppert, G., Carvalho, T., Ramos, F., de Geus, P., Dec 2017. Malicious software classification using transfer learning of resnet-50 deep neural network. In: 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 1011–1014.
Rhode, M., Tuson, L., Burnap, P., Jones, K., June 2019. Lab to soc: robust features for dynamic malware detection. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Industry Track, pp. 13–16.
Rieck, K., Trinius, P., Willems, C., Holz, T., Dec. 2011. Automatic analysis of malware behavior using machine learning. J. Comput. Secur. 19 (4), 639–668. http://dl.acm.org/citation.cfm?id2011216.2011217.
Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M., Feb. 2018. Microsoft malware classification challenge. ArXiv e-prints.
Salehi, Z., Sami, A., Ghiasi, M., 2017. Maar: robust features to detect malicious activity based on api calls, their arguments and return values. Eng. Appl. Artif. Intell. 59, 93–102. http://www.sciencedirect.com/science/article/pii/S0952197616302512.
Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., Hamze, A., 2010. Malware detection based on mining api calls. In: Proceedings of the 2010 ACM Symposium on Applied Computing. SAC 10. ACM, New York, NY, USA, pp. 1020–1025, https://doi.org/10.1145/1774088.1774303.
Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G., 2013. Opcode sequences as representation of executables for data-mining-based unknown malware detection. Inf. Sci. 231, 64–82 data Mining for Information Security, http://www.sciencedirect.com/science/article/pii/S0020025511004336.
Saxe, J., Berlin, K., Oct 2015. Deep neural network based malware detection using two dimensional binary program features. In: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), pp. 11–20.
Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C., 2009. Detection of Malicious Code by Applying Machine Learning Classifiers on Static Features: A State-Of-The-Art Survey. Information Security Technical Report 14 (1). , pp. 16–29 malware http://www.sciencedirect.com/science/article/pii/S1363412709000041.
Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., Elovici, Y., Feb, 2012. Detecting unknown malicious code by applying classification techniques on opcode patterns. Security Informatics 1 (1), 1, https://doi.org/10.1186/2190-_8532-_1-_1.
Shirataki, S., Yamaguchi, S., Dec 2017. A study on interpretability of decision of machine learning. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 4830–4831.
Sikorski, M., Honig, A., 2012. Practical Malware Analysis: the Hands-On Guide to Dissecting Malicious Software, first ed. No Starch Press, San Francisco, CA, USA.
Snort, 2015. Snort Network Intrusion Detection System. Tech. rep., Snort.. https://www.snort.org/.
Sorokin, I., Jun 2011. Comparing files using structural entropy. J. Comput. Virol. 7 (4), 259, https://doi.org/10.1007/s11416-_011-_0153-_9.
Souri, A., Hosseini, R., Jan 2018. A state-of-the-art survey of malware detection approaches using data mining techniques. Human-centric Computing and Information Sciences 8 (1), 3, https://doi.org/10.1186/s13673-_018-_0125-_x.
Storlie, C., Anderson, B., Wiel, S., Quist, D., Hash, C., Brown, N., 2014. Stochastic identification of malware with dynamic traces. Ann. Appl. Stat. 8 (1), 1–18.
Suciu, O., Coull, S.E., Johns, J., 2018. Exploring Adversarial Examples in Malware Detection. CoRR abs/1810.08280. http://arxiv.org/abs/1810.08280.
Ucci, D., Aniello, L., Baldoni, R., 2019. Survey of machine learning techniques for malware analysis. Comput. Secur. 81, 123–147. http://www.sciencedirect.com/science/article/pii/S0167404818303808.
Uppal, D., Sinha, R., Mehra, V., Jain, V., Sep. 2014. Malware detection and classification based on extraction of api sequences. In: 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 2337–2342.
VirusShare, 2011. Vxshare. https://virusshare.com/.
Wojnowicz, M., Chisholm, G., Wolff, M., Zhao, X., 2016. Wavelet decomposition of software entropy reveals symptoms of malicious code. Journal of Innovation in Digital Ecosystems. 3 (2), 130–140. http://www.sciencedirect.com/science/article/pii/S2352664516300220.
Yan, X., Han, J., Afshar, R., 2003. Clospan: mining closed sequential patterns in large datasets. In: Proceedings of the 2003 SIAM International Conference on Data Mining, pp. 166–177.
Ye, Y., Wang, D., Li, T., Ye, D., Jiang, Q., Nov 2008b. An intelligent pe-malware detection system based on association mining. J. Comput. Virol. 4 (4), 323–334, https://doi.org/10.1007/s11416-_008-_0082-_4.
Ye, Y., Li, T., Adjeroh, D., Iyengar, S.S., Jun 2017. A survey on malware detection using data mining techniques. ACM Comput. Surv. 50 (3), https://doi.org/10.1145/3073559 41:141:40.
Ye, Y., Chen, L., Wang, D., Li, T., Jiang, Q., Zhao, M., Nov 2008a. Sbmds: an interpretable string based malware detection system using svm ensemble with bagging. J. Comput. Virol. 5 (4), 283, https://doi.org/10.1007/s11416-_008-_0108-_y.
You, I., Yim, K., Nov 2010. Malware obfuscation techniques: a brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, pp. 297–300.
Yuval Nativ, L.L., 2015. 5fingers. The zoo https://github.com/ytisf/theZoo.
Yuxin, D., Siyi, Z., Feb 2019. Malware detection based on deep learning algorithm. Neural Comput. Appl. 31 (2), 461–472, https://doi.org/10.1007/s00521-_017-_3077-_6.
Zhang, X., Zhao, J., LeCun, Y., 2015. Character-level convolutional networks for text classification. In: Proceedings of the 28th International Conference on Neural Information Processing Systems, ume 1. MIT Press, Cambridge, MA, USA, pp. 649–657 NIPS15, http://dl.acm.org/citation.cfm?id2969239.2969312.
Zhao, G., Xu, K., Xu, L., Wu, B., 2015. Detecting apt malware infections based on malicious dns and traffic analysis. IEEE Access 3, 1132–1142.

Daniel Gibert is a PhD student at the Department of Computer Science and Industrial Engineering of the University of Lleida, in Spain. He graduated in 2014 and received a Masters degree in Artificial Intelligence in 2016 from the Polytechnic University of Catalonia. His research interests include intrusion detection and machine learning.

Carles Mateu received his B. Sc. from Universitat de Lleida, M. Sc. from Open University of Catalonia, and Ph.D. from University of Lleida in 2009. He is currently an Associate Professor at the University of Lleida. His research interests include Security, Energy Storage, Energy Efficiency and Artificial Intelligence.

Jordi Planes received his B. Sc. from Universitat de Lleida, M. Sc. from University Rovira i Virgili, and Ph.D. from University of Lleida in 2007. He is currently an Associate Professor at the University of Lleida. His research interests include Applications of Neural Networks.

Chapter 6

Using Convolutional Neural Networks for Classification of Malware Represented as Images

This article has been published in the Journal of Computer Virology and Hacking Techniques (SJR: 0.439), belonging to the Second Quartile (Q2) as classified by Scimago Journal Rank. See Table 6.1.

Table 6.1: Journal metrics corresponding to the Journal of Computer Virology and Hacking Techniques for the year 2019.

Journal Metric	Value
Citescore	Not applicable
Impact Factor	1.79
SNIP	Not applicable
SJR	0.439

In the article *"Using Convolutional Neural Networks for Classification of Malware Represented as Images"* [32] malicious software is visualized as gray scale images [59] since they capture minor changes while retaining the global structure of the executable helping to detect variations. To visualize a malware sample as an image, every byte has to be interpreted as one pixel in an image. Then, the resulting array has to be organized as a 2-D array and visualized as a gray scale image.

Figure 6.1 shows the representation of samples of malware belonging to nine different families as gray scale images. It can be observed that images of software executables from a given family are similar visually while distinct from those belonging to a different family. The main benefit of visualizing a malicious executable as an image is that the different sections of a binary can be easily differentiated. In addition, malware authors only used to change a small part of the code to produce new variants. Thus, if old malware is re-used to create new binaries the resulting ones would be very similar. Additionally, by representing malware as an image it is possible to detect the small changes while retaining the global structure of samples belonging to the same family. Thus, this research article proposes a file agnostic deep learning approach for malware categorization to efficiently group malicious software into families based on a set of discriminant patterns extracted from their visualization as gray scale images.

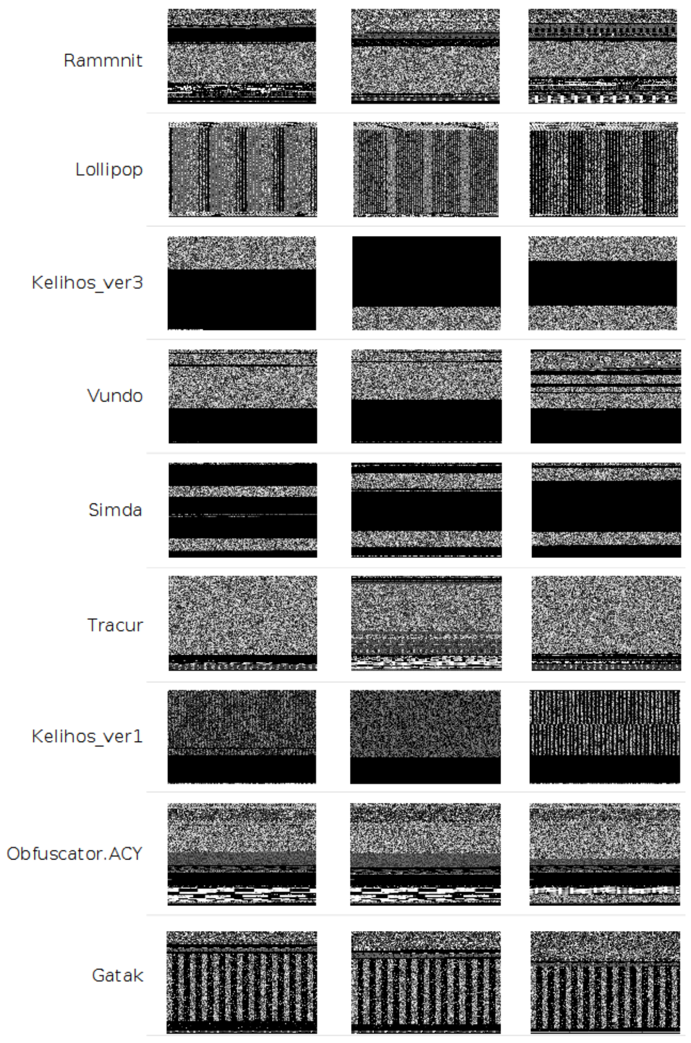


Figure 6.1: Gray scale images of malicious software belonging to various families. Note that the images of malware belonging to the same family are similar while distinct from the images of malware from the rest of families.

Journal of Computer Virology and Hacking Techniques (2019) 15:15–28
https://doi.org/10.1007/s11416-018-0323-0

ORIGINAL PAPER

Using convolutional neural networks for classification of malware represented as images

Daniel Gibert[1,2] · **Carles Mateu**[2] · **Jordi Planes**[2] · **Ramon Vicens**[1]

Received: 17 February 2018 / Accepted: 10 August 2018 / Published online: 27 August 2018

Abstract
The number of malicious files detected every year are counted by millions. One of the main reasons for these high volumes of different files is the fact that, in order to evade detection, malware authors add mutation. This means that malicious files belonging to the same family, with the same malicious behavior, are constantly modified or obfuscated using several techniques, in such a way that they look like different files. In order to be effective in analyzing and classifying such large amounts of files, we need to be able to categorize them into groups and identify their respective families on the basis of their behavior. In this paper, malicious software is visualized as gray scale images since its ability to capture minor changes while retaining the global structure helps to detect variations. Motivated by the visual similarity between malware samples of the same family, we propose a file agnostic deep learning approach for malware categorization to efficiently group malicious software into families based on a set of discriminant patterns extracted from their visualization as images. The suitability of our approach is evaluated against two benchmarks: the MalImg dataset and the Microsoft Malware Classification Challenge dataset. Experimental comparison demonstrates its superior performance with respect to state-of-the-art techniques.

Keywords Malware visualization · Malware classification · Convolutional neural network · Deep learning

1 Introduction

Malware, short for malicious software, refers to software programs designed to perform any kind of unwanted or harmful action on a computer system. This actions are targeted towards spying and stealing sensitive and critical information or damaging or modifying a compromised system. Nowadays, malware is a mature and growing business involving networks of developers and criminal organizations. It is a global industry worth millions of dollars that grows every year. According to McAfee [16] more than 600 million malicious programs were detected during the first quarter of 2017. However, the majority of new malware samples are deployed as variants of previously known samples. As a result, malicious software can be grouped together into families with each family originating from a single source base and exhibiting a set of consistent behaviors. In consequence, these shared characteristics between samples belonging to the same family might be used for detection and classification of unseen programs.

As the complexity and distribution of malware rises, the techniques used by malware developers to hide their malicious intent have improved. On the one hand, polymorphic malware uses a polymorphic engine to mutate the code while keeping the original functionality intact. Packing and encryption are the two most common ways to hide code. On the other hand, metamorphic malware rewrites its code every time it is propagated to an equivalent one. Traditionally, antivirus solutions relied on signature-based and heuristic-based methods. A signature is an algorithm or hash that uniquely identifies a specific malware. Given an unknown file, this method looks for any match between existing signatures in a database and the generated signature of the unknown file. Signatures are sensitive even to small program variations. On the contrary, heuristics are a set of rules determined by experts after

Daniel Gibert
daniel.gibert@diei.udl.cat

Carles Mateu
carlesm@diei.udl.cat

Jordi Planes
jordi.planes@diei.udl.cat

Ramon Vicens
ramon.vicens@blueliv.com

1 Blueliv, Leap in Value, Barcelona, Spain

2 University of Lleida, Lleida, Spain

analyzing the behavior of malware. The main drawback of both approaches is that the malware has to be analyzed prior to the creation of these rules and heuristics. Unfortunately, such systems fail to predict new unseen malware and it is unfeasible to analyze manually every sample received. Thus, techniques to automatically categorize malware are required.

The main contributions of this paper are the following:

- The development of the first, to our knowledge, file agnostic deep learning system that learns visual features from executable files to classify malware into families. This is achieved by training a convolutional neural network on the representation of malware's binary content as gray scale images. Malware executables are visualized as gray scale images because its ability to capture minor changes while retaining the global structure is useful for detecting variations in samples.
- An extensive assessment of our technique using standard evaluation metrics (e.g. accuracy, precision, recall, F1-score) on two publicly available datasets, the MalIMG and the Microsoft Malware Classification Challenge datasets, one containing 9458 samples of 25 different malware families and a second containing 10868 samples of 9 different families, respectively.
- A comparative study of image-based machine learning techniques for malware classification. The experiments show that our model is capable of classifying samples from both datasets with 98.48% and 97.49% of accuracy, respectively, outperforming state-of-the-art methods in the literature.

The rest of the paper is organized as follows: Section 2 discusses the related research and describes the insights from representing malicious software as gray scale images. Section 3 introduces our deep learning approach for malware classification. Section 4 evaluates the performance of our method. Finally, Section 5 concludes with our remarks and future work suggestions.

2 Background

This section provides an overview of the types of analysis for determining the purpose and functionality of a given malware sample. Afterwards, it is provided a description of features and machine learning algorithms for addressing the problem of malware detection and classification.

2.1 Malware analysis

Malware analysis involves mainly two techniques: static analysis and dynamic analysis. Static analysis consists of examining the code or structure of a program without executing it. This kind of analysis can confirm whether a file is malicious, provide information about its functionality and can also be used to produce a simple set of signatures. The most common static analysis approaches are:

- Finding sequences of characters or strings. Searching through the strings of a program is the most simple way to obtain hints about its functionality. For instance, you can find strings related to printed messages, URLs accessed by the program, the location of files modified by the executable and names of common Windows dynamic link libraries (DLLs).
- Analysis of the Portable Executable File Format. The Portable Executable(PE) file format is used by Windows executables, object code and DLLs. Among the information it includes, the most useful pieces of information are the linked libraries and functions as well as the metadata about the file included in the headers.
- Searching for packed/encrypted code. Malware writers usually use packing and encryption to make their files more difficult to analyze. Software programs that have been packed or encrypted usually contain very few strings and higher entropy compared to legitimate programs.
- Disassembling the program, i.e. recovering the symbolic representation from the machine code instructions.

Dynamic analysis involves executing the program and monitoring its behavior on the system. Unlike static analysis, dynamic analysis allows to observe the actual actions executed by the a program. It is typically performed when static analysis has reached a dead end, either due to obfuscation and packing, or by having exhausted the available static analysis techniques. A survey on automated dynamic analysis techniques and tools is found in M. Egele et al [6]. Some techniques are:

- Function Call Monitoring. The behavior of the program is analyzed by using the traces containing the sequence of functions invoked by the executable under analysis.
- Function Parameter Analysis. Consists of tracking the values of parameters and function return values.
- Information Flow Tracking. Analyze how a program processes data and how data is propagated through the system.
- Instruction Trace. Analysis of the complete sequence of machine instructions executed by the program.

Both methods have its own advantages and disadvantages. On the one hand, static analysis is faster but suffers from code obfuscation, techniques used by malware authors to conceal the malicious purpose of the program. On the other hand, code obfuscation techniques and polymorphic malware fails at dynamic analysis because it analyses the runtime behavior of a program by monitoring it while in execution. However, each malware sample must be executed in a safe environment for a specific time for monitoring its behavior, which is a very time consuming process. In addition, the environment might be quite different from the real runtime environment and malware may behave in different ways in the two environments, and under some conditions the actions of malware might not be triggered and thus, not logged.

2.2 Machine learning techniques for malware detection and classification

The use of machine learning algorithms to address the problem of malicious software detection and classification has increased during the last decade. The success of these approaches has increased thanks to: (1) the rise of commercial feeds of malware; (2) the reduction in cost of computing power; and (3) the advances made in the machine learning field. Several methods have been applied based on features extracted from both static and dynamic analysis. An overview is provided in Ranvee and Hiray [23] and Gandotra et al. [7]. Instead of directly dealing with raw malware, machine learning solutions first extract features that provide an abstract view of the software program. Then the features extracted are used to train a model. The type of features can be broadly divided into two groups: (1) static features and (2) dynamic features.

Static features are those extracted from the binary of the malware without executing it. One of the most common types of features is byte n-grams. An n-gram is a contiguous sequence of n items from a given sequence of text. In the work of Tesauro et al. [29] they extracted a list of byte-sequence trigrams and used an artificial neural network to classify malware. Similar to byte-sequence n-grams, approaches in the literature have used opcodes n-grams [4,25]. An opcode (abbreviated from operation code) is the portion of machine language instruction that specifies the operation to be performed. In particular, D. Yuxin et al. [32] used deep belief networks as a feature extractor to generate deep features to represent executables from their opcodes sequences. In addition, D. Gibert et al. [9] presented a convolutional neural network architecture to learn a set of discriminative subsequences from assembly language instructions. Features can also be extracted from the Portable Executable (PE) Header. The PE Header contains information about the files themselves such as the associated dynamically linked libraries and the sections of the program and their sizes. For instance, Ravi and Manoharan et al. [5] proposed a malware detection system based on Windows API call sequences. In addition, entropy has proven to be an effective feature to detect malware. The entropy of a program, refers to the amount of disorder (uncertainty) or its statistical variation. Entropy has commonly been used to detect encrypted and packed executables because these programs often have higher entropy. For example, in the dataset studied by Lydia et al. [17], native executables had an average entropy of 5.099 while packed and encrypted executables had an average entropy of 6.801, 7.175, respectively. Moreover, Bat-Erdene et al. [3] used entropy analysis to classify packing algorithms of given unknown packed executables.

Furthermore, dynamic features are those extracted by executing and observing the behavior of malware. Malware API calls have been used to model the behavior of malware. In [24], Z. Salehi et al. constructed dynamic features based on the name of API calls and each argument and return value recorded in a controlled environment during runtime. M. Ghiasi et al. [8] presented a framework for malware detection based on the changes in register contents. In [28], C.Storlie et al. presented a spine logistic regression model for malware detection, trained on the instruction traces extracted dynamically from computer programs. Additionally, B.Anderson [2] constructed graphs from the instruction traces of executables and applied graph kernels to create and compare similarity matrices of different computer programs.

Besides, several visualization techniques have been proposed in the literature to help malware analysis. Sorokin et al. [26] visualized a given executable using its structural entropy, obtained by dividing a binary into non-overlapping chunks and computing the entropy for each chunk. This representation was exploited by M. Wojnowicz et al. [31] and D. Gibert et al. [10] for detecting and categorizing malware, respectively. Lastly, Nataraj et al. [20] used image processing techniques to classify malicious software according to its visualization as gray scale images. In their work, they extracted GIST features from the malware images and trained a k-nearest neighbour for classification. The approach presented in this paper is motivated by the experiments of Nataraj et al. and in its visualization of the malware's binary content. Next, it is provided a detailed description of the visualization technique.

2.3 Visualizing malware as an image

To visualize a malware sample as an image, every byte has to be interpreted as one pixel in an image. Then, the resulting array has to be organized as a 2-D array and visualized as a gray scale image. Values are in the range [0,255] (0:black, 255:white).

Fig. 1 shows the representation of samples of malware belonging to nine different families as gray scale images. It can be observed that images of software executables from a given family are similar visually while distinct from those belonging to a different family. The main benefit of visualizing a malicious executable as an image is that the different sections of a binary can be easily differentiated. In addition, malware authors only used to change a small part of the code

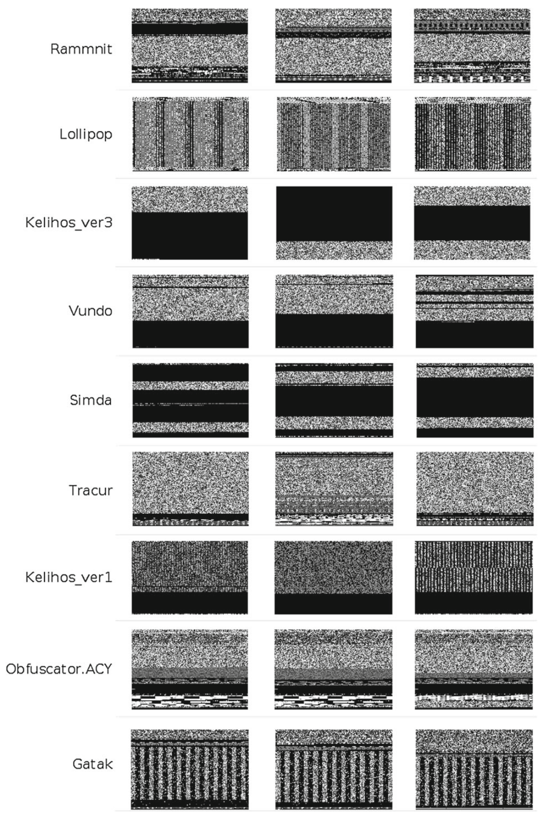


Fig. 1 Gray scale images of malicious software belonging to various families. Note that the images of malware belonging to the same family are similar while distinct from the images of malware from the rest of families

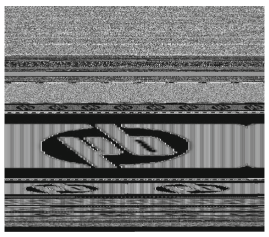


Fig. 2 Gray scale image representation of a malware sample containing a logo on their resources section

to produce new variants. Thus, if old malware is re-used to create new binaries the resulting ones would be very similar. Additionally, by representing malware as an image it is possible to detect the small changes while retaining the global structure of samples belonging to the same family.

In most cases, when observed in detail, one can notice several sections in the program, which usually have distinct feature patterns. Furthermore, the images stored in the resources section (.rsrc) of the PE file are also displayed (See Fig. 2). In addition, with this representation you can detect where zero-padding has been applied. Zero-padding is mainly used for block alignment but malware authors also use it to reduce the overall entropy of an executable.

2.4 Texture analysis and feature extraction

Traditional recognition approaches are composed of two stages: (1) feature extraction, transforming an observed signal into a robust representation, and (2) classification to model decision-making. Consequently, their performance relies heavily on the discriminative power of the features extracted. Hand-engineered feature extractors [1,19,20] gather relevant information about an input and eliminate irrelevant variabilities. In summary, Nataraj et al. [20] and Narayanan et al. [19] extracted GIST and PCA features, respectively, and Ahmadi et al. [1] extracted both Haralick and local binary pattern features. Below is a brief description of the aforesaid methods.

GIST descriptors [22]. Given an image, the process to compute a GIST descriptor is as follows. First, the image is convolved with 32 Gabor filters with 8 orientations and 4 scales. Second, each feature map is divided into 16 regions of 4×4 values, and then the feature values are averaged within each region. Last, the 16 averaged values of all feature maps are concatenated.

PCA [12] features. Principal Component Analysis is a statistical procedure used for dimensionality reduction. It uses an orthogonal transformation to convert a set of observations of n possible correlated variables into a set of values with m linearly uncorrelated variables named principal components, where $n \leq m$.

Haralick [11] features have been used for image classification for years. The features are calculated by constructing a co-occurrence matrix, and computed by using the equations defined by Haralick such as the angular second-moment, contrast, correlation, etc.

Local binary pattern [21] features. The local binary pattern (LBP) feature of a given pixel is computed as follows. First, an 8 bit binary array is initialized as 0. Then, each pixel is compared with its neighboring pixels in clockwise direction. If the value of the neighboring pixel is greater or equal to 1 is assigned to its corresponding position. This gives an 8 bit binary array with zeros and ones. The 8-bit binary pattern is converted to a decimal number and is stored in the corresponding pixel location in the LBP mask. This process is applied to all pixels in an image. Once all LBP values have been calculated, the mask is normalized, resulting in 256 features.

3 Convolutional neural networks for malware classification

Convolutional neural networks (CNNs) [15] are a type of artificial neural network inspired by the mammals visual cortex [13], whose receptive field comprises sub-regions layered over each other to cover the entire visual field. This type of networks has long been used in visual recognition tasks such as image classification or object detection due to its ability to automatically extract discriminant and local features from images.

The core of a convolutional neural network consists of one or more convolutional layers and one or more fully connected layers. In particular, convolutional layers act as detection filters for the presence of specific features or patterns present in the data. The first layers detect lower level features whereas later layers detect increasingly high level features. On the contrary, fully connected layers are used at the end of a CNN to combine all the specific features detected by the previous layers and determine a specific target output. Figure 3 presents an overview of our CNN architecture. The hyperparameters of the network had been selected using a grid search. The search was performed over the learning rate, the number of convolutional and fully-connected layers, and the size and number of kernels. The network architecture presented in this section correspond to the one that achieved better results during the evaluation.

The input of the network is an executable represented as a grayscale image $x^{w,h,d}$, where w and h are the width and the height of the image, respectively, and d is the depth ($d = 1$). Notice that in the work of Nataraj et al. [20] gray scale images had distinct widths and heights. This is because the size of each sample is different and with their preprocessing the width of the image was based on visual experiments and the height varied according to the file size. Thus, images had to be rescaled previously to feeding the convolutional neural network. The size of the resulting images has been adapted for every dataset during the experimentation in order to provide the best trade-off between accuracy and computational resources usage. Nevertheless, images have been resampled using the Lanzcos filter [30] as it provides the best compromise among several filters for multivariate interpolation.

The proposed neural network architecture has an input layer and three 4-stage feature extractors which learn hierarchical features through convolution, activation, pooling and normalization layers, as follows:

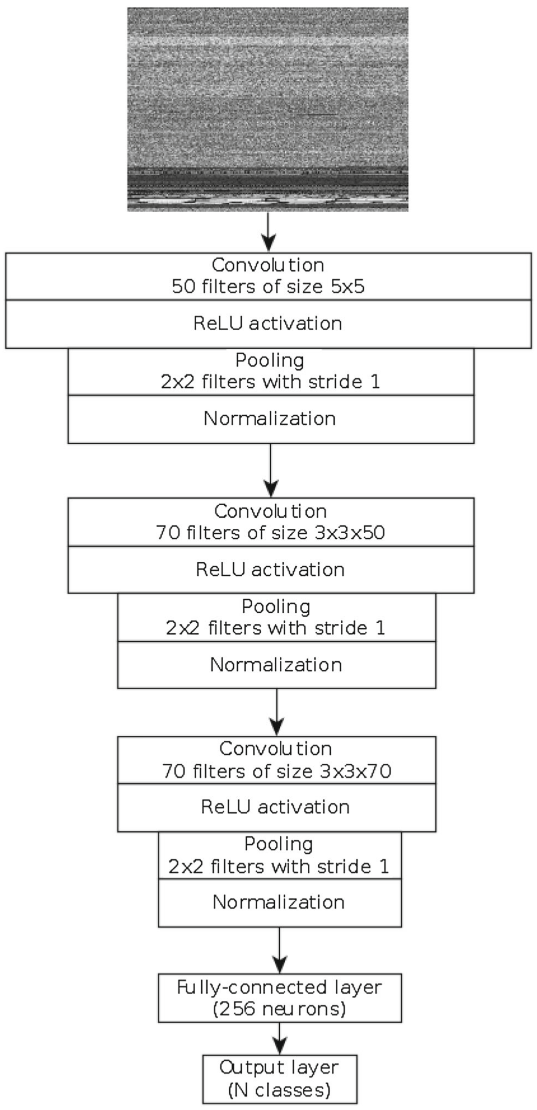


Fig. 3 Convolutional neural network for classification of malware represented as gray-scale images. It is composed by 3 convolutional layers followed by one fully-connected layer. The input of the network is a malicious program represented as a gray-scale image. The output of the network is the predicted class of the malware sample

Convolution is an operation that takes an input signal of size $w \times h \times d$ and a filter of size $k \times k \times d$, where $k \leq w, h$, and produces one output signal. The kernel slides over each value of the input signal, multiplies the corresponding entries of the input signal and the kernel and adds them up. Figure 4 presents three of the filters learned in the first convolutional layer. It can be observed that the features these kernels detect are high changes in the pixels intensities. In particular, the convolutional layers are composed of 50, 70 and 70 filters of size $5 \times 5 \times 1$, $3 \times 3 \times 50$ and $3 \times 3 \times 70$ for the first, second and third convolutional layers, respectively.

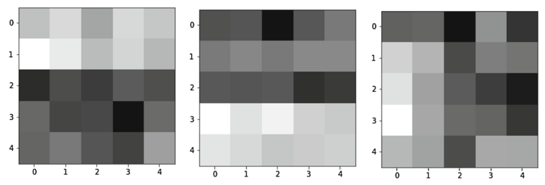


Fig. 4 Images of 3 filters randomly selected from the first convolutional layer

The activation function is used to signal distinct identification of likely features. Specifically, we used the ReLU [18] non-linear function, $y(x) = max(x, 0)$, in all activation layers.

The pooling operation reduces the spatial size of the features and provides some sort of robustness against noise and distortion. We applied max-pooling with filters of size $2 \times 2 \times 1$ with stride 1, which reduces the input signal by half, i.e. if the input signal's size is $128 \times 128 \times 1$, the output of applying max-pooling is an output signal of size $64 \times 64 \times 1$.

Normalization. The input values for different neurons in the layer are normalized using local response normalization [14] to inhibit and boost the neurons with relatively larger activations.

At the end of the extractor, the feature maps are flattened and combined to be used as input of the following fully-connected layer composed of 256 neurons. Afterwards, the output of the aforementioned layer is multiplied by the neurons of the output layer. Then, it is applied the softmax function to classify the binary program into its corresponding family. Notice that the number of neurons in the output layer depends on the number of families to classify: (i) the convolutional neural network trained to classify malware samples of the MalIMG dataset has 25 output neurons; (ii) the network trained to classify malicious software of the Microsoft Malware Classification Challenge dataset has 9 output neurons. In addition to normalization, to prevent overfitting we employed dropout [27], a regularization mechanism which randomly drops a proportion of p units during forward propagation and prevents the co-adaptation between neurons.

4 Evaluation

This section presents an empirical evaluation of the results obtained by our method in two datasets: the MalImg dataset [20], and the dataset provided by Microsoft for the Big Data Innovators Gathering Anti-Malware Prediction Challenge.

4.1 Experimental setup

The experiments were run on a single computer with the following hardware specifications:

- CPU: Intel i7-7700K
- Memory: 32 GB RAM
- GPU: Nvidia GTX 1080 Ti

To estimate the generalization performance of our approach we used K-fold cross validation. The dataset is divided into K equal size folds. Of the K subsamples, a single subsample is retained as the validation data for testing the model and the remaining subsamples are used as training data. This procedure is repeated as many times as there are folds, with each of the K folds used exactly once as the validation data.

Furthermore, to select the best model, additional evaluation metrics have been used: precision, recall and F1 score. This is because accuracy can be a misleading measure. Sometimes it may be desirable to select a model with a lower accuracy but with a greater predictive power on the problem (a.k.a. accuracy paradox). This occurs when there is as large class imbalance, where a model can predict the value of the majority class for all predictions and achieve a high classification accuracy while making mistakes on the minority or critical classes.

Precision (P) is the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p).

$$P = \frac{T_p}{T_p + F_p}.$$

Recall (R) is the number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n).

$$R = \frac{T_p}{T_p + F_n}.$$

The F1 score is the weighted average of precision, defined as following:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}.$$

Since our target task is a multi-class classification problem, we used an adapted version of the F1 score named macro-averaged F1 score, defined as the average of the individual F1 scores obtained for each class.

$$macro\_F_1 = \frac{1}{q} \sum_{i=1}^{q} F_1^i$$

Table 1 MalImg: Distribution of Samples

Family	Class ID	#samples
Adialer.C	1	125
Agent.FYI	2	116
Allaple.A	3	2949
Allaple.L	4	1591
Allueron.gen!J	5	198
Autorun.K	6	106
C2Lop.P	7	146
C2Lop.gen!g	8	200
Dialplatform.B	9	177
Dontovo.A	10	162
Fakerean	11	381
Instantaccess	12	431
Lolyda.AA1	13	213
Lolyda.AA2	14	184
Lolyda.AA3	15	123
Lolyda.AT	16	159
Malex.gen!J	17	136
Obfuscator.AD	18	142
Rbot!gen	19	158
Skintrim.N	20	80
Swizzor.gen!E	21	128
Swizzor.gen!I	22	132
VB.AT	23	408
Wintrim.BX	24	97
Yuner.A	25	800

where q is the number of classes in the dataset and F_1^i is the F1 score of class i. Macroaveraging gives equal weight to each class. Thus, large classes will not dominate over small classes.

4.2 MalImg dataset

The MalImg dataset was provided by Nataraj et al. [20] and consists of 9342 gray scale images of 25 malware families. It contains samples of malicious software packed with UPX from different families such as Yuner.A, VB.AT, Malex.gen!J, Autorun.K and Rbot!gen. Additionally, there are images of family variants like the C2Lop.p and the C2Lop.gen!g or the Swizzor.gen!I and the Swizzor.gen!E. For more details, see Table 1.

4.2.1 Results

In order to train the network we downsampled the images to a fixed size. The width and height of the new images were set to 256. A lower value did not retain all the important information (i.e. lost discriminative information about a family) while higher values only increased the computational time without increasing the overall accuracy. For instance, if images were downsampled to 28 $\times$ 28 pixels, samples from the Yuner.A and the Autorun.K families became indistinguishable from one another and the model failed to classify correctly any sample belonging to the Autorun.K family. On the contrary, if images were downsampled to 128 $\times$ 128 pixels, the model classified correctly 42.45% of samples belonging to the Autorun.K family. Finally, if images are downsampled to 256 $\times$ 256 pixels, the percentage of correctly classified samples belonging to the Autorun.K family increased to 80.02%. In consequence, the macro-averaged F1 score increased from 0.948 to 0.958. See Tables 2 and 3 for more information.

The overall classification accuracy achieved by our method for the 25 malware families is higher than the approach of Nataraj et al., 0.9848 and 0.9718, respectively. As can be observed in Table 3, there were two major sources of misclassifications. On the one hand, the model classified incorrectly 21 samples of the Autorun.K family as belonging to the Yuner. That is because both families are compressed with UPX and their corresponding executables visualized as gray scale images only differ in the .rsrc section. As can be seen in Fig. 5, samples from the Autorun.K and Yuner.A families are indistinguishable to the human eye. On the other hand, the model had problems classifying samples belonging to variants of the same family, such as Swizzer.gen!E and Swizzer.gen!I. In particular, it only classified correctly 71% and 62% of their samples. If family variants are combined as one, the overall accuracy and F1 score is increased to 0.993 and 0.984, respectively. Specifically, the following families were grouped in one.

- Allaple.A and Allaple.L as Allaple.
- C2Lop.P and C2Lop.gen!g as C2Lop.
- Lolyda.AA1, Lolyda.AA2, Lolyda.AA3 and Lolyda.AT as Lolyda.
- Swizzor.gen!E and Swizzor.gen!I as Swizzor.

As observed in Table 4, by grouping the samples of family variants into a single family, the number of samples incorrectly classified was reduced. In particular, the samples misclassified belonging to variants of the Swizzor family was reduced from 87 to 26.

The main advantage of our approach with respect to the method of Ahmadi et al. [20] is two fold. First, our classification time is not penalized by the size of the training set, as it is the k-nearest neighbor algorithm; i.e. k-nn is a non-parametric, lazy learning algorithm and it does not learn a discriminative function from the training data but it "memorizes" the training data instead. In consequence, when a new file is received it goes through all training instances. Second, as GIST extracted features are based on

Table 2 MalIMG dataset confusion matrix for 10-fold cross validation using images of 128 × 128 pixels

Family	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	Precision	Recall	F1 Score
1	122	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
2	0	116	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
3	0	0	2949	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
4	0	0	0	1591	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
5	0	0	0	0	198	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
6	0	0	0	0	0	45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	61	0.42	1.0	0.59
7	0	0	0	0	0	0	131	8	0	0	0	0	0	0	0	0	0	0	0	0	1	6	0	0	0	0.94	0.90	0.92
8	0	0	0	0	0	0	4	191	0	0	0	0	0	0	0	0	0	0	0	0	4	1	0	0	0	0.96	0.93	0.95
9	0	0	0	0	0	0	0	0	177	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
10	0	0	0	0	0	0	0	0	0	162	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
11	0	0	0	0	0	0	0	0	0	0	380	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
12	0	0	0	0	0	0	0	0	0	0	0	431	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
13	0	0	0	0	0	0	0	0	0	0	0	0	213	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
14	0	0	0	0	0	0	0	0	0	0	0	0	3	181	0	0	0	0	0	0	0	0	0	0	0	0.98	0.99	0.99
15	0	0	0	0	1	0	0	0	0	0	0	0	0	0	120	0	0	0	0	0	0	0	2	0	0	0.98	1.0	0.99
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	158	0	0	0	0	0	0	1	0	0	0.99	1.0	0.99
17	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	134	0	0	0	0	0	0	0	0	0.99	0.99	0.99
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	142	0	0	0	0	0	0	0	1.0	1.0	1.0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	156	0	1	0	0	0	1	0.99	1.0	0.99
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	80	0	0	0	0	0	1.0	1.0	1.0
21	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	86	38	0	0	0	0.67	0.72	0.69
22	0	0	0	0	0	0	11	2	0	0	0	0	0	0	0	0	0	0	0	0	27	92	0	0	0	0.69	0.67	0.68
23	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	405	0	0	0.99	0.99	0.99
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	0	1.0	1.0	1.0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	800	1.0	0.93	0.96
																									Macro-averaged F1 Score = 0,948			

Table 3 MalIMG dataset confusion matrix for 10-fold cross validation using images of 256 × 256 pixels

Family	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	Precision	Recall	F1 Score
1	122	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
2	0	116	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
3	0	0	2949	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
4	0	0	0	1591	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
5	0	0	0	0	198	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
6	0	0	0	0	0	85	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	21	0.80	1.0	0.89
7	0	0	0	0	0	0	138	6	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0.95	0.87	0.88
8	0	0	0	0	0	0	5	192	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0.96	0.94	0.95
9	0	0	0	0	0	0	0	0	177	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
10	0	0	0	0	0	0	0	0	0	162	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
11	0	0	3	0	0	0	1	0	0	0	376	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0.99	1.0	0.99
12	0	0	0	0	0	0	0	0	0	0	0	431	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
13	0	0	0	0	0	0	0	0	0	0	0	0	213	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
14	0	0	0	0	0	0	0	0	0	0	0	0	3	181	0	0	0	0	0	0	0	0	0	0	0	0.98	0.99	0.98
15	0	0	0	0	1	0	0	0	0	0	0	0	0	0	121	0	0	0	0	0	0	0	1	0	0	0.98	1.0	0.99
16	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	157	0	0	0	0	0	0	0	0	0	0.99	1.0	0.99
17	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	134	0	0	0	0	0	0	0	0	0.99	0.99	0.99
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	142	0	0	0	0	0	0	0	1.0	1.0	1.0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	157	0	1	0	0	0	0	0.99	1.0	0.99
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	80	0	0	0	0	0	1.0	1.0	1.0
21	0	0	0	0	0	0	1	5	0	0	0	0	0	0	0	0	0	0	0	0	91	31	0	0	0	0.71	0.71	0.71
22	0	0	0	0	0	0	14	2	0	0	0	0	0	0	0	0	0	0	0	0	32	82	1	0	1	0.62	0.71	0.66
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	406	0	0	0.99	0.99	0.99
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	0	1.0	1.0	1.0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	800	1.0	0.97	0.98

Macro-averaged F1 Score = 0.958

Table 4 MalIMG dataset confusion matrix for 10-fold cross validation using images of 256 × 256 pixels with family variants grouped into a single family. 1:Adialer.C; 2:Agent.FYI, 3:Allaple.A, Allaple.L; 4:Allueron.gen!J; 5:Autorun.K; 6:C2Lop.P, C2Lop.gen!g; 7:Dialplatform.B; 8:Dontovo.A; 9:Fakerean; 10:Instantaccess; 11:Lolyda,AA1; Lolyda.AA2; Lolyda.AA3, Lolyda.AT, 12:Malex.gen!J; 13:Obfuscator.AD; 14:Rbot!gen; 15:Skintrim.N; 16:Swizzor.gen!E, Swizzor.gen!I; 17:VB.AT; 18:Wintrim.BX; 19:Yuner.A

Family	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Precision	Recall	F1 Score
1	122	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
2	0	116	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
3	0	0	4540	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
4	0	0	0	198	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
5	0	0	0	0	98	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0, 92	1.0	0.96
6	0	0	0	0	0	330	0	0	0	0	0	0	0	0	0	0	16	0	0	0.95	0.92	0.93
7	0	0	0	0	0	0	177	0	0	0	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
8	0	0	0	0	0	0	0	162	0	0	0	0	0	0	0	0	0	0	0	1.0	0.99	0.99
9	0	0	1	0	0	2	0	0	377	0	0	0	0	0	0	0	1	0	0	0.99	1.0	0.99
10	0	0	0	0	0	0	0	0	0	431	0	0	0	0	0	0	0	0	0	1.0	1.0	1.0
11	0	1	0	1	0	0	0	1	0	0	676	0	0	0	0	0	0	0	0	0.99	0.99	0.99
12	0	0	2	0	0	0	0	0	0	0	1	133	0	0	0	0	0	0	0	0.98	0.99	0.98
13	0	0	0	0	0	0	0	0	0	0	0	0	142	0	0	0	0	0	0	1.0	1.0	1.0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	158	0	0	0	0	0	1.0	1.0	1.0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	80	0	0	0	0	1.0	1.0	1.0
16	0	0	0	0	0	25	0	0	0	0	0	0	0	0	0	234	1	0	0	0.90	0.99	0.94
17	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	2	405	0	0	0.99	0.96	0.97
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	0	1.0	1.0	1.0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	800	1.0	0.99	0.99
																				Macro-averaged F1 Score = 0.984		

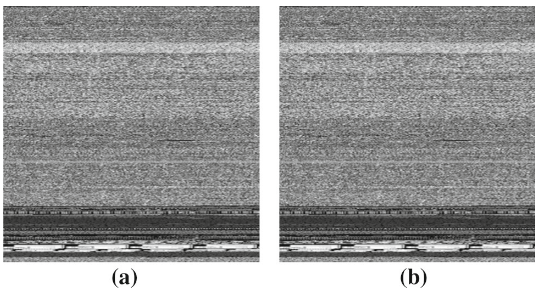


Fig. 5 Autorun.K and Yuner.A samples downsampled to 256 × 256 pixels. The left image corresponds to the gray-scale visualization of a malware sample belonging to the Autorun.K family while the image in the right belongs to the Yuner.A family. Notice that both images are indistinguishable to the human eye

Table 5 BIG 2015: Distribution of Samples

Family	Class ID	#samples
Ramnit	1	1541
Lollipop	2	2478
Kelihos_ver3	3	2942
Vundo	4	475
Simda	5	42
Tracur	6	751
Kelihos_ver1	7	398
Obfuscator.ACY	8	1228
Gatak	9	1013

the global structure of an image if an adversary knows the technique it could avoid detection by reallocating different parts of the code. On the contrary, the changes produced by the code reallocation technique might not produce such undesired effects in our approach because convolutional networks are able to learn features invariant to translation, i.e. detect patterns which may be displaced in space, through the convolution and max-pooling operations. The convolution operation provides equivariance to translation. Afterwards, the max-pooling operation returns the largest value in its receptive field. In consequence, if the location of this value is still within the receptive field, the output of the pooling operation would not be altered. As a result, the combination of both operations together provide invariance to translation.

4.3 Microsoft malware classification challenge

Microsoft provided a dataset composed of 21741 samples for the Big Data Innovators Gathering (BIG 2015) Anti-Malware Prediction Challenge, 10868 for training and 10873 for testing. Every program in the dataset has a file containing the hexadecimal representation of the malware's binary content and its corresponding assembly file. However, only the label for the samples belonging to the training dataset is provided. Table 5 shows the distribution of malware programs present in the training dataset.

4.3.1 Results

Similar to the MalImg dataset, we downsampled the gray scale images. In particular, images from the BIG dataset were downsampled to 128 × 128 pixels because greater width and height did not improve the performance of the classifier. In addition, we performed 5-fold and 10-fold cross validation to evaluate our model. Tables 6 and 7 show the confusion matrices obtained for 5-fold cross validation and 10-fold cross validation.

Table 8 presents the results obtained by state-of-the-art approaches in the literature that had extracted image-based features to classify malware from the BIG dataset. To sum up, Narayanan et al. [19] used PCA to extract the first 10, 12 and 52 principal components and classify malware using different machine learning classification algorithms. Moreover, Ahmadi et al. [1] extracted Haralick and local binary pattern features from images and trained an ensemble of trees for classification. Their approaches were evaluated using 5-fold and 10-fold cross validation, respectively. As can be observed, our method outperformed the rest of approaches in

Table 6 BIG 2015 dataset confusion matrix for 5-fold validation using images of 128 × 128 pixels

Family	1	2	3	4	5	6	7	8	9	Accuracy
1	1492	7	0	2	2	11	3	19	5	0.968
2	6	2424	0	1	3	10	0	3	31	0.978
3	1	0	2937	0	0	0	4	0	0	0.998
4	2	1	2	461	2	2	1	2	2	0.971
5	3	3	0	4	25	1	0	6	0	0.595
6	10	5	1	3	1	701	1	18	11	0.933
7	2	0	1	0	0	0	392	0	3	0.985
8	36	4	1	13	2	19	5	1144	4	0.932
9	2	6	0	1	0	2	2	2	998	0.985

Table 7 BIG 2015 dataset confusion matrix for 10-fold validation using images of 128 × 128 pixels

Family	1	2	3	4	5	6	7	8	9	Accuracy
1	1490	4	2	2	2	9	1	28	3	0.967
2	6	2440	0	0	1	7	0	8	16	0.985
3	0	1	2938	1	0	0	2	0	0	0.999
4	3	0	2	461	2	1	1	3	2	0.971
5	3	2	0	1	29	2	0	5	0	0.690
6	8	6	1	2	0	713	2	10	9	0.948
7	1	0	5	1	0	0	391	0	0	0.982
8	44	4	2	8	2	17	5	1138	8	0.923
9	2	2	0	0	0	6	2	5	996	0.983

Table 8 Performance comparison of various methods for classification of BIG 2015 training dataset. 1-nearest neighbor (1-NN). Support vector machines (SVM). Static feed-forward network (SFN1 & SFN2). Dynamic feed-forward network (DFN)

Method	5-fold accuracy	Macro-averaged F1 Score
Haralick features + XGBoost	0.955	–
LBP features + XGBoost	0.951	–
CNN	0.973	0.927
	10-fold accuracy	
12 PCA features + 1-NN	0.966	0.910
10 PCA features + SVM	0.946	0.864
52 PCA features + SFN1	0.956	0.884
52 PCA features + SFN2	0.942	0.849
52 PCA features + DFN	0.955	0.889
CNN	0.975	0.940

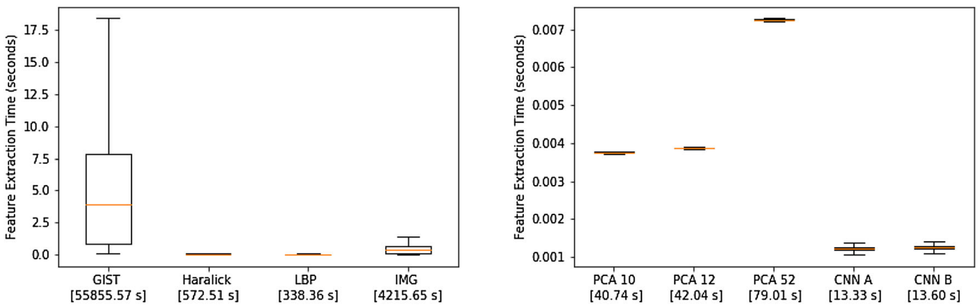


Fig. 6 The required time of feature extraction from the grayscale representation for each method. The time in brackets shows the total time of extraction for all training samples. LBP stands for local binary patterns. IMG denotes how much time it takes to transform a binary executable into a gray-scale image. PCA 10, PCA 12 and PCA 52 refer to the number of principal components used. CNN A refers to the time needed to extract the features by the convolutional network and CNN B refers to the time the networks needs to extract features and classify a sample

the literature, achieving 0.973 and 0.975 accuracy, for 5-fold and 10-fold cross validation, respectively. Furthermore, the average classification time of our approach is 0,001 seconds. Fig. 6 shows the computational time for every feature extraction method evaluated. The improvement of our method is equal to 99.98%, 98.47% and 96.06% with respect to the computational time needed to extract GIST, Haralick and local binary pattern features. Additionally, our method is 67.35%, 68.29% and 83.13% faster than the calculation of the 10, 12 and 52 principal components.

5 Conclusions

This paper presents a novel file agnostic deep learning system for classification of malware based on its visualization as gray-scale images. As far as we know, it is the first approach to apply deep learning to find patterns from malware's binary content represented as images. The proposed solution has a number of advantages that allow malicious programs to be detected in a real-time environment. Firstly, it is file agnostic and is based solely on the binary code of an executable. Secondly, the transformation of an executable into a gray-scale image is inexpensive. Thirdly, the prediction time is lower than the rest of approaches. Fourthly, it obtained greater classification accuracy than all previous methods in the literature that were based on the representation of malware as gray-scale images.

5.1 Limitations and future work

Despite the fact that our approach was able to outperform state-of-the-art methods in terms of accuracy and classification time, it has some issues that are directly related to the visualization of malware as grayscale images. Even though it can be seen that the visualization of malware programs belonging to the same family has similar patterns, this approach has problems with some samples that have been compressed or encrypted, which may have a completely different overall structure. For instance, the visualization of samples from the Autorun.K and Yuner.A families are almost equal. To deal with such cases, we suggest combining the features extracted by the convolutional neural network with hand-designed features as input to a machine learning model based on distinct types of file features [1].

Acknowledgements We would like to thank the Blueliv Labs team, especially Daniel Solís, and Àngel Puigventós for their support and the feedback provided during the development of this work. This work has been partially funded by the Spanish MICINN Projects TIN2014-53234-C2-2-R, TIN2015-71799-C2-2-P and by AGAUR DI-2016-091.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Ahmadi, M., Giacinto, G., Ulyanov, D., Semenov, S., Trofimov, M.: Novel feature extraction, selection and fusion for effective malware family classification. CoRR abs/1511.04317 (2015)
2. Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T.: Graph-based malware detection using dynamic analysis. J. Comput. Virol. **7**(4), 247–258 (2011). https://doi.org/10.1007/s11416-011-0152-x
3. Bat-Erdene, M., Park, H., Li, H., Lee, H., Choi, M.S.: Entropy analysis to classify unknown packing algorithms for malware detection. Int. J. Inf. Secur. **16**(3), 227–248 (2017)
4. Billar, D.: Opcodes as predictor for malware. Int. J. Electron. Secur. Digit. Forensics **1**, 156–168 (2007)
5. Chandrasekar Ravi, R.M.: Malware detection using windows API sequence and machine learning. Int. J. Comput. Appl. **43**, 12–16 (2012)
6. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2), 6:1–6:42 (2008). https://doi.org/10.1145/2089125.2089126
7. Gandotra, E., Bansal, D., Sofat, S.: Malware analysis and classification: a survey. J. Inf. Secur. **5**, 56–64 (2014)
8. Ghiasi, M., Sami, A., Salehi, Z.: Dynamic VSA: a framework for malware detection based on register contents. Eng. Appl. Artif. Intell. **44**, 111–122 (2015)
9. Gibert, D., Bejar, J., Mateu, C., Planes, J., Solis, D., Vicens, R.: Convolutional neural networks for classification of malware assembly code. In: International Conference of the Catalan Association for Artificial Intelligence, pp. 221–226 (2017). https://doi.org/10.3233/978-1-61499-806-8-221
10. Gibert, D., Mateu, C., Planes, J., Vicens, R.: Classification of malware by using structural entropy on convolutional neural networks. In: AAAI Conference on Artificial Intelligence (2018)
11. Haralick, R.M., Shanmugam, K., Dinstein, I.: Textural Features for Image Classification. IEEE Trans. Syst. Man Cybern. **SMC–3**(6), 610–621 (1973)
12. Hotelling, H.: Analysis of a complex of statistical variables into principal components. J. Educ. Psych. **24**, 417–441 (1933)
13. Hubel, D.H., Wiesel, T.N.: Receptive fields and functional architecture of monkey striate cortex. J. Physiol. (Lond.) **195**, 215–243 (1968)
14. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems, NIPS'12, pp. 1097–1105. Curran Associates Inc., USA (2012)
15. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. In: Proceedings of the IEEE, pp. 2278–2324 (1998)
16. LLC, M.: Mcafee labs threats report (2017). https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-jun-2017.pdf. Accessed 20 Sept 2017
17. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Secur. Anal. **5**, 40–45 (2007)
18. Nair, V., Hinton, G.E.: Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML'10, pp. 807–814. Omnipress, USA (2010)
19. Narayanan, B.N., Djaneye-Boundjou, O., Kebede, T.M.: Performance analysis of machine learning and pattern recognition algorithms for malware classification. In: Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS), 2016 IEEE National, pp. 338–342. IEEE (2016)
20. Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S.: Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security, VizSec '11, pp. 4:1–4:7. ACM, New York, NY, USA (2011)
21. Ojala, T., Pietikainen, M., Harwood, D.: Performance evaluation of texture measures with classification based on Kullback discrimination of distributions. In: Proceedings of the 12th IAPR International Conference on Pattern Recognition, 1994. Vol. 1—Conference A: Computer Vision amp; Image Processing, vol. 1 (1994)
22. Oliva, A., Torralba, A.: Modeling the shape of the scene: a holistic representation of the spatial envelope. Int. J. Comput. Vis. **42**(3), 145–175 (2001)
23. Ranvee, S., Hiray, S.: Comparative analysis of feature extraction methods of malware detection. Int. J. Comput. Appl. **120**, 1–7 (2015)

24. Salehi, Z., Sami, A., Ghiasi, M.: MAAR: robust features to detect malicious activity based on api calls, their arguments and return values. Eng. Appl. Artif. Intell. **59**, 93–102 (2017)
25. Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., Elovici, Y.: Detecting unknown malicious code by applying classification techniques on OpCode patterns. Secur. Inf. **1**(1), 1 (2012). https://doi.org/10.1186/2190-8532-1-1
26. Sorokin, I.: Comparing files using structural entropy. J. Comput. Virol. **7**(4), 259 (2011)
27. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. **15**, 1929–1958 (2014)
28. Storlie, C., Anderson, B., Vander Wiel, S., Quist, D., Hash, C., Brown, N.: Stochastic identification of malware with dynamic traces. Ann. Appl. Stat. **8**(1), 1–18 (2014). https://doi.org/10.1214/13-AOAS703
29. Tesauro, G., Kephart, J., Sorkin, G.B.: Neural networks for computer virus recognition. In: IEEE International Conference on Intelligence and Security Informatics, vol. 11 (1996)
30. Turkowski, K.: Filters for common resampling tasks. In: Glassner, A.S. (ed.) Graphics Gems, pp. 147–165. Academic Press Professional Inc., San Diego, CA (1990)
31. Wojnowicz, M., Chisholm, G., Wolff, M.: Suspiciously structured entropy: wavelet decomposition of software entropy reveals symptoms of malware in the energy spectrum. In: Florida Artificial Intelligence Research Society Conference (2016)
32. Yuxin, D., Siyi, Z.: Malware detection based on deep learning algorithm. Neural Comput. Appl. (2017). https://doi.org/10.1007/s00521-017-3077-6

Chapter 7

A Computer-Implemented Method, a System And a Computer Program For Identifying a Malicious File

The following invention, whose co-ownership belongs to Leap in Value, S.L and the University of Lleida, is part of a patent application currently under revision and not published (changes might occur during the revision process).

The full list of co-authors are: Daniel Solis, Gerard Cervelló, Àngel Puigventós, Daniel Gibert, Teresa Alsinet, Jordi Planes and Carles Mateu.

The present invention relates to a computer-implemented method, system and computer program for identifying a malicious file that combines different types of analysis, processes and procedures that allow detecting and classifying malicious files.

A computer-implemented method, a system and a computer program for identifying a malicious file

Field of the Invention

The present invention relates to a computer-implemented method, system and computer program for identifying a malicious file that combines different types of analysis, processes and procedures that allow detecting and classifying malicious files.

Background of the Invention

In the last few years, machine learning has been applied successfully to the task of malware detection and classification. One of the most used techniques is neural networks. Algorithms based on neural networks have recently achieved state-of-the-art results in a wide range of tasks. Examples of proposals based on neural networks can be found in the following patents: US9690938B1, US9705904B1 and US9495633B2.

However, the main limitation of neural networks is that it is difficult to obtain a rational explanation about the decisions they make.

A method for identifying malware file using multiple classifiers is known by US patent application, US2010192222A1. However, such method uses multiple classifiers including static and dynamic classifiers, and thus is unable to identify malware based only on static analysis.

Besides the above solutions, the patent EP2882159 discloses a computer implemented method of profiling cyber threats detected in a target environment, that comprises receiving, from a Security Information and Event Manager (SIEM) monitoring the target environment, alerts triggered by a detected potential cyber threat, and, for each alert: retrieving captured packet data related to the alert; extracting data pertaining to a set of attributes from captured packet data triggering the alert; applying fuzzy logic to data pertaining to one or more of the attributes to determine values for one or more output variables indicative of a level of an aspect of risk attributable to the cyber threat.

Apart from that, following the General Data Protection Regulation (GDPR) (Regulation (EU) 2016/679), automated individual decision-making, including profiling (Article 22) is contestable, similarly to the Data Protection Directive (Article 15). Citizens have rights to question and fight significant decisions that affect them that have been made on a solely algorithmic basis. In traditional deep learning (neural networks) systems, this right cannot be given.

New improved methods, systems and computer programs for identifying a malicious file are therefore needed.

Brief Description of the Invention

To that end, the present invention relates, in accordance with a first aspect, to a computer-implemented method for identifying a malicious file. The method comprises:

- performing a static analysis of a potentially malicious file to obtain a set of features that provide an abstract view of the malicious file (i.e. a view that reflects the obtained features from different points of view);

- performing a static machine learning classification process using as inputs said set of features, to obtain a preliminary classification output (i.e. a score); and

- performing a fuzzy inference procedure based on possibilistic logic, for reasoning under possibilistic uncertainty and disjunctive vague knowledge, for example originated by the rules created either by the system or by experts, using as input variables said set of features and said preliminary classification output, to generate an enhanced classification output that identifies the potentially malicious file as a malicious file or a benign file.

For an embodiment, the method comprises:

- performing several static analyses of different types of said potentially malicious file to obtain corresponding sets of features that provide abstract views of the malicious file;

- performing said static machine learning classification process using as inputs said sets of features, to obtain said preliminary classification output; and

- performing said fuzzy inference procedure based on possibilistic logic using as input variables said sets of features and the preliminary classification output.

For an alternative embodiment, the method comprises:

- performing several static analyses of different types of said potentially malicious file to obtain corresponding sets of features that provide abstract views of the malicious file;

- performing several static machine learning classification processes, each using as inputs at least one respective of said sets of features, to obtain corresponding several preliminary classification outputs (i.e. scores); and

- performing said fuzzy inference procedure based on possibilistic logic using as input variables said sets of features and said preliminary classification outputs.

According to an implementation of said alternative embodiment, the method comprises performing a further static machine learning classification process, using as inputs several or all of the above mentioned sets of features, to obtain a corresponding further preliminary classification output; and

- performing said fuzzy inference procedure based on possibilistic logic using as input variable also said further preliminary classification output.

For an embodiment, the above mentioned fuzzy inference procedure comprises a fuzzification process that converts the input variables into fuzzy variables.

For an implementation of said embodiment, the fuzzification process comprises deriving membership functions relating the input variables with output variables through membership degrees of values of the input variables in predefined fuzzy sets, and representing said membership functions with linguistic variables, said linguistic variables being said fuzzy variables.

In addition to the fuzzification process, according to an embodiment, the fuzzy inference procedure further comprises an inference decision-making process comprising firing fuzzy possibilistic rules with values of said linguistic variables for said input variables, to generate a fuzzy output that identifies the degree of belief that the potentially malicious file has to be a malicious file or a benign file.

Additionally, for an embodiment, the method of the first aspect of the present invention further comprises selecting which fuzzy possibilistic rules to fire in said inference decision-making process, based on at least said values of the linguistic variables for the input variables.

According to an embodiment, in addition to the fuzzification process and the inference decision-making process, the fuzzy inference procedure based on possibilistic logic further comprises a defuzzification process that converts the above mentioned fuzzy possibilistic output into a crisp output, wherein said crisp output constitutes the above mentioned enhanced classification output.

Depending on the embodiment, the above mentioned set or sets of features may comprise:

- the frequency of use of Application Programming Interfaces (API) and their function calls;
- the representation of an executable file as a stream of entropy values, where each value describes the amount of entropy over a small chunk of code in a specific location of the potentially malicious file;
- the sequence of assembly language instructions executed by a software program constituting the potentially malicious file, in particular, the operational codes of the machine language instructions;
- the representation of an executable file, constituting the potentially malicious file, as an image, where every byte is interpreted as one pixel

4

in the image, wherein the resulting array is organized as a 2-D array and visualized as a gray scale image; and/or

- applicable program characteristics, at least including alphanumeric strings occurring in the body of the software program constituting the potentially malicious file and the fields from the header of the potentially malicious file.

Two or more of the above indicated sets of features are comprised by the above identified as corresponding sets of features for implementations of the above described embodiments of the method of the first aspect of the invention for which the method comprises performing several static analyses.

In an embodiment, the fuzzy inference procedure based on possibilistic logic is based on a PGL+ algorithm. The proof method for PGL+ is complete and involves a semantical unification model of disjunctive fuzzy constants and three other inference patterns together with a deductive mechanism based on a modus ponens style.

In a particular embodiment, the PGL+ algorithm can comprise applying three algorithms sequentially: a first algorithm that extends the fuzzy possibilistic rules by means of implementing a first set of rules; a second algorithm that translates the fuzzy possibilistic rules into a semantically equivalent set of 1-weighted clauses by means of implemented a second set of rules; and a third algorithm that computes a maximum degree of possibilistic entailment of a goal from the equivalent set of 1-weighted clauses.

In an embodiment, the fuzzy inference procedure based on possibilistic logic comprises executing the following formulas in the form (A, c), where A is a Horn clause (fact or rule) with disjunctive fuzzy constants and c is a degree in the unit interval [0,1] which denotes a lower bound on the belief on A in terms of necessity measures. Every fact and rule is attached with a degree of belief or weight in the real interval [0, 1] that denotes a lower bound on the belief on the fact and rule in terms of necessity measures. So, those facts and rules that are demonstrated to be key for the decision system have a higher weight, and facts and rules not so useful in the decision system have a lower weight. The rules created by the system can have a higher degree of belief than the rules created by a human, or vice versa. For example, the system may create rules of the following form:

- Rule 1: IF (entropy(.text) is “high” OR entropy(.text) is “very_high” OR entropy(.text) is “extreme”) AND (call(“CryptAcquireContext”) OR call(“CryptEncrypt”) OR call(“CryptReleaseContext”)) THEN file 100 is encrypted with a degree of belief of 1.0

● Rule 2: IF entropy(file) is "very_high" AND ML_score(ENTROPY) is "high" THEN file 100 is encrypted with a degree of belief of at least 0.9.

Additionally, the facts can have different degrees of belief depending on the source of the information. For instance, file management API functions (e.g. CopyFile, CreateFile, EncryptFile, etc.) can have a higher belief degree than networking APIs (e.g.HttpCreateServerSession,DnsAcquireContextHandle, RpcStringBindingCompose, etc.).

In some embodiments, the machine learning models can be enhanced by further using Reinforcement Learning methods. Reinforcement Learning (RL) is a set of techniques that allow to solve problems in highly uncertain or almost unknown domains. The method can use machine learning to select the most relevant features, using RL guided methods to derive the future reward (i.e. accuracy) of using such feature. After several iterations (training process), the machine learning technique will be able to use a Q-Table (rewards table) of the RL method to accurately predict which feature and split set use for prediction, thus creating a quasi-optimal DT from which to derive the rules for the system. This last module makes the system keep learning from new threats, a key aspect when it comes to cybersecurity.

In a second aspect, the present invention also relates to a system for identifying a malicious file, the system comprising one or more computing entities adapted to perform the steps of the method of the first aspect of the invention for all its embodiments, said one or more computing entities including at least the following modules operatively connected to each other:

- a preprocessing computing module configured and arranged to perform a static analysis of a potentially malicious file to obtain a set of features that provide an abstract view of the malicious file;

- a machine learning module configured and arranged to perform a static machine learning classification process using as inputs said set of features, to obtain a preliminary classification output; and

- a fuzzy inference module configured and arranged to perform a fuzzy inference procedure based on possibilistic logic using as input variables said set of features and said preliminary classification output, to generate an enhanced possibilistic classification output that identifies the potentially malicious file as a malicious file or a benign file.

Other embodiments of the invention that are disclosed herein also include software programs to perform the method embodiment steps and operations summarized above and disclosed in detail below. More particularly, a computer program product is one embodiment that has a computer-readable medium including computer program instructions encoded thereon that when executed on at least one processor in a computer system causes the processor to perform the operations indicated herein as embodiments of the invention.

With the present invention, according to its three aspects, the limitations mentioned above associated to the prior art methods are addressed by aggregating and combining multiple static features and the output of preferably multiple static classifiers to infer the maliciousness of a file based on a set of fuzzy rules. These rules might be inferred using the knowledge of cyber security experts or using any machine learning technique.

With the present invention, the user has access to all the decisions taken in order to decide if a file is malicious. Additionally, an expert user can create additional rules, or modify the ones created by the method, system or computer program of the present invention.

Brief Description of the Figures

The previous and other advantages and features will be more fully understood from the following detailed description of embodiments, with reference to the attached figures, which must be considered in an illustrative and non-limiting manner, in which:

Fig. 1 schematically shows the system of the second aspect of the invention, for an embodiment, depicting its main modules.

Fig. 2 is an Entropy versus Chunk diagram showing an example of a static analysis of the method of the first aspect of the invention to provide a set of features of an abstract view of an executable file in the form of a stream of entropy values of a structural entropy, computed using the Shannon's formula, of the executable file, according to an embodiment, by means of the pre-processing computing module of the system of the second aspect of the invention.

Fig. 3 shows gray scale images constituting sets of features obtained by respective static analyses of the method of the first aspect of the invention, representing abstracts views of different malware files (Rammnit, Lollipop, Kelihos_ver3), according to corresponding embodiments, by means of the pre-processing computing module of the system of the second aspect of the invention.

Fig. 4 schematically shows an overview of a preprocessing module of the system of the second aspect of the present invention, decomposed into five components for performing five corresponding static analyses, including those associate to the embodiments of Figures 2 and 3, to an executable file.

Fig. 5 schematically shows the system of the second aspect of the invention, for an embodiment for which the machine learning module includes one submodule, or static classifier, per each set of features provided by a respective static analyser of the pre-processing module.

Fig. 6 schematically shows the system of the second aspect of the invention, for an embodiment for which the machine learning module includes only one static classifier that includes as inputs all the set of features provided by all the static analysers of the pre-processing module.

Fig. 7 schematically shows the system of the second aspect of the invention, for an embodiment that differs to that of Figure 5 in that the machine learning module comprises, in addition, a further submodule that includes as inputs all the set of features provided by all the static analyzers of the pre-processing module.

Fig. 8 schematically shows the system of the second aspect of the present invention, for an embodiment, including the preprocessing module, the machine learning module, and a fuzzy inference module decomposed in several functional blocks.

Fig. 9 is a diagram that shows the membership function of some fuzzy subsets of sets of features obtained with a static analyzer, particularly of entropy values, for an embodiment of the fuzzification process performed according to the method of the first aspect of the invention, by means of the fuzzy inference module of the system of the second aspect of the invention.

Fig. 10 graphically shows the membership function of some fuzzy subsets associated to scores obtained from a machine learning process applied on the sets of features of Figure 9, for an embodiment of the fuzzification process performed according to the method of the first aspect of the invention, by means of the fuzzy inference module of the system of the second aspect of the invention.

Fig. 11 is a diagram that shows membership functions of scores obtained at the fuzzification process, as part of a defuzzification process to obtain crisp values, according to an embodiment of the method and system of the present invention.

Detailed Description of Preferred Embodiments

Fig. 1 shows an embodiment of the system of the second aspect of the present invention. As seen in the figure, the proposed system includes three components: a preprocessing module 110; a machine learning module 120 and a fuzzy interference module 130.

The preprocessing module 110 is responsible of the extraction of features/characteristics 111 of a given software program 100 (also termed file or executable). The machine learning module 120, which can be composed of one or more machine learning modules 121, given one or more of said extracted features/characteristics 111, can output a score 123 (i.e. a preliminary classification output) indicating the maliciousness of the software program 100 with respect to the input features 111. The fuzzy inference module 130 is responsible of performing inference upon fuzzy rules and given facts, i.e. characteristics of the software program 100 and the output scores 123 of the machine learning methods implemented by the machine learning modules 121, to derive a reasonable output or conclusion 140 (i.e. an enhanced classification output), that is whether a file 100 is malicious or not. Notice that the invention might be applied to classifying malware into families without needing to make any significant modification.

The terms “given facts” refer herein to the facts, data and input information of the fuzzy inference module 130. These data are the features extracted by the pre-processing 110 and machine learning 120 modules.

Depending on the embodiment, the preprocessing 110 and machine learning 120 modules are independent module or are comprised by a common feature extraction module.

In an embodiment of the proposed method, a file 100 is received at a client or server computer, and then a static type of analysis of the file 100, i.e. without executing the file, is initiated. This static analysis is performed by the preprocessing module 110, which processes the file 100 and generates an abstract view thereof. This abstract view might be represented by sets of features 111.

These features 111 are used as input to one or more static classifiers 122, each implemented in one of the cited machine learning submodules 121. The output 123 of each machine learning classifier 122 is a value in the range [0, 1]. A value close to 0 means that the executable 100 does not contain suspicious/malicious indicators with regard of a specific group of features 111, otherwise, values close to 1 indicates maliciousness. Any machine learning method can be used as classifier. For instance, neural networks, support vector machines or decision trees.

The fuzzy inference module 130 receives as input at least one or more features 111 extracted by the preprocessing module 110 and the output 123 of one or more static classifiers 122, and performs the inference procedure upon the rules and given facts to derive a reasonable output or conclusion 140, that is whether a file is malicious or not.

Preprocessing module description

The preprocessing module 110 is responsible of the feature extraction process. It analyses the software program 100 with static techniques (i.e. the program 100 is not executed). It extracts various characteristics from the programs' 100 syntax and semantic.

The software program 100 can take varying formats including, but not limited to, Portable Executable (PE), Disk Operating System (DOS) executable files, New Executable (NE) files, Linear Executable (LE) files, Executable and Linkable Format (ELF) files, JAVA Archive (JAR) files, and SHOCKWAVE/FLASH (SWF) files.

While the present embodiments describe the application of the present invention to Portable Executable (PE) format files, it will be appreciated that the methodologies described herein can be applied to other types of structured files as the ones previously mentioned.

Given a software program 100, the preprocessing module 110 extracts at least one, but not limited to, of the following sets or subsets (groups) of features:

1. API function calls,
2. Assembly language instructions,
3. Structural entropy,
4. Image representation of the binary program,
5. Miscellaneous features.

The use of every set of features is explained in the following sections.

1. API function calls

The frequency of use of Application Programming Interfaces (API) and their function calls are regarded as very important features. Literature has shown that API call can be explored to model the program behavior.

API functions and system calls are related with services provided by operating systems. It supports various key operations such as networks, security, system services, file managements, and so on. In addition, they include various functions for utilizing system resources, such as memory, file system, network or graphics.

There is no other way for software to access system resources that are managed by operating systems without using API functions or system calls, and thus, API function calls can provide key information to represent the behavior of the software 100. In consequence, every API function and system call had been associated a feature. The feature range is [0, 1]; 0 (or False) if the API function or system call hasn't been called by the program; 1 (or True) otherwise. Alternatively, one can count how many times every API function has been called during the execution of the program.

Because many malware programs are packed, leaving only the stub of the import table or perhaps even no import table at all, the malware classifier will search for the name of the dynamic link library or function in the body of the suspected malware (by disassembling the executable 100).

2. Structural entropy

An executable file 100 is represented as a stream of entropy values, where each value describes the amount of entropy over a small chunk of code in a specific location of the file 100. For each chunk of code, the entropy is computed using the Shannon's formula. There exists empirical evidence that the entropy time series from a given family are similar and distinct from those belonging to a different family. This is the result of reusing the code to create new malware variants. In consequence, the structural entropy of an executable 100 can be used to detect whether it is benign or malware and to classify it into their corresponding family.

The diagram of Fig. 2 shows an example of the above mentioned computed entropy versus chunk, for an embodiment.

3. Assembly language instructions

A software program 100 is disassembled (IDA Pro, Radare2, Capstone, etc.) and its sequence of assembly language instructions is extracted for further analysis. In particular, the operational codes of the machine language instruction were extracted. An operational code (opcode) is the portion of a machine language instruction that specifies the operation to be performed: arithmetic or data manipulation, logical operation or program control. Opcodes reveal significant statistical differences between malware and legitimate software. Thus, a sequence of opcodes can be extracted and then used to detect whether a file 100 either is benign or malware (opcodes = [mov, pop, push, add, sub, mul, etc.]).

4. Gray scale image-based visualization of the program hexadecimal content

To visualize a software program 100 as an image, every byte has to be interpreted as one pixel in an image. Then, the resulting array has to be organized as a 2-D array and visualized as a gray scale image, as shown in Fig. 3.

The main benefit of visualizing a malicious executable 100 as an image is that the different sections of a binary can be easily differentiated. In addition, malware authors only used to change a small part of the code to produce new variants. Thus, if old malware is re-used to create new binaries the resulting ones would be very similar.

Additionally, by representing malware as an image it is possible to detect the small changes while retaining the global structure of samples.

This technique for malware visualization was first presented in the work of Nataraj et al. named “Malware Images. Visualization and Automatic Classification”.

5. <u>Miscellaneous features</u>

This group of features comprises hand-crafted features defined by cyber security experts. For instance, the size in bytes and the entropy of the sections of the Portable Executable file, the frequency of use of the registers, the frequency of a set of keywords from an executable, the attributes of the headers of the Portable Executable, among others.

Fig. 4 presents an overview of the preprocessing module 110 decomposed into the five aforementioned components.

<u>Machine learning module description</u>

The use of machine learning algorithms to address the problem of malicious software detection and classification has increased during the last decade. Instead of directly dealing with raw malware, machine learning solutions first have to extract features that provide an abstract view of the software. Then the features extracted can be used to feed one machine learning method at least.

In one embodiment, shown in Fig. 5, the system of the second aspect of the invention comprises and uses multiple machine learning submodules 121, each receiving as inputs the output provided by a respective of the static classifiers of the preprocessing module 110.

The system receives a file 100 (such as an executable file) at a client or server computer. The preprocessing module 110 is responsible of extracting a set of features 111 from the file 100, by means of the static classifiers. These features 111 are used as input to the machine learning submodules 121. The system has at least as many machine learning submodules 121 as groups of features.

The output of each machine learning submodule 121 is a value in the range [0,1]. A value close to 0 means that the executable 100 do not contain suspicious/malicious indicators with regard of a specific group of features 111, otherwise, the value will be close to 1. Any machine learning method can be used as static classifier. For instance, neural networks, support vector machines or decision trees.

In an embodiment, a feed-forward neural network with at least three layers: (1) an input layer, (2) one fully-connected layer and (3) an output layer can be used. The input layer has size equal to the length of the feature vector. The output layer has only one neuron and outputs the probability of an executable of being malicious or not. Additionally, a dropout after every fully-connected layer can be added.

Alternatively, depending on the nature of the data, i.e. images and time series, it might be useful to use a convolutional or a recurrent neural network as a classifier. In particular, convolutional neural networks have achieved great success in image and time series related classification tasks. Convolutional neural networks consist of a sequence of convolutional layers, the output of which is connected only to local regions in the input. This structure allows learning filters able to recognize specific patterns in the input data. The convolutional network can be composed by 5 or more layers: (1) the input layer, (2) one convolutional layer, (3) one pooling layer, (4) one fully-connected layer and (5) the output layer.

In particular, the following embodiments present concrete implementations of static classifiers for each group of features.

1. Static classifier embodiment 1: API function calls.

In some implementations, the behavior of an executable file can be modelled by their use of the API functions. In those implementations, the executable file is disassembled to analyze and extract the API function calls it performs. In some implementations, every API function and system call has associated a feature. The feature range is [0,1]; 0 (or False) if the API function or system call hasn't been called by the program; 1 (or True) otherwise. Alternatively, one can count how many times every API function has been called during the execution of the program. In other implementations, only a subset of the available API function calls a program can execute is considered. That is because the number of API function calls a program can execute is huge and some of them are irrelevant to model the program's behavior. Thus, in some implementations only a subset of the available API function calls is considered. To select which are the most informative API function calls to record, any feature selection technique might be considered.

A feed-forward network can be utilized to analyze the API functions invoked by a computer program. The feed-forward network may have one or more hidden layers followed by an output layer, which generates a classification for the file (e.g. malicious or benign). The classification of the file can be provided at an output of the convolutional neural network.

2. Static classifier embodiment 2: Structural entropy.

In some implementations, an executable file can be represented as a stream of entropy values, where each value describes the amount of entropy over a small chunk of code in a specific location of the file. For each chunk of code, the entropy is computed using the Shannon's formula. A convolutional neural network can be utilized to analyze the stream of entropy values by applying a plurality of kernels to detect certain patterns in the variation between entropy values of adjacent chunks.

The convolutional network can detect malicious executables by providing a classification of the disassembled binary file (maliciousness score: [0,1]). The convolutional neural network may include a convolutional layer, a pooling layer, a fully connected layer and an output layer. The convolutional neural network can be configured to process streams variable in length. As such, one or more techniques can be applied to generate fixed length representations of the entropy values. In some implementations, the first convolutional layer can be configured to process the stream of entropy values by applying a plurality of kernels K1,1, K1,2,..., K1,x to the entropy values. Each kernel applied to the first convolutional layer can be configured to detect changes between entropy values of adjacent chunks in a file. According to some implementations, each kernel applied to the first convolutional layer can be adapted to detect a specific sequence of entropy values, having w values.

Although the convolutional neural network has been indicated as comprising 3 convolutional layers, it should be appreciated that the convolutional neural network can include less or more convolutional layers.

In some implementations, the pooling layer can be configured to further process the output from a preceding convolutional layer by compressing (e.g. subsampling or down sampling) the output from the preceding convolution layer. The pooling layer can compress the output by applying one or more pooling functions, including for example a maximum pooling functions.

In some implementations, the output of the pooling layer can be further processed by the one or more fully connected layers and the output layer in order to generate a classification for the file (e.g. malicious or benign). The classification of the file can be provided at an output of the convolutional neural network.

3. Static classifier embodiment 3: Assembly language instructions.

In some implementations, a binary file can be disassembled thereby forming a discernible sequence of instructions having one or more identifying features (e.g. instruction mnemonics). A convolutional neural network (CNN) can be utilized to analyze the disassembled binary file by applying a plurality of kernels (filters) adapted to detect certain sequences of instructions in the disassembled file. The convolutional network can detect malicious executables by providing a classification of the disassembled binary file (maliciousness score: [0,1]).

14

The convolutional neural network may include a convolutional layer, a pooling layer, a fully connected layer and an output layer. The convolutional neural network can be configured to process a sequence of instructions that are variable in length. As such, one or more techniques can be applied to generate fixed length representations of the instructions. Moreover, the fixed length of instructions can be encoded in a way the network understands their meaning. Remember that neural networks cannot deal with not numerical features. Thus, mnemonics are encoded using one-hot vector representations. Afterwards, each one-hot vector is represented as a word embedding, that is a vector of real numbers. This vector representation of the opcodes can be generated during the training phase of the convolutional network or using any other approach such as neural probabilistic language models, i.e. SkipGram model, Word2Vec model, Recurrent Neural Network models, etc.

In some implementations, the first convolutional layer can be configured to process the encoded fixed mnemonics representations by applying a plurality of kernels K1,1, K1,2,... K1,x to the encoded fixed mnemonics representations. Each kernel applied at the first convolutional layer can be configured to detect a specific sequence of instructions. According to some implementations, each kernel applied to the first convolutional layer can be adapted to detect a sequence having a number of instructions. That is, kernels K can be adapted to detect instances where a number of instructions appear in a certain order. For example, kernel K1,1 can be adapted to detect the instruction sequence [cmp, jne, dec] while kernel K1,2 can be adapted to detect the instruction set [dec, mov, jmp]. The size of each kernel (w, the number of instructions) corresponds to the window size of the first convolutional layer.

In some implementations, the convolutional layer may have kernels of different size. For instance, one kernel may be adapted to detect the instruction sequence [dec, mov, jmp] while another kernel may be adapted to detect the instruction set [dec, mov, jmp, pull, sub].

Although the convolutional neural network is shown to include one convolutional layer, it should be appreciated that the convolutional neural network can include a different number of convolutional layers. For instance, the convolutional neural network can include more convolutional layers such as 2.

Thus, in some implementations, the kernels K2,1, K2,2, …. K,2,x applied to the second convolutional layer can be adapted to detect specific sequences of two or more of the sequences of instructions detected at the first convolutional layer. Consequently, the second convolutional layer would generate increasingly abstract representations of the sequence of instructions from the disassembled binary file.

In some implementations, the pooling layer can be configured to further process the output from a preceding convolutional layer by compressing (e.g. subsampling or down sampling) the output from the preceding convolution layer. The pooling layer can compress the output by applying one or more pooling functions, including for example a maximum pooling functions.

In some implementations, the output of the pooling layer can be further processed by the one or more fully connected layers and the output layer in order to generate a classification for the disassembled binary file (e.g. malicious or benign). The classification of the disassembled binary file can be provided at an output of the convolutional neural network.

4. Static classifier embodiment 4: Image-based representation of malware's hexadecimal content.

In some implementations, a software program can be visualized as an image, where every byte interpreted as one pixel in the image. Then, the resulting array is organized as a 2-D array and visualized as a gray scale image. Approaches such as convolutional neural networks can yield classifiers that can learn to extract features that are at least as effective as human-engineered features. A convolutional neural network implementation to extract features can advantageously make use of the connectivity structure between feature maps to extract local and invariant features from an image. A convolutional neural network (CNN) can be utilized to analyze the file by applying a plurality of kernels (filters) adapted to detect certain local and invariant patterns in the pixels of the representation of the software program as a gray-scale image. The convolutional network can detect malicious executables by providing a classification of the disassembled binary file (maliciousness score: [0,1]).

The convolutional neural network at least may include a convolutional layer, a pooling layer, a fully connected layer and an output layer. In some implementations, it may include more than one convolutional, pooling and fully connected layers. According to some implementations, each kernel applied to the first convolutional layer can be adapted to detect a pattern in the pixels of the image having w x h size, where w is the width and h is the height of the kernel. Subsequent convolutional layers detect increasingly abstract features.

In some implementations, the pooling layer can be configured to further process the output from a preceding convolutional layer by compressing (e.g. subsampling or down sampling) the output from the preceding convolution layer. The pooling layer can compress the output by applying one or more pooling functions, including for example the maximum pooling function.

In some implementations, the output of the pooling layer can be further processed by the one or more fully connected layers and the output layer in order to generate a classification for the file (e.g. malicious or benign). The classification of the file can be provided at an output of the convolutional neural network.

5. Static classifier embodiment 5: Miscellaneous features.

In any embodiment of the invention, the so-called “miscellaneous” features include those applicable software characteristics. These characteristics at least include the keywords occurring in the software of the program and the fields of the header of a file in any format. Other type of features may also be used.

Next table illustrates the fields of the header of a file in portable executable format. For example, these fields are: MajorLinkedVersion, MinorLinkerVersion, SizeOfCode, SizeOfInitializedData, etc. Shown is relevant information that contains suitable characteristics to use as features. These characteristics are specific to information of a Portable Executable file header, but other file types will have other relevant header information and characteristics.

Feature Name Feature Value	Feature Name Feature Value
MajorLinkerVersion The linker minor version number.	MajorLinkerVersion The linker minor version number.
MinorLinkerVersion The linker minor version number.	MinorLinkerVersion The linker minor version number.
SizeOfCode The size of the code (text) section, or the sum of all	SizeOfCode The size of the code (text) section, or the sum of all
SizeOfInitializedData	The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
SizeOfUninitializedData	The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections

In another embodiment, shown in Fig. 6, the preprocessing module 110 is responsible of extracting a set of informative features 111 from the file 100. These features 111 are then aggregated and fed as input to a common static classifier 122, which will determine whether the file 100 is malicious or not. The input of the static classifier 123 is the features 111 from the distinct groups extracted by the preprocessing module 110. The output 123 of the static classifier 122 is a value in the range [0,1]. A value close to 0 means that the executable 100 does not contain suspicious/malicious indicators with regard to the features 111, otherwise, the value will be close to 1. Any machine learning method can be used as classifier. For instance, neural networks, support vector machines or decision trees.

In another embodiment, showed in Fig. 7, the preprocessing module 110 is responsible of extracting a set of informative features 111 from the file 100. These features 111 are used as input to static classifiers. The system has as many static classifiers as set of features and, in contrast to the embodiment of Fig. 5, a further static classifier that would aggregate and use the features of all groups as input. The output 123 of each machine learning classifier 122 is a value in the range [0, 1]. A value close to 0 means that the executable 100 do not contain suspicious/malicious indicators with regard of a specific group of features 111, otherwise, the value will be close to 1. Any machine learning method can be used as classifier. For instance, neural networks, support vector machines or decision trees.

<u>Fuzzy inference module description</u>

The last component of the malware detection system is the fuzzy inference engine 130. Its aim is to define a set of fuzzy rules of whether an executable is malicious based on the output of the machine learning methods and the features extracted by the preprocessing module.

This component 130 performs the following steps:

- receives one or more input values and generates an array of values each representing a membership degree of a respective input value in a predefined fuzzy set (fuzzification);
- combines the membership values on the premise part to get firing strength (degree of fulfilment) of each rule;
- generates the qualified consequent part (either fuzzy or crisp) of each rule depending on the firing strength;
- aggregates the qualified consequent part to produce a crisp output (defuzzification).

The fuzzy inference module 130 can be decomposed into functional blocks, as depicted in Fig. 8, and described below in detail.

<u>A/ Fuzzification:</u>

First, the input values of the system have to be converted to fuzzy variables. This process is named fuzzification 131. Fuzzification 131 involves two processes: derive the membership functions for input and output variables, and represent them with linguistic variables. (Given two inputs, x1 and y1, determine the degree to which input variables belong to each of the appropriate fuzzy sets.)

The input values are two-fold: a feature vector of program characteristics named F, of size |F|, where $F_i \in F$ corresponds to the value of the i-th feature of the program 100. This feature vector is extracted by the preprocessing module 110; and a score vector containing the output scores 123 of the machine learning algorithms named S of size |S|, where |S| is equal to the number of distinct algorithms that have been applied to predict the maliciousness of the program based on distinct groups of features. This score vector is generated by the machine learning module 120.

To illustrate the process of fuzzification 131, a vector of only two features containing the entropy of the .text section of a Portable Executable 100 and the score 123 generated by a machine learning algorithm will be considered as input.

The entropy of a bytes sequence refers to the amount of disorder (uncertainty) or its statistical variation. The entropy value ranges from 0 to 8. If occurrences of all values are the same, the entropy will be largest. On the contrary, if certain byte values occur with high probabilities, the entropy value will be smaller. According to studies, the entropy of plain text, native executables, packed executables and encrypted executables tend to differ greatly. In consequence, the [0,8] range can be further divided into at least six sub-ranges or subsets, which are:

- VERY LOW: From 0 to 4.328 entropy
- LOW: From 4.066 to 5.030 entropy
- MEDIUM: From 4.629 to 6.369 entropy
- HIGH: From 6.219 to 7.267 entropy
- VERY HIGH: From 6.838 to 7.312
- EXTREME: From 7.215 to 8.0

The membership function of these subsets is shown in Fig. 9. To make thing simple, a trapezoidal waveform is utilized for this type of membership function. For instance, 4.0 entropy will belong to “very low” to 0.6 degree and to “low” to 0.4 degree.

The score 123 of a given machine learning classifier 122 is a value in the range [0, 1]. A value close to 0 means that the executable 100 do not contain suspicious/malicious indicators with regard of a specific group of features 111, and it is a low threat, otherwise, the value will be close to 1. This score 123 can be further divided into at least three sub-ranges or subsets which are:

- LOW: From 0.0 to 0.5
- MEDIUM: From 0.2 to 0.75
- HIGH: From 5.0 to 1.0

The membership function of these subsets is shown in Fig. 10. For example, 0.4 score belongs to “LOW” to 0.38 degree and to “MEDIUM” to 1.0 degree.

In the implementation of the current subject matter, the fuzzy sets corresponding to all machine learning classifiers 122 are defined using the same membership functions for simplicity purposes. However, this is not a constraint and they might be defined with different membership functions and fuzzy sets.

<u>B/ Knowledge Base:</u>

The rule base and the database of the invention are jointly referred to as the knowledge base 132. The knowledge base 132 comprises:

- a rules base 133 containing a number of fuzzy IF-THEN rules. These IF-THEN rules lead to what action or actions should be taken in terms of the currently observed information. A fuzzy rule associates a condition described using linguistic variables and fuzzy sets to an output or a conclusion. The IF part is mainly used to capture knowledge and the THEN part can be utilized to give the conclusion or output in linguistic variable form. IF-THEN rules are widely used by the inference engine to compute the degree to which the input data matches the condition of a rule.
- a database 134 which defines the membership functions of the fuzzy sets. Fuzzy sets are sets whose elements have degrees of membership. Fuzzy set theory permits the gradual assessment of the membership of elements in a set; this is described with the aid of a membership function valued in the real unit interval [0,1]. The membership function represents the degree of truth. The system has associated one fuzzy set to every input feature. See the membership functions of features “entropy” and “machine learning score” previously presented.

The IF-THEN rules and the membership functions of the fuzzy sets might be defined by experts in the field or by exploiting approximation techniques from neural networks. On the one hand, experts extract comprehensible rules from their vast knowledge of the field. These rules are fine-tuned using the available input-output data. On the other hand, neural network techniques are used to automatically derive rules from the data.

Every rule is attached with a degree of belief or weight in the real interval (0, 1] that denotes a lower bound on the belief on the rule in terms of necessity measures. So, that rules that are demonstrated to be key for the decision system have a higher weight, and rules not so useful in the decision system have a lower weight. The rules created by the system may have a higher degree of belief than the rules created by a human, or vice versa.

For example, the system may create rules of the following form:

- Rule 1: IF (entropy(.text) is “high” OR entropy(.text) is “very_high” OR entropy(.text) is “extreme”) AND (call(“CryptAcquireContext”) OR call(“CryptEncrypt”) OR call(“CryptReleaseContext”)) THEN file 100 is encrypted with a degree of belief of 1.0
- Rule 2: IF entropy(file) is “very_high” AND ML_score(ENTROPY) is “high” THEN file 100 is encrypted with a degree of belief of at least 0.9
- Rule 3: IF has_section(UPX0) OR has_section(UPX1) or has_section(“X”) THEN file 100 is compressed with a degree of belief of at least 0.9
- Rule 4: IF file 100 is encrypted AND ML_score(API) is “low” and ML_score(Opcodes) is “low” THEN file 100 is benign with a degree of belief of at least 0.7
- Rule 5: IF file 100 is encrypted AND ML_score(API) is “medium” and ML_score(Opcodes) is “medium” THEN file 100 is suspicious with a degree of belief of at least 0.8
- Rule 6: IF file 100 is encrypted AND (ML_score(API) is “high” OR ML_score(Opcodes) is “high” THEN file 100 is malicious with a degree of belief of at least 0.9
- Rule 7: IF file 100 is compressed AND ML_score(ENT) is “high” AND (ML_score(API) is “medium” OR ML_score(Opcodes) is “medium” THEN file 100 is malicious with a degree of belief of at least 0.8
- Rule 8: IF file 100 is compressed AND ML_score(ENT) is “medium” or “low” AND ML_score(API) is “low” AND ML_score(Opcodes) is “low” THEN file 100 is benign with a degree of belief of at least 0.8

Due to the complexity and number of rules, in this implementation of the system only few rules related to very few fuzzy sets (entropy and ML scores) were presented. Notice that some of the conditions of rules are crisp values. For instance call(“CryptAcquireContext”) is TRUE if the executable calls “CryptAcquireContext” and FALSE otherwise.

<u>C/ Inference Engine</u>

The decision-making unit (Inference Engine) 135 is the inference procedure upon the fuzzy rules and given facts to derive a reasonable output or conclusion 140. Even that any fuzzy inference system could be used, e.g. Mandani Fuzzy Models, Sugeno Fuzzy Models, Tsukamoto Fuzzy Models, etc., in the current embodiment, the inference engine is based on the PGL+ reasoning system, for reasoning under possibilistic uncertainty and disjunctive vague knowledge. PGL+ is a possibilistic logic programming framework with fuzzy constants based on the Horn-rule fragment of Gödel infinitely-valued logic with an efficient proof algorithm based on a complete calculus and oriented to goals (conclusions). Fuzzy constants are interpreted as disjunctive imprecise knowledge and the partial matching between them is computed by means of a fuzzy unification mechanism based on a necessity-like measure.

For instance, if the entropy of the ".text" section is 7.2, the score returned by a given machine learning model trained on the structural entropy of the executable is 0.65 and the executable calls the functions "CryptAcquireContext" and "CryptEncrypt", then rules 1 and 2 are fired.

Rules fired:

● Rule 1: IF (entropy(.text) is "high" OR entropy(.text) is "very_high" OR entropy(.text) is "extreme") AND (call("CryptAcquireContext") OR call("CryptEncrypt") OR call("CryptReleaseContext")) THEN file 100 is encrypted with a degree of belief of 1.0

● Rule 2: IF entropy(file) is "very_high" AND ML_score(ENTROPY) is "high" THEN file 100 is encrypted with a degree of belief of at least 0.9

To aggregate the output, it is used the minimum as defined by the PGL+ reasoning system.

● File 100 is encrypted with a degree of belief >= min(degree of belief of rule 1, degree of belief of rule 2) == file 100 is encrypted with a degree of belief >= min(1.0, 0.9) == file 100 is encrypted with a degree of belief >= 0.9

Considering that the ML_score(API)=0.21 and the ML_score(Opcodes)=0.15 then, if rule 1 or rule 2 are activated, consequently rule 4, 5 and 6 are fired but only rule 4 is satisfied

● Rule 4: IF file 100 is encrypted AND ML_score(API) is "low" and ML_score(Opcodes) is "low" THEN file 100 is benign with a degree of belief of at least 0.7

As a result, the output of the inference engine 135 is: file 100 is benign with a degree of belief >= min(0.7, min(degree of belief of rule 1, degree of belief of rule 2)) -> file 100 is benign with a degree of belief >= min(0.7, min(1.0, 0.9)) ->"file 100 is benign with a degree of belief >= 0.7"

D/ Defuzzification.

The output of the Inference Engine 135 is a conclusion involving fuzzy constants together with the degree on the belief on the conclusion. The belief degree to classify the file 100 as malware is used, and fuzzy constants are transformed into crisp values using membership functions analogous to the ones used by the fuzzifier 131. The invention may use, but not limited to, one of the following defuzzification 136 methods:

1. Centroid of Area (COA)
2. Bisector of Area (BOA)
3. Mean of Maximum (MOM)
4. Smallest of Maximum (SOM)
5. Largest of Maximum (LOM)

In the current embodiment, the output fuzzy set might be decomposed into at least three sub-ranges or subsets, which are represented as membership functions in Fig. 11:

● BENIGN: from 0 to 0.4

● SUSPICIOUS: from 0.2 to 0.8

● MALICIOUS: from 0.5 to 1.0

Given the degree of fulfilment and the degree of belief of the consequent part of each fired rule, the fuzzy output is converted to a crisp output using, but not limited to, any of the aforementioned defuzzification methods 136. For instance, following the example presented in C/, if the output of the fuzzy inference engine is “file 100 is benign with a degree of belief >= 0.7” and the defuzzification method 136 is the “Mean of Maximum(MOM)”, then the crisp value of the fuzzy set using MOM is y* = (a + b) / 2, where a is the minimum highest value of the membership function “benign”, aka 0, and b is the maximum highest value of the membership function, aka 0.2. In consequence y* = (0 +0.2) / 2 = 0.1.

Use Case

The steps for predicting the maliciousness of a previously unseen executable 100 in a concrete implementation of the system of the second aspect of the present invention are described below. Due to its complexity, in this implementation of the system only a reduced subset of features 111 are extracted. Therefore, the number of machine learning methods and fuzzy rules and fuzzy sets has been reduced accordingly to fit the needs of this concrete implementation.

Steps:

1. An unseen executable (XXXXXXXXXXXX.exe) 100 is passed as input to the system. The preprocessing module 110 extracts a subset of features 111 that

provides an abstract view of the program. In particular, the preprocessing module 110 extracts at least the following features 111:

a. File entropy: 7.2

b. Windows API function calls: {“CryptAcquireContext”: True, “CryptEncrypt”: True, “CreateFile“: True, “CopyFile”: False, …}

c. Sequence of assembly language instructions: [“inc eax”, “call Clrsc”, “jump L1”, “add ebx, eax”, ...]

d. Structural entropy

2. The aforementioned data is passed as input to some machine learning methods to calculate a maliciousness score 123 based on a particular feature or subset of features 111.

a. Machine learning model 1 outputs a maliciousness score 123 equal to 0.65 with respect to the structural entropy of the executable 100. (A machine learning model is defined as the output generated when a machine learning algorithm is trained with your training data).

b. Machine learning model 2 outputs a maliciousness score 123 equal to 0.15 with respect to the sequence of instructions of the executable 100.

c. Machine learning model 3 outputs a maliciousness score 123 equal to 0.21 with respect to the imported Windows API functions.

3. Next, the features 111 and M.L. scores 123 are passed as input to the fuzzy inference module 130 to calculate the final maliciousness score 140 of the executable 100. Considering a rule base consisting of the following rules:

a. Rule 1: IF (entropy(.text) is “high” OR entropy(.text) is “very_high” OR entropy(.text) is “extreme”) AND (call(“CryptAcquireContext”) OR call(“CryptEncrypt”) OR call(“CryptReleaseContext”)) THEN file 100 is encrypted with a degree of belief of 1.0

b. Rule 2: IF entropy(file) is “very_high” AND ML_score(ENTROPY) is “high” THEN file 100 is encrypted with a degree of belief of at least 0.9

c. Rule 3: IF has_section(UPX0) OR has_section(UPX1) or has_section(“X”) THEN file 100 is compressed with a degree of belief of at least 0.9

d. Rule 4: IF file 100 is encrypted AND ML_score(API) is “low” and ML_score(Opcodes) is “low” THEN file 100 is benign with a degree of belief of at least 0.7

e. Rule 5: IF file 100 is encrypted AND ML_score(API) is “medium” and ML_score(Opcodes) is “medium” THEN file 100 is suspicious with a degree of belief of at least 0.8

f. Rule 6: IF file 100 is encrypted AND (ML_score(API) is “high” OR ML_score(Opcodes) is “high” THEN file 100 is malicious with a degree of belief of at least 0.9

g. Rule 7: IF file 100 is compressed AND ML_score(ENT) is “high” AND (ML_score(API) is “medium” OR ML_score(Opcodes) is “medium” THEN file 100 is malicious with a degree of belief of at least 0.8

h. Rule 8: IF file 100 is compressed AND ML_score(ENT) is “medium” or “low” AND ML_score(API) is “low” AND ML_score(Opcodes) is “low” THEN file 100 is benign with a degree of belief of at least 0.8

If the entropy of the “.text” section is 7.2, the score 123 returned by a given machine learning module 121 trained on the structural entropy of executables is 0.65 and the executable invokes the functions “CryptAcquireContext” and “CryptEncrypt”, then rules 1 and 2 are fired.

Rules fired:

● Rule 1: IF (entropy(.text) is “high” OR entropy(.text) is “very_high” OR entropy(.text) is “extreme”) AND (call(“CryptAcquireContext”) OR call(“CryptEncrypt”) OR call(“CryptReleaseContext”)) THEN file 100 is encrypted with a degree of belief of 1.0

● Rule 2: IF entropy(file) is “very_high” AND ML_score(ENTROPY) is “high” THEN file 100 is encrypted with a degree of belief of at least 0.9

To aggregate the output 140, it is used the minimum as defined by the PGL+ reasoning system.

● file 100 is encrypted with a degree of belief >= min(degree of belief of rule 1, degree of belief of rule 2) == file 100 is encrypted with a degree of belief >= min(1.0, 0.9) == file 100 is encrypted with a degree of belief >= 0.9

Considering that the ML_score(API)=0.21 and the ML_score(Opcodes)=0.15 then, if rule 1 or rule 2 are activated, consequently rule 4, 5 and 6 are fired but only rule 4 is satisfied.

● Rule 4: IF file 100 is encrypted AND ML_score(API) is “low” and ML_score(Opcodes) is “low” THEN file 100 is benign with a degree of belief of at least 0.7

As a result, the output of the inference engine 133 is: file 100 is benign with a degree of belief >= min(0.7, min(degree of belief of rule 1, degree of belief of rule 2)) ->

file 100 is benign with a degree of belief >= min(0.7, min(1.0, 0.9)) ->“file 100 is benign with a degree of belief >= 0.7”.

Afterwards the output of the fuzzy inference engine 133 (“file 100 is benign with a degree of belief >= 0.7”) is defuzzified 136 using the “Mean of Maximum(MOM”) defuzzification method. In consequence, the resulting crisp value returned by the system is calculated using the formula: y* = (a + b) / 2, where a is the minimum highest value of the membership function “benign”, i.e. 0, and b is the maximum highest value of the membership function, i.e. 0.2. In consequence y* = (0 +0.2) / 2 = 0.1.

In an embodiment, the PGL+ involves a semantical unification model of disjunctive fuzzy constants and three other inference patterns together with a deductive mechanism based on a modus ponens style. The PGL+ system allows expressing both ill-defined properties and weights with which properties and patterns can be attached with. For instance, suppose that the problem observation corresponds to the following statement “it is almost sure that the entropy file is around_20”. This statement can be represented in the proposed system with the formula:

(entropy (around_20), 0.9),

where entropy(.) is a classical predicate expressing the entropy property of the problem domain; around_20 is a fuzzy constant; and the degree 0.9 expresses how much is believed the formula entropy(around_20) in terms of a necessity measure.

In case around_20 denotes a crisp interval of entropy values, the formula (entropy (around_20), 0.9) is interpreted as the sentence “exists x in around_20 such that entropy(x)” being certain with a necessity of at least 0.9. So, fuzzy constants can be seen as (flexible) restrictions on an existential quantifier. Moreover, suppose the fuzzy pattern “we are more or less sure that the file is encrypted when its entropy is high” is considered. This pattern can be represented in the proposed system with the formula:

(entropy (high) -> encrypted, 0.7),

where high is a fuzzy constant and the degree 0.7 expresses how much is believed the file is encrypted since entropy is high.

From knowledge {(entropy(around_20), 0.9), (entropy(high) -> encrypted, 0.7)}, the PGL+ system computes the degree of belief of the crisp property encrypted by conveniently combining the degrees of belief 0.9 and 0.7 together with the degree of partial matching between both fuzzy constants high and around_20.

In another embodiment, the inference procedure based on the PGL+ reasoning system is divided in three algorithms which are applied sequentially. First, a completion algorithm, which extends the set of rules and facts with all valid clauses by means of the following Generalized Resolution and Fusion inference rules:

Generalized resolution:

$$\frac{\begin{array}{c}(p \wedge s \rightarrow q(A), \alpha)\\ (q(B) \wedge t \rightarrow r, \beta)\end{array}}{(p \wedge s \wedge t \rightarrow r, \min(\alpha, \beta))}\ [\mathrm{GR}], \quad \text{if } A \leq B$$

Fusion:

$$\frac{\begin{array}{c}(p(A) \wedge s \rightarrow q(D), \alpha)\\ (p(B) \wedge t \rightarrow q(E), \beta)\end{array}}{(p(A \cup B) \wedge s \wedge t \rightarrow q(D \cup E), \min(\alpha, \beta))}\ [\mathrm{FU}]$$

Second, a translation algorithm, which translates the completed set of rules and facts into a semantically equivalent set of 1-weighted clauses by means of the following inference rules:

Intersection:

$$\frac{(p(A), \alpha) \quad (p(B), \beta)}{(p(A \cap B), \min(\alpha, \beta))}\ [\mathrm{IN}]$$

Resolving uncertainty:

$$\frac{(p(A), \alpha)}{(p(A'), 1)}\ [\mathrm{UN}], \quad \text{for } A' = \max(1-\alpha, A)$$

Semantical unification:

$$\frac{(p(A), \alpha)}{(p(B), \min(\alpha, N(B \mid A)))}\ [\mathrm{SU}], \ \text{where } N(B \mid A) = \inf_{u \in U_\omega} A(u) \Rightarrow B(u),$$

where => is the reciprocal of Gödel’s many-valued implication, defined as x => y = 1 if x≤ y and x=> y = 1-x, otherwise.

Modus ponens:

$$\frac{\begin{array}{c}(p_1 \wedge \ldots \wedge p_n \rightarrow q, \alpha)\\ (p_1, \beta_1), \ldots, (p_n, \beta_n)\end{array}}{(q, \min(\alpha, \beta_1, \ldots, \beta_n))}\ [\mathrm{MP}]$$

And, finally, a deduction algorithm, based on the Semantical unification rule, which computes the maximum degree of possibilistic entailment of a goal from the equivalent set of 1-weighted facts.

The completion algorithm first computes the set of valid clauses that can be derived by applying the Generalized resolution rule (i.e. by chaining clauses). Then, from this new set of valid clauses, the algorithm computes all valid clauses that can be derived by applying the Fusion rule (i.e. by fusing clauses). As the Fusion rule stretches the body of rules and the Generalized resolution rule modifies the body or the head of rules, the chaining and fusion steps have to be performed while new valid clauses are derived. As the chaining and fusion steps cannot produce infinite loops and each valid clause is either an original clause or can be derived at least from two clauses, in the worst-case each combination of clauses derives a different valid clause. Hence, as a finite set of facts and rules N is had, in the worst-case the number of valid clauses is

$$N + \sum_{i=2}^{N} \binom{N}{i} \in \Theta(\frac{N^{N/2}}{N/2}).$$

However, in general, only a reduce set of clauses can be combined to derive new valid clauses. Indeed, c1, c2 and c3 can derive a new valid clause if c1 and c2, c1 and c3, or c2 and c3 derive a valid clause different to c1, c2 and c3.

The algorithm for translating a set of facts and rules into a set of 1-weighted clauses is based on the following result:

$$\|q(C)\|_P = \|q(C)\|_{\{(q(D_q),1)\}},$$

where, $D_q(u) = \inf\{D(u) \mid P \models (q(D),1)\}$ and *P* denotes the set of facts and rules; i.e. the maximum degree of satisfiability of a goal *q*(*C*) can be computed from a single 1-weighted clause (*q*(D_q); 1) instead of considering the entire original knowledge base *P*. Then, as D_q can be determined just from $P_q^{+¿¿}$(i.e. the clauses of $P^{+¿¿}$ whose heads are *q* or *q* depends on their heads), and each rule in $P_q^{+¿¿}$ can be replaced by a fact applying the Semantical unification and Modus ponens rules: each clause

$$(p_1 \wedge \ldots \wedge p_n \rightarrow q, \alpha) \in P_q^+$$

can be replaced by $(q, \min(\alpha, \min_{i=1,\ldots,n} \|p_i\|_P)).$

At this point, D_q can be computed from this finite set of facts by applying the UN and IN rules. As non-recursive programs are only considered, the above mechanism can be recursively applied for determining $\|p\|_P$ for each predicate *P* such that *q* depends on *p* in *P*, and thus, the time complexity of the translation algorithm is linear in the total number of occurrences of predicates symbols in $(P)^+$.

Finally, if (*q*(D_q); 1) is the 1-weighted clause computed by the translation algorithm for a propositional variable *q*, we have that $\|q(C)\|_P = \|q(C)\|_P = \text{N}\,(C|Dq)$

and thus, after applying the completion and translation algorithms, the maximum degree of satisfiability of a goal *q*(*C*) corresponds with the maximum degree of deduction of *q*(*C*) from *P* and can be computed in a constant time complexity in the sense that it is equivalent to compute the partial matching between two fuzzy constants:

$$N(C \mid D_q) = \inf_{u \in U} D_q(u) \Rightarrow C(u),$$

where => is the reciprocal of Gödel's many-valued implication.

One important feature of inference procedure based on the PGL+ reasoning system is that when extending the knowledge with new facts only the set of 1-weighted clauses must be computed again, and thus, the set of hidden clauses, which from a computational point of view is the hard counterpart of dealing with fuzzy constants, must be computed again only if new rules are added to the model.

Various aspects of the proposed method may be embodied in programming. Program aspects of the technology may be thought of as “products” or “articles of manufacture” typically in the form of executable code and/or associated data that is carried on or embodied in a type of machine readable medium. Tangible non-transitory “storage” type media include any or all of the memory or other storage for the computers, processors, or the like, or associated modules thereof, such as various semiconductor memories, tape drives, disk drives and the like, which may provide storage at any time for the software programming.

All or portions of the software may at times be communicated through a network such as the Internet or various other telecommunication networks. Such communications, for example, may enable loading of the software from one computer or processor into another, for example, from a management server or host computer of a scheduling system into the hardware platform(s) of a computing environment or other system implementing a computing environment or similar functionalities in connection with image processing. Thus, another type of media that may bear the software elements includes optical, electrical and electromagnetic waves, such as used across physical interfaces between local devices, through wired and optical landline networks and over various air-links. The physical elements that carry such waves, such as wired or wireless links, optical links or the like, also may be considered as media bearing the software. As used herein, unless restricted to tangible “storage” media, terms such as computer or machine “readable medium” refer to any medium that participates in providing instructions to a processor for execution.

A machine-readable medium may take many forms, including but not limited to, a tangible storage medium, a carrier wave medium or physical transmission medium. Non-volatile storage media include, for example, optical or magnetic disks, such as any of the storage devices in any computer(s), or the like, which may be used to implement the system or any of its components shown in the drawings. Volatile storage media may include dynamic memory, such as a main memory of such a computer platform. Tangible transmission media may include coaxial cables; copper wire and fiber optics, including the wires that form a bus within a computer system. Carrier-wave transmission media may take the form of electric or electromagnetic signals, or acoustic or light waves such as those generated during radio frequency (RF) and infrared (IR) data communications. Common forms of computer-readable media may include, for example: a floppy disk, a flexible disk, hard disk, magnetic tape, any other magnetic medium, a CD-ROM, DVD or DVD-ROM, any other optical medium, punch cards paper tape, any other physical storage medium with patterns of holes, a RAM, a PROM and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave transporting data or instructions, cables or links transporting such a carrier wave, or any other medium from which a computer may read programming code and/or data. Many of these forms of computer readable media may be involved in carrying one or more sequences of one or more instructions to a physical processor for execution.

Those skilled in the art will recognize that the present teachings are amenable to a variety of modifications and/or enhancements. For example, although the implementation of various components described herein may be embodied in a hardware device, it may also be implemented as a software only solution—e.g., an installation on an existing server.

The present disclosure and/or some other examples have been described in the above. According to descriptions above, various alterations may be achieved. The topic of the present disclosure may be achieved in various forms and embodiments, and the present disclosure may be further used in a variety of application programs. All applications, modifications and alterations required to be protected in the claims may be within the protection scope of the present disclosure.

The scope of the present invention is defined in the following set of claims.

Claims

1. A computer-implemented method for identifying a malicious file, the method comprising:

- performing a static analysis of a potentially malicious file, obtaining a set of features that provide an abstract view of the malicious file;

- performing a static machine learning classification process using as inputs said set of features, obtaining a preliminary classification output; and

- performing a fuzzy inference procedure based on possibilistic logic, for reasoning under possibilistic uncertainty and disjunctive vague knowledge, using as input variables said set of features and said preliminary classification output, generating an enhanced classification output that identifies the potentially malicious file as a malicious file or as a benign file.

2. A computer-implemented method according to claim 1, comprising:

- performing several static analyses of different types of the potentially malicious file, obtaining corresponding sets of features that provide abstract views of the malicious file;

- performing the static machine learning classification process using as inputs said sets of features, obtaining the preliminary classification output; and

- performing the fuzzy inference procedure based on possibilistic logic using as input variables the sets of features and the preliminary classification output.

3. A computer-implemented method according to claim 1, comprising:

- performing several static analyses of different types of the potentially malicious file, obtaining corresponding sets of features that provide abstract views of the malicious file;

- performing several static machine learning classification processes, each using as inputs at least one respective of the sets of features, obtaining corresponding several preliminary classification outputs; and

- performing the fuzzy inference procedure based on possibilistic logic using as input variables the sets of features and the preliminary classification outputs.

4. A computer-implemented method according to claim 3, comprising performing a further static machine learning classification process, using as inputs several or all of the sets of features, obtaining a corresponding further preliminary classification output; and

- performing the fuzzy inference procedure based on possibilistic logic using as input variable also the further preliminary classification output.

5. A computer-implemented method according to any one of the previous claims, wherein the fuzzy inference procedure comprises a fuzzification process that converts the input variables into fuzzy variables.

6. A computer-implemented method according to claim 5, wherein the fuzzification process comprises deriving membership functions relating the input variables with output variables through membership degrees of values of the input variables in predefined fuzzy sets, and representing the membership functions with linguistic variables, the linguistic variables being the fuzzy variables.

7. A computer-implemented method according to claim 5 or 6, wherein the fuzzy inference procedure further comprises an inference decision-making process comprising firing fuzzy possibilistic rules with values of the linguistic variables for the input variables, generating a fuzzy output that identifies a degree of belief that the potentially malicious file has to be a malicious file or a benign file.

8. A computer-implemented method according to claim 7, further comprising selecting which fuzzy possibilistic rules to fire in the inference decision-making process, based on at least the values of the linguistic variables for the input variables.

9. A computer-implemented method according to claim 7 or 8, wherein the fuzzy inference procedure further comprises a defuzzification process that converts said fuzzy possibilistic output into a crisp output, wherein said crisp output constitutes the enhanced classification output.

10. A computer-implemented method according to any of the previous claims, wherein the set or sets of features comprise at least one of the following:

- the frequency of use of Application Programming Interfaces (API) and their function calls;
- the representation of an executable file as a stream of entropy values, where each value describes the amount of entropy over a small chunk of code in a specific location of the potentially malicious file;
- the sequence of assembly language instructions executed by a software program constituting the potentially malicious file, in particular, the operational codes of the machine language instructions;
- the representation of an executable file, constituting the potentially malicious file, as an image, where every byte is interpreted as one pixel in the image, wherein the resulting array is organized as a 2-D array and visualized as a gray scale image;

- applicable program characteristics, at least including alphanumeric strings occurring in the body of the software program constituting the potentially malicious file and the fields from the header of the potentially malicious file.

11. A computer-implemented method according to claim 10, wherein the sets of features comprise at least two of the features sets.

12. A computer-implemented method according to any of the previous claims wherein the fuzzy inference procedure based on possibilistic logic is based on a PGL+ algorithm.

13. A computer-implemented method according to claim 12, wherein the PGL+ algorithm comprises applying three algorithms sequentially: a first algorithm that extends the fuzzy possibilistic rules by means of implementing a first set of rules; a second algorithm that translates the fuzzy possibilistic rules into a semantically equivalent set of 1-weighted clauses by means of implemented a second set of rules; and a third algorithm that computes a maximum degree of possibilistic entailment of a goal from the equivalent set of 1-weighted clauses.

14. A computing system for identifying a malicious file, comprising:

- a preprocessing computing module (110), configured and arranged to perform a static analysis of a potentially malicious file (100) to obtain a set of features that provide an abstract view of the malicious file;

- a machine learning module (120), configured and arranged to perform a static machine learning classification process using as inputs said set of features, to obtain a preliminary classification output; and

- a fuzzy inference module (130), configured and arranged to perform a fuzzy inference procedure based on possibilistic logic using as input variables said set of features and said preliminary classification output, to generate an enhanced possibilistic classification output (140) that identifies the potentially malicious file (100) as a malicious file or as a benign file.

15. A system according to claim 14, wherein:

- the preprocessing computing module (110) is further configured and arranged to perform several static analyses of different types of the potentially malicious file (100) to obtain corresponding sets of features that provide abstract views of the malicious file;

- the machine learning module (120) is further configured and arranged to perform the static machine learning classification process using as inputs the sets of features, to obtain the preliminary classification output; and

- the fuzzy inference module (130) is further configured and arranged to perform the fuzzy inference procedure based on possibilistic logic using as input variables the sets of features and the preliminary classification output.

16. A system according to claim 14, wherein:

- the preprocessing computing module (110) is further configured and arranged to perform several static analyses of different types of the potentially malicious file (100) to obtain corresponding sets of features that provide abstract views of the malicious file;

- the machine learning module (120) is further configured and arranged to perform several static machine learning classification processes, each using as inputs at least one respective of the sets of features, to obtain corresponding several preliminary classification outputs; and

- the fuzzy inference module (130) is further configured and arranged to perform the fuzzy inference procedure based on possibilistic logic using as input variables the sets of features and the preliminary classification outputs.

17. A non-transitory computer program product comprising computer executable software stored on a computer readable medium, the software being adapted to run at a computer or other processing means characterized in that when said computer executable software is loaded and read by said computer or other processing means, said computer or other processing means is able to perform the steps of the method according to any of claims 1-13.

Abstract

A computer-implemented method, a system and computer programs for identifying a malicious file are disclosed. The method comprises performing a static analysis of a potentially malicious file to obtain a set of features that provide an abstract view of the file; performing a static machine learning classification process using as inputs said set of features, to obtain a preliminary classification output; and performing a fuzzy inference procedure based on possibilistic logic using as input variables said set of features and said preliminary classification output, to generate an enhanced classification output that identifies the potentially malicious file as a malicious file or a benign file.

1/6

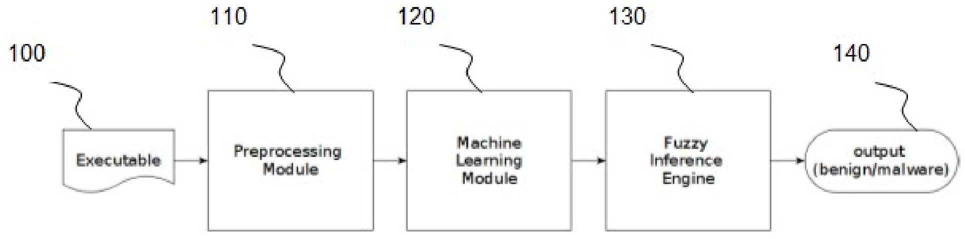

100
110
120
130
140
Executable
Preprocessing Module
Machine Learning Module
Fuzzy Inference Engine
output (benign/malware)

Fig. 1

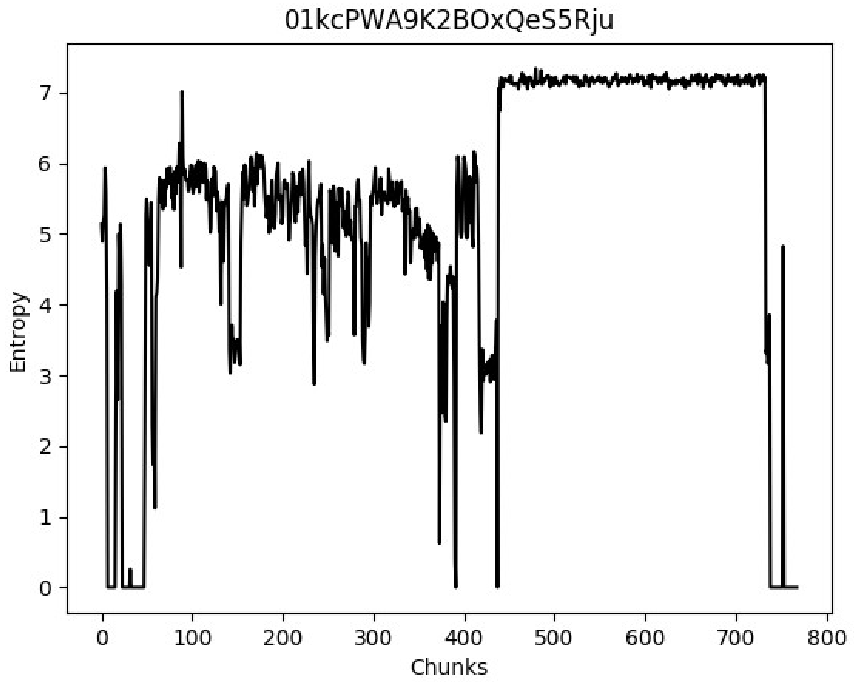

01kcPWA9K2BOxQeS5Rju
Entropy
Chunks
0
1
2
3
4
5
6
7
0
100
200
300
400
500
600
700
800

Fig. 2

2/6

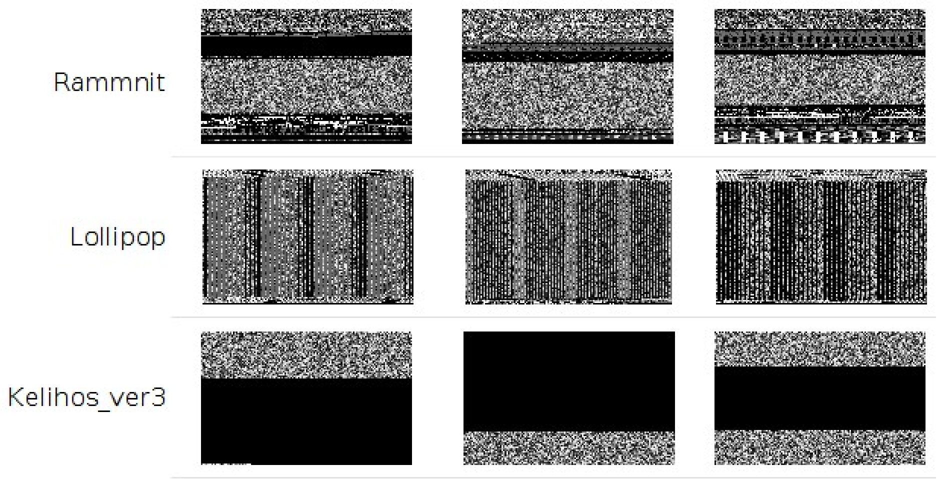

Rammnit
Lollipop
Kelihos_ver3

Fig. 3

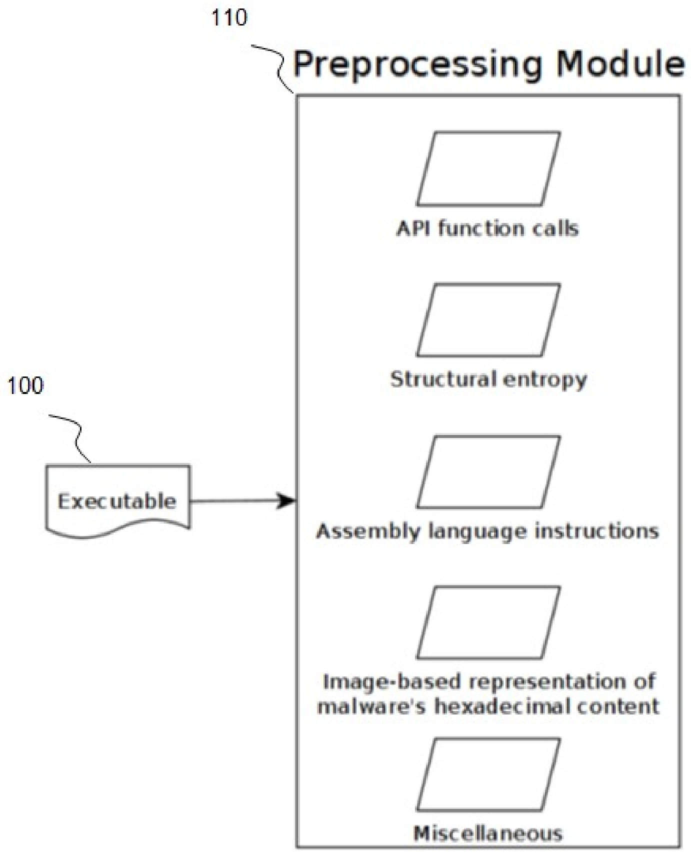

110
Preprocessing Module
API function calls
Structural entropy
Assembly language instructions
Image-based representation of malware's hexadecimal content
Miscellaneous
100
Executable

Fig. 4

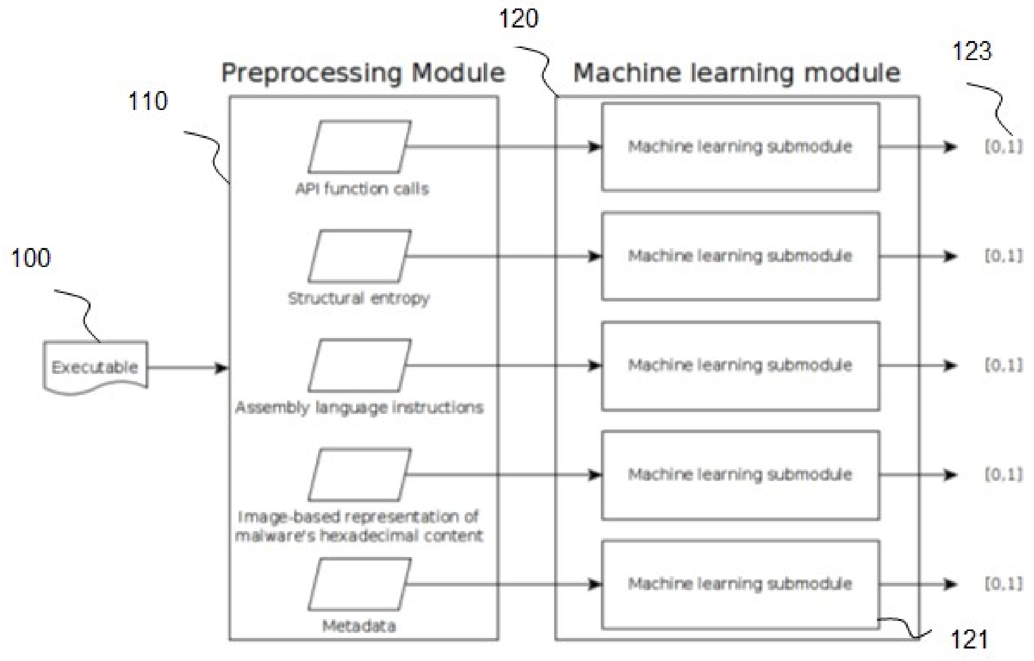

120
123
Preprocessing Module
Machine learning module
110
API function calls
Structural entropy
Assembly language instructions
Image-based representation of malware's hexadecimal content
Metadata
Machine learning submodule
Machine learning submodule
Machine learning submodule
Machine learning submodule
Machine learning submodule
[0,1]
[0,1]
[0,1]
[0,1]
[0,1]
100
Executable
121

Fig. 5

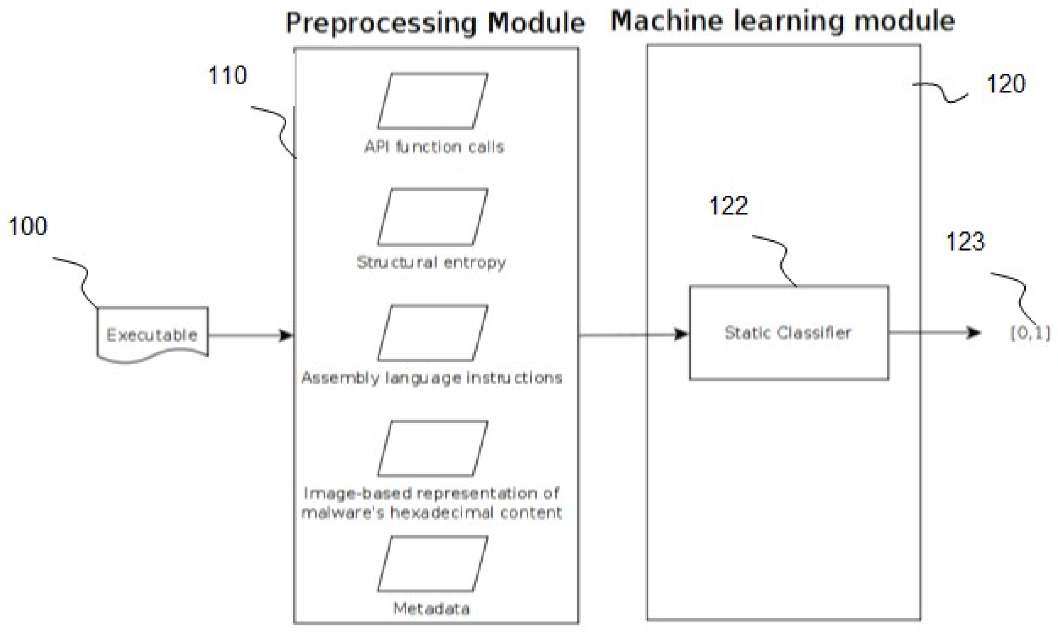

Preprocessing Module
Machine learning module
110
120
API function calls
Structural entropy
Assembly language instructions
Image-based representation of malware's hexadecimal content
Metadata
100
Executable
122
Static Classifier
123
[0,1]

Fig. 6

4/6

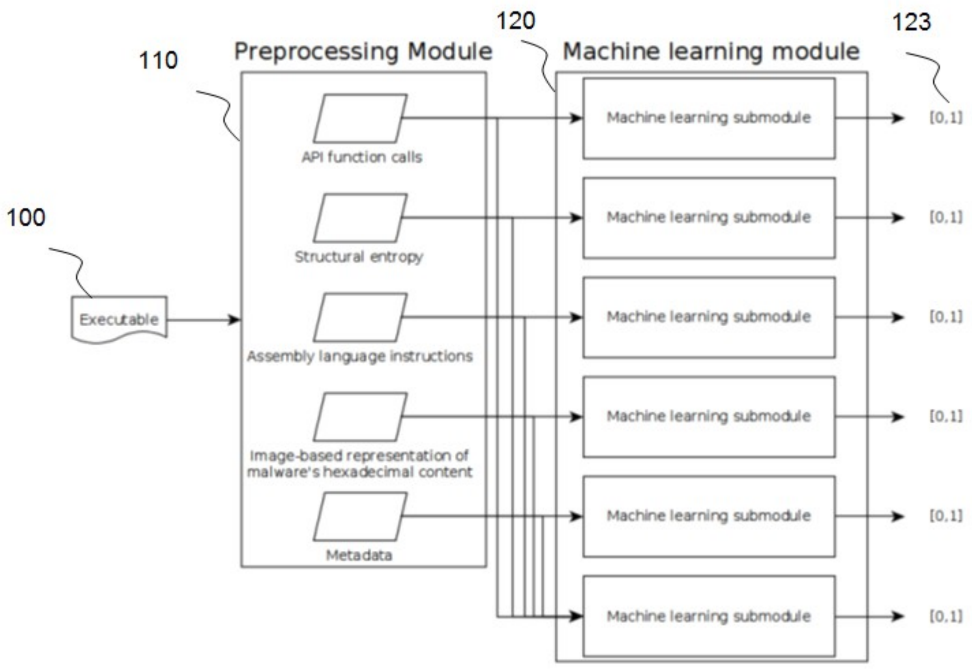

120
123
110
Preprocessing Module
Machine learning module
100
Executable
API function calls
Structural entropy
Assembly language instructions
Image-based representation of malware's hexadecimal content
Metadata
Machine learning submodule
Machine learning submodule
Machine learning submodule
Machine learning submodule
Machine learning submodule
Machine learning submodule
[0,1]
[0,1]
[0,1]
[0,1]
[0,1]
[0,1]

Fig. 7

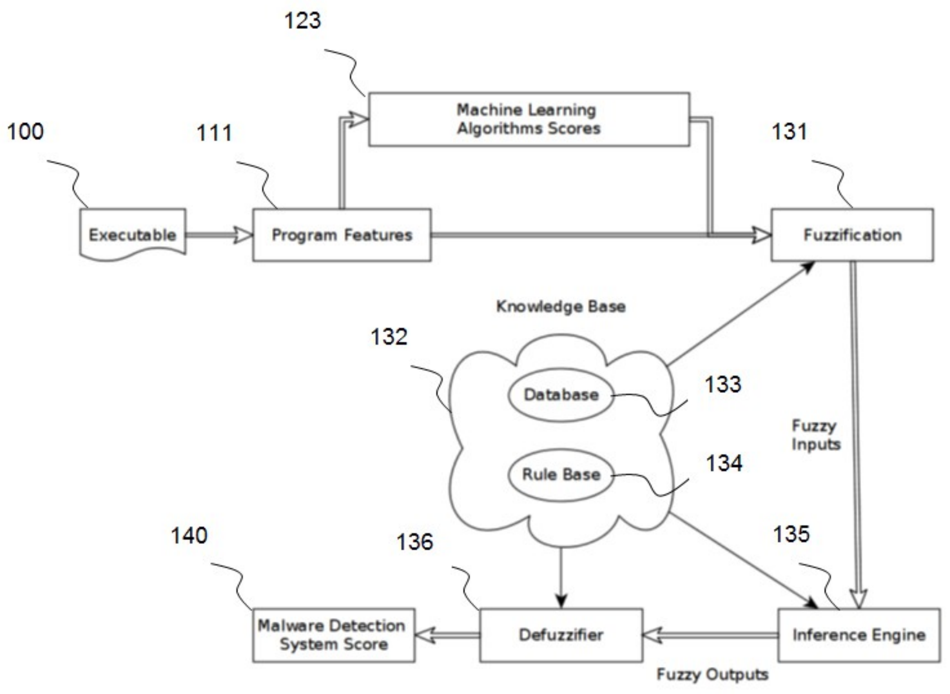

123
Machine Learning Algorithms Scores
100
111
131
Executable
Program Features
Fuzzification
Knowledge Base
132
Database
133
Rule Base
134
Fuzzy Inputs
140
136
135
Malware Detection System Score
Defuzzifier
Inference Engine
Fuzzy Outputs

Fig. 8

5/6

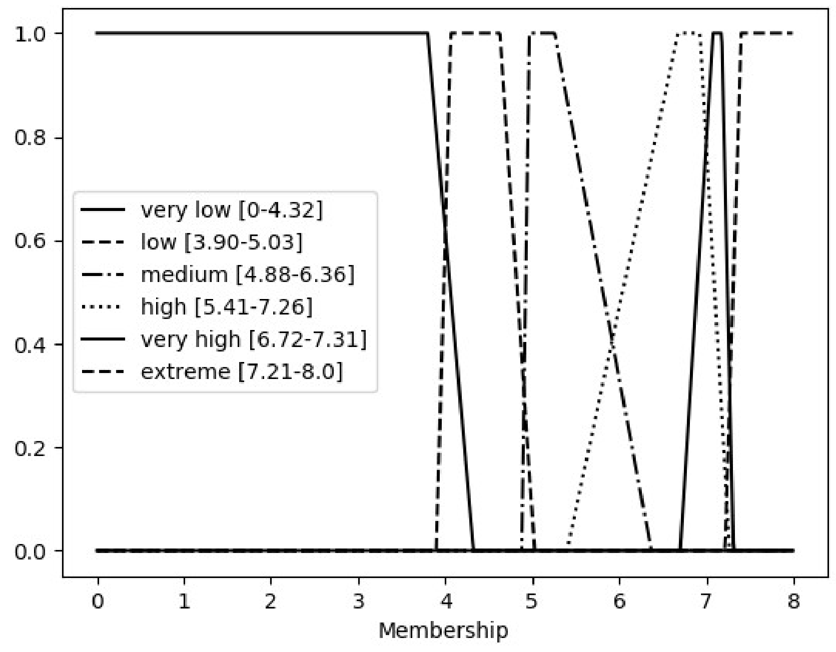


Fig. 9

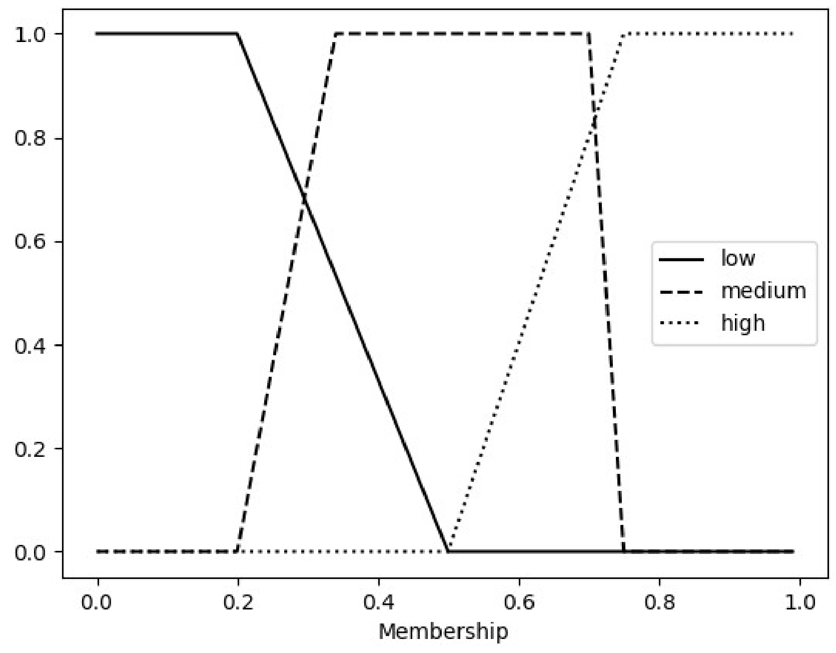


Fig. 10

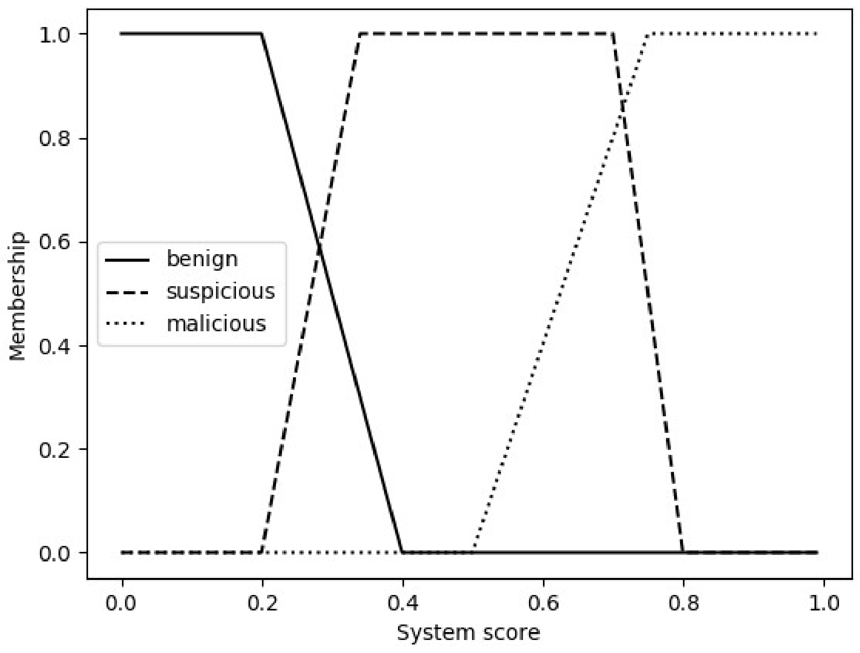


Fig. 11

Chapter 8

HYDRA: A Multimodal Deep Learning Framework for Malware Classification

This article has been published in the Journal Computers & Security (SJR:0.984), belonging to the First Quartile (Q1) as classified by Scimago Journal Rank. See Table 8.1.

Table 8.1: Journal metrics corresponding to the Journal Computers & Security for the year 2019.

Journal Metric	Value
Citescore	7.5
Impact Factor	3.579
SNIP	2.536
SJR	0.984

This research article *HYDRA: A Multimodal Deep Learning Framework for Malware Classification* [28] presents a framework for malware classification that combines both hand-crafted feature engineered and end-to-end components in a wide & deep learning architecture. The aim of this work is to combine various types of features to discover and learn the relationships between distinct modalities and maximize the benefits of multiple feature types to reflect the characteristics of malware executables. This is achieved through a modular architecture that can be broken down into three subnetworks, according to the different types of input in the system:

- The list of Windows API function calls.
- The sequence of assembly language instructions representing malware's assembly language source code.
- The sequence of hexadecimal values representing malware's binary content.

An extensive analysis of state-of-the-art methods on the Microsoft Malware Classification Challenge benchmark shows that the proposed solution achieves comparable results to gradient boosting methods in the literature and higher yield in comparison with deep learning approaches.

Computers & Security 95 (2020) 101873

Contents lists available at ScienceDirect

Computers & Security

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

HYDRA: A multimodal deep learning framework for malware classification

Daniel Gibert[*], Carles Mateu, Jordi Planes

University of Lleida, Jaume II, 69, Lleida, Spain

A R T I C L E   I N F O

Article history:
Received 26 December 2019
Revised 31 March 2020
Accepted 8 May 2020
Available online 12 May 2020

Keywords:
Malware classification
Machine learning
Deep learning
Feature fusion
Multimodal learning

A B S T R A C T

While traditional machine learning methods for malware detection largely depend on hand-designed features, which are based on experts' knowledge of the domain, end-to-end learning approaches take the raw executable as input, and try to learn a set of descriptive features from it. Although the latter might behave badly in problems where there are not many data available or where the dataset is imbalanced. In this paper we present HYDRA, a novel framework to address the task of malware detection and classification by combining various types of features to discover the relationships between distinct modalities. Our approach learns from various sources to maximize the benefits of multiple feature types to reflect the characteristics of malware executables. We propose a baseline system that consists of both hand-engineered and end-to-end components to combine the benefits of feature engineering and deep learning so that malware characteristics are effectively represented. An extensive analysis of state-of-the-art methods on the Microsoft Malware Classification Challenge benchmark shows that the proposed solution achieves comparable results to gradient boosting methods in the literature and higher yield in comparison with deep learning approaches.

1. Introduction

In recent years the number, and damage, of cyberattacks has drastically increased, up to the point that cyberthreats are considered among the top most notable risks for the upcoming years. The role cyberwarfare plays in our daily lives should not be underestimated, we have recenlty seen its influence on major elections and on crippling businesses overnight. The most notorious cyberespionage campaign against a political party took place in 2015 and 2016, affecting the Democratic National Committee (DNC), in which hackers infiltrated the DNC computer network and ended up releasing private and confidential information in a collection including approximately 19,000 emails and 8000 attachments from the DNC. Additionally, according to Symantec (Chandrasekar et al., 2017), the number of new ransomware families discovered during 2016 tripled, and they logged a 36% growth in ransomware infections. In 2017 two major cyberattacks, Wannacry in May, followed by Petya in July, held computer systems from all over the globe to ransom. Both malicious programs exploited a vulnerability of Microsoft Windows OS, codenamed EternalBlue (Vulnerabilities and Exposures, 2016), to rapidly spread from one computer to other computers on the same network.

The global malware industry is estimated to be worth millions or even billions of dollars, and continues to grow every year. The underground services market is maturing at increased rates, providing malicious software, cyber-capabilities, and products to other criminals, gangs, and even nation states. It has evolved into a powerful ecosystem, built to exploit every opportunity and weakness in an increasingly connected world. For instance, this year malware developers aimed to mine cryptocurrencies by stealing users' computing power or directly taking the credentials of their cryptocurrency wallet (Cleary et al., 2018; Daniely et al., 2018).

To keep up with malware evolution and be able to reduce the impact of cyberattacks it is necessary to improve computer systems' cyber defenses. One essential defense element is endpoint protection. These defenses range from appropriately keeping up-to-date with patches, to using host-based firewalls against malware. Specifically, anti-malware solutions are the last layer of defense against a cyberattack by preventing, detecting, and removing malicious software. Malware classification approaches can be classified into two categories: (1) static analysis-based detection and (2) dynamic analysis-based detection. On the one hand, static analysis examines the code of a program without executing it. On the contrary, dynamic analysis monitors the behavior of the program in

[*] Corresponding author.
E-mail addresses: daniel.gibert@diei.udl.cat (D. Gibert), carlesm@diei.udl.cat (C. Mateu), jplanes@diei.udl.cat (J. Planes).

https://doi.org/10.1016/j.cose.2020.101873
0167-4048/© 2020 Elsevier Ltd. All rights reserved.

the system. Afterwards, based on the information extracted from both static and dynamic analysis, experts manually define a set of rules to detect current and incoming threats.

Decades ago the number of malware threats was relatively low and simple hand-crafted rules were often enough to detect threats. Lately, however, the massive growth of malware streams does not allow anti-malware solutions to rely solely on expensive hand-designed rules. Consequently, machine learning has become an appealing signature-less approach for detecting and classifying malware due to its ability to generalize in relation to never-before-seen malware. Traditional machine learning approaches rely mainly on feature engineering to extract a set of discriminative features that provide a feature vector representation of malware that a classifier uses to determine the maliciousness of an executable. However, these solutions depend almost entirely on the ability of the domain experts to extract characterizing features that accurately represent the malware. Nevertheless, following recent advances in the machine learning field, there has been a trend towards replacing traditional machine learning pipeline systems with an end-to-end learning algorithm. An end-to-end learning algorithm takes the raw executable as input and tries to directly recognize whether or not it is malicious, or the malware family to which it belongs. Despite the benefits of end-to-end learning, almost no preprocessing and no hand-engineered knowledge, these systems might behave badly in problems where there are not many data available or the dataset is imbalanced.

To mitigate the limitations of end-to-end learning, in this paper we present HYDRA, a novel framework for malware classification using information from many sources to reflect various characteristics of malware executables. To the best of our knowledge, this research is the first application of multimodal deep learning to malware classification. The multimodal learning pipeline combines both hand-engineered and end-to-end components to build a robust classifier. This is achieved by means of a modular architecture that can be broken down into one or more subnetworks, depending on the different types of input of the system. Each of the subnetworks can be either independently trained to solve the same task and then combined, or jointly trained. The features learned by each component are gradually fused into a shared representation layer constructed by merging units with connections coming into this layer from multiple modality-specific paths. To avoid overfitting, during training we randomly drop out the information provided by one or more modalities of information. This prevents the co-adaptation of the subnetworks to a specific feature type. The performance of our multimodal learning algorithm has been evaluated on the Microsoft Malware Classification benchmark. In addition, we provide a comparison with state-of-the-art methods in the literature, including gradient boosting and deep learning methods.

The rest of the paper is organized as follows. Section 2 details the research in the machine learning field to address the problem of malware detection and classification. Section 3 introduces the problem of malware classification and, specifically, the task of classifying malicious Windows executables. Section 4 provides a detailed description of the different types of features or modalities used in our baseline framework. Section 5 presents the architecture of the multimodal neural network and Section 6 describes the experimentation. Lastly, Section 7 summarizes our research and provides future remarks on the ongoing research trends.

2. Related work

Machine learning approaches for tackling the problem of malware detection and classification can be divided into two groups: (1) static approaches (Ahmadi et al., 2016; Yuxin and Siyi, 2017) and (2) dynamic approaches (Bidoki et al., 2017; Ghiasi et al., 2015; Salehi et al., 2017).

Static approaches extract features without involving the execution of malware. Dynamic approaches require the program's execution.

Machine learning approaches are appealing to detect and classify malicious software because of their ability to recognize unseen malware by detecting patterns drawn from previous data. The machine learning workflow involves gathering available data, cleaning/preparing data, building models, validating and deploying in production. The data preparation process in traditional machine learning approaches includes preprocessing the executable, feature extraction, selection and reduction. Afterwards, the remaining features are used to train a model to solve the problem at hand, either to detect malware or to group malware into families. Thus, traditional machine learning methods rely mainly on feature engineering to extract discriminant features from a computer program that provide an abstract view that a classifier uses to make decisions about the inputs. On the contrary, end-to-end learning approaches jointly perform feature extraction and classification, replacing the aforementioned feature engineering process by a fully trainable system. An up-to-date review of machine learning approaches applied to either the problem of malware detection and classification is provided in (Souri and Hosseini, 2018; Ucci et al., 2019).

A description of the most relevant static methods in the literature, divided by the type of input features, is provided below.

The first machine learning classifiers were based on n-gram analysis. An n-gram is a contiguous sequence of n items from a text. In our domain, the items can be byte values (Jain and Meena, 2011; Moskovitch et al., 2008) or assembly language instructions (Santos et al., 2013; Shabtai et al., 2012), depending on the source of information. However, dealing with long n-grams is computationally prohibitive, as the number of unique combinations jointly increases exponentially with N. Consequently, researchers proposed various methods to learn n-gram like signatures without having to enumerate all n-grams during training. Gibert et al. (2017) and McLaughlin et al. (2017) proposed a shallow convolutional neural network architecture to extract n-gram like signatures from a sequence of opcodes to classify Windows and Android malware, respectively. Raff et al. (2018) and Krčál et al. (2018) designed end-to-end systems to learn directly from raw byte inputs, by stacking one or more convolutional layers to learn features from the hexadecimal representation of executables.

Malware authors usually protect malicious software against reverse engineering and detection by using encryption or packing methods to hide the malicious code. Entropy analysis has long been employed to detect the presence of encrypted and packed segments of code, as those segments tend to have higher entropy than native code. For instance, Lyda and Hamrock (2007) minutely examined a corpus of files consisting of plain text files, native, compressed and encrypted executables and noted that the average entropy of the files was 4.347, 5.09, 6.80 and 7.17, respectively. However, malware developers employ more or less sophisticated techniques to bypass simple entropy filters. As a result, researchers started examining what is known as the structural entropy of an executable (Sorokin, 2011). That is, an executable is split into non-overlapping chunks of fixed length and, for each chunk, we measure its entropy. Thus, each file is represented as an entropy time series. Wojnowicz et al. (2016) developed a method to quantify the extent to which variations in a file's structural entropy make it suspicious. In addition, Gibert et al. (2018b) proposed a convolutional neural network-based system to group malware into families.

The file format of the executables is a source of interesting features. In particular, Portable Executable (PE) files have information on the associated dynamically linked libraries, the sections of the program and their respective sizes, among others. More specifically, the invocation of Application Programming Interface (API) functions and system calls offers information about services and resources provided by the OS, which can be used to model program behavior (Aafer et al., 2013; Sami et al., 2010).

Moreover, it is common to combine information about the API function calls with other types of features in order to build a more robust classifier (Ahmadi et al., 2016; Hassen et al., 2017; Zhang et al., 2016).

An interesting approach is to represent the function calls as a directed graph, known as Function Call Graph (CFG), where its vertices represent the functions a computer program comprehends and the edges symbolize the function calls. For example, Kinable and Kostakis (2011) proposed to clusterizing malware based on the structural similarities between function call graphs, using the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm. Hassen and Chan (2017) proposed a linear time function call graph vector representation for malware classification and showed how to successfully combine the graph features with other non-graph features.

An original way to represent an executable is to reorganize its byte code as a gray scale image (Nataraj et al., 2011), where every byte is interpreted as one pixel in the image, and values range from 0 to 255 (0:black, 255:white). From this representation, it is possible to extract features describing the textures in an image, such as GIST (Nataraj et al., 2011), Haralick (Ahmadi et al., 2016), Local Binary Patterns (Ahmadi et al., 2016) and PCA (Narayanan et al., 2016) features that a classifier can use to classify malware. Additionally, Gibert et al. (2018c) and Khan et al. (2018) evaluated the use of convolutional neural network architectures to detect the presence of specific features and patterns in the data that can be used to group malware into families.

However, using only one type of features is not enough to correctly detect or classify malware in a real-world environment as the obfuscation techniques employed by malware authors might conceal one or more features used by the machine learning model. Thus, efforts are being made to design algorithms that can handle multiple categories of features. Current methods can be divided into two groups, depending on where the features are combined. On the one hand, early or data-level fusion approaches involve the integration of multiple data sources into a single feature vector that is used as input to a machine learning algorithm. For instance, Ahmadi et al. (2016) presented a categorization system that fuses multiple feature types (entropy statistics, image representation, frequency of opcodes, registers, symbols and Windows Application Programming Interfaces) into a single feature vector used to train boosting trees. On the contrary, late or decision-level fusion approaches are those that aggregate the decisions from multiple classifiers, each trained in separate modalities. To illustrate the point, Hassen et al. (2017) proposed an ensemble of individual malware classifiers to precisely classify malware, with a convolutional neural network to process the binary content represented as an image and a feedforward neural network feed with opcode n-gram features as input. As far as we know, there are no approaches in the literature that have successfully tried a deep or intermediate fusion strategy, where all modalities are fused into a single shared representation at some depth or gradually fused, for the task at hand.

3. The task of malware classification

This paper addresses the task of malware classification, which refers to the task of grouping or categorizing malware into families based on its characteristics and behavior. Distinguishing and classifying different types of malware is an important task as it provides information to better understand how the malware has infected the computers or devices, their threat level and how to protect against them. Notice that the only difference between the malware detection and classification tasks is the output of the system implemented. For instance, a malware detection system would receive as input an executable x and would output a single value $y = f(x)$ in the range from 0 to 1, indicating the maliciousness of the executable. A value closer to 0 indicates that the executable is benign and a value closer to 1 indicates that the executable is malicious. On the contrary, a classification system outputs the probability of a given executable belonging to each output category or family. Furthermore, the features extracted from a computer program are useful both for detecting if it is malicious and for classifying it.

Table 1
Class distribution in the microsoft malware classification challenge dataset.

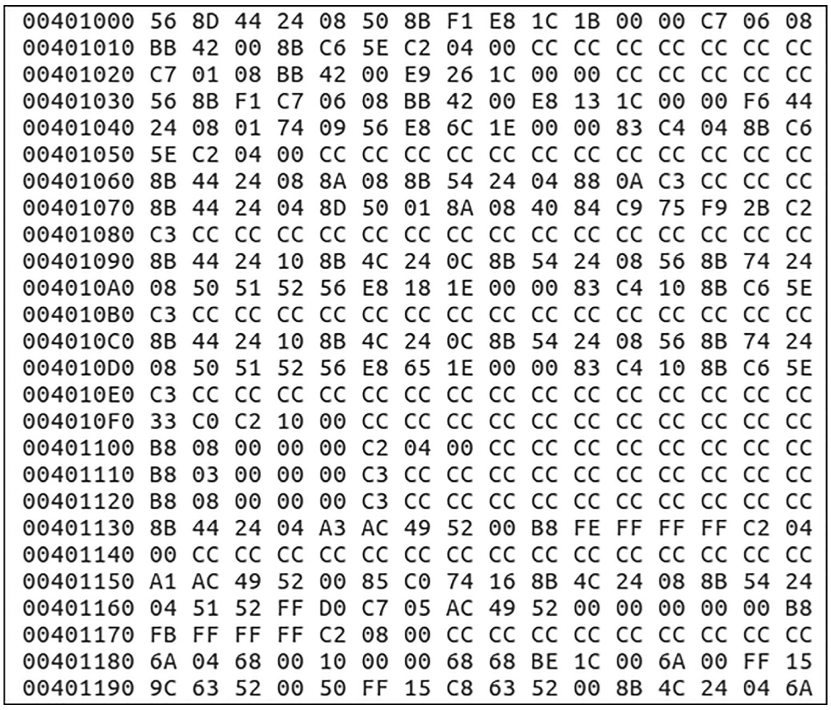
Family name	#Samples	Type
Ramnit	1541	Worm
Lollipop	2478	Adware
Kelihos_ver3	2942	Backdoor
Vundo	475	Trojan
Simda	42	Backdoor
Tracur	751	TrojanDownloader
Kelihos_ver1	398	Backdoor
Obfuscator.ACY	1228	Any kind of obfuscated malware
Gatak	1013	Backdoor

The task of malware detection and classification has not received the same attention in the research community as other applications, where rich labeled datasets exist, including image classification, speech recognition, etc. Due to legal restrictions, benign binaries are not shared, as they are often protected by copyright laws and thus, researchers cannot share the binaries used in their research. On the other hand, malicious binaries are shared through web sites such as VirusShare and VXHeaven. Nevertheless, unlike other domains where data may be labeled very quickly and in many cases by a non-expert, determining whether a file is malicious or its corresponding family or class can be a time-consuming process, even for security experts. Furthermore, services like VirusTotal specifically restrict sharing the vendor anti-malware labels to the public. Thus, for reproducibility purposes, we evaluated the multimodal deep learning system on the data provided by Microsoft for the Big Data Innovators Gathering Challenge (Ronen et al., 2018) of 2015, a high-quality public labeled benchmark. A complete description of the dataset is provided in the next section.

3.1. The microsoft malware classification challenge

Microsoft provided almost half a terabyte of malicious software for the Big Data Innovators Gathering Challenge (Ronen et al., 2018) of 2015. Nowadays, the dataset is hosted on Kaggle[1] and is publicly accessible. The dataset has become the standard benchmark to evaluate machine learning techniques for the task of malware classification. The set of samples represents 9 different malware families, where each sample is identified by a hash and its class, an integer representing one of the 9 malware families to which the malware belongs: (1) Ramnit, (2) Lollipop, (3) Kelihos_ver3, (4) Vundo, (5) Simda, (6) Tracur, (7) Kelihos_ver1, (8) Obfuscator.ACY and (9) Gatak. Fig. 1 displays the distribution of classes of the training data. We can observe that the number of instances of some families significantly outnumbers the number of instances of other families. There are two kinds of repre-

[1] https://www.kaggle.com/c/malware-classification/.



```
00401000 56 8D 44 24 08 50 8B F1 E8 1C 1B 00 00 C7 06 08
00401010 BB 42 00 8B C6 5E C2 04 00 CC CC CC CC CC CC CC
00401020 C7 01 08 BB 42 00 E9 26 1C 00 00 CC CC CC CC CC
00401030 56 8B F1 C7 06 08 BB 42 00 E8 13 1C 00 00 F6 44
00401040 24 08 01 74 09 56 E8 6C 1E 00 00 83 C4 04 8B C6
00401050 5E C2 04 00 CC CC CC CC CC CC CC CC CC CC CC CC
00401060 8B 44 24 08 8A 08 8B 54 24 04 88 0A C3 CC CC CC
00401070 8B 44 24 04 8D 50 01 8A 08 40 84 C9 75 F9 2B C2
00401080 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401090 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010A0 08 50 51 52 56 E8 18 1E 00 00 83 C4 10 8B C6 5E
004010B0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010C0 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010D0 08 50 51 52 56 E8 65 1E 00 00 83 C4 10 8B C6 5E
004010E0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010F0 33 C0 C2 10 00 CC CC CC CC CC CC CC CC CC CC CC
00401100 B8 08 00 00 00 C2 04 00 CC CC CC CC CC CC CC CC
00401110 B8 03 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401120 B8 08 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401130 8B 44 24 04 A3 AC 49 52 00 B8 FE FF FF FF C2 04
00401140 00 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401150 A1 AC 49 52 00 85 C0 74 16 8B 4C 24 08 8B 54 24
00401160 04 51 52 FF D0 C7 05 AC 49 52 00 00 00 00 00 B8
00401170 FB FF FF FF C2 08 00 CC CC CC CC CC CC CC CC CC
00401180 6A 04 68 00 10 00 00 68 68 BE 1C 00 6A 00 FF 15
00401190 9C 63 52 00 50 FF 15 C8 63 52 00 8B 4C 24 04 6A
```

Fig. 1. Hex view.

sentations when speaking about binary executables. On the one hand, an executable can be represented as a sequence of hexadecimal values for corresponding bytes of a binary file. For instance, approaches (Gibert et al., 2018b; Nataraj et al., 2011; Wojnowicz et al., 2016) are based on features extracted from this hexadecimal representation. On the other hand, the content of a binary file can be reverted/translated to assembly language. This process is known as disassembly. Common disassemblers are IDA Pro or Radare2. Approaches (Gibert et al., 2017; Kinable and Kostakis, 2011; Sami et al., 2010) illustrate this point.

3.1.1. Hexadecimal representation

The hex view represents the machine code as a sequence of hexadecimal digits. See Fig. 1. Each line is composed of the starting address of the machine codes in the memory and an accumulation of consecutive 16 byte values.

From this kind of representation one can extract byte n-grams, calculate the structural entropy of an executable, represent its binary content as a gray scale image, etc.

3.1.2. Assembly language source code

The assembly language source code contains the symbolic machine code of the executable as well as metadata information such as rudimentary function calls, memory allocation and variable information. A snapshot of a piece of one assembly file is shown in Fig. 2.

Assembly language consists of three types of statements:

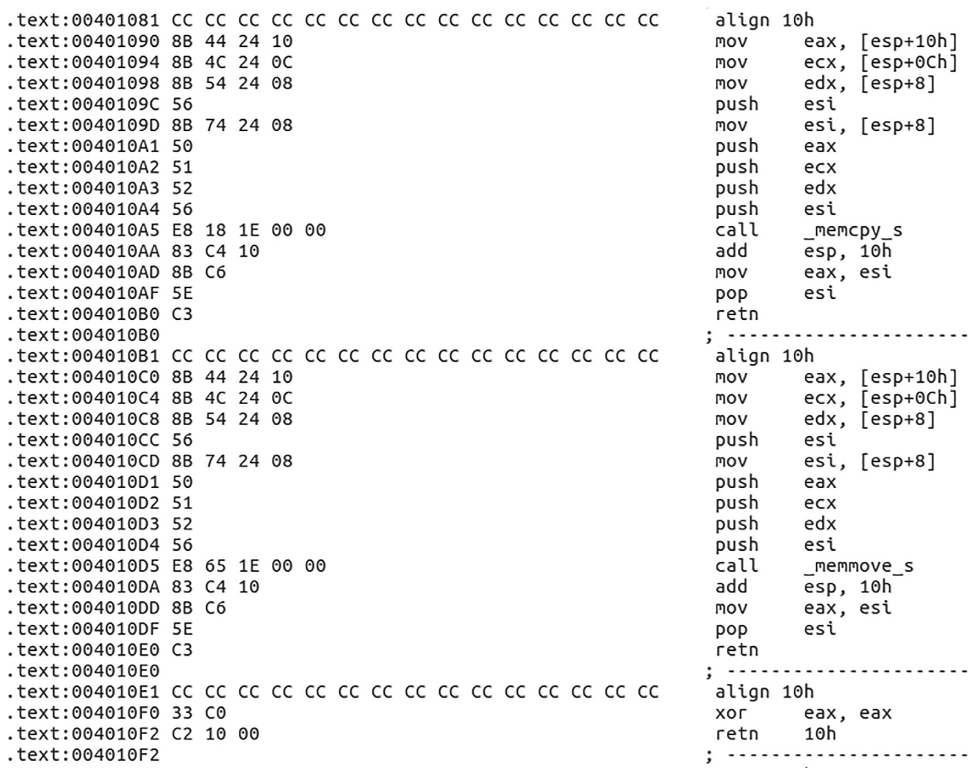
1. Instructions or assembly language statements. An instruction defines the operation to execute. Instructions are entered one instruction per line. Their format is as follows:
   `[label] mnemonic [operands]`
   An instruction is composed of two parts: (1) the name of the instruction to be executed and (2) the operands or parameters of the command.
   ```
   INC COUNT
   MOV TOTAL, 48
   ```
2. Assembler directives or pseudo-ops. Assembler directives are the commands part of the assembly syntax but not related to the processor instruction set.
3. Macros. A macro is a sequence of instructions assigned by a name that could be used anywhere in the program.
   ```
   %macro macro_name num_params
   <macro body>
   % endmacro
   ```

4. Modalities description

This paper proposes a multimodal deep learning system to categorize malware into families that involves multiple modalities of data:

1. The list of Windows API functions calls.
2. The sequence of assembly language instructions representing malware's assembly language source code.
3. The sequence of hexadecimal values representing malware's binary content.

These feature types have been chosen because of their respective advantages and limitations. A detailed description and an in-depth anlysis of the aforementioned modalities are provided below, together with the definition of the individual components of the multimodal architecture.



```
.text:00401081 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC      align 10h
.text:00401090 8B 44 24 10                                      mov     eax, [esp+10h]
.text:00401094 8B 4C 24 0C                                      mov     ecx, [esp+0Ch]
.text:00401098 8B 54 24 08                                      mov     edx, [esp+8]
.text:0040109C 56                                               push    esi
.text:0040109D 8B 74 24 08                                      mov     esi, [esp+8]
.text:004010A1 50                                               push    eax
.text:004010A2 51                                               push    ecx
.text:004010A3 52                                               push    edx
.text:004010A4 56                                               push    esi
.text:004010A5 E8 18 1E 00 00                                   call    _memcpy_s
.text:004010AA 83 C4 10                                         add     esp, 10h
.text:004010AD 8B C6                                            mov     eax, esi
.text:004010AF 5E                                               pop     esi
.text:004010B0 C3                                               retn
.text:004010B0                                                 ; ---------------------
.text:004010B1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC      align 10h
.text:004010C0 8B 44 24 10                                      mov     eax, [esp+10h]
.text:004010C4 8B 4C 24 0C                                      mov     ecx, [esp+0Ch]
.text:004010C8 8B 54 24 08                                      mov     edx, [esp+8]
.text:004010CC 56                                               push    esi
.text:004010CD 8B 74 24 08                                      mov     esi, [esp+8]
.text:004010D1 50                                               push    eax
.text:004010D2 51                                               push    ecx
.text:004010D3 52                                               push    edx
.text:004010D4 56                                               push    esi
.text:004010D5 E8 65 1E 00 00                                   call    _memmove_s
.text:004010DA 83 C4 10                                         add     esp, 10h
.text:004010DD 8B C6                                            mov     eax, esi
.text:004010DF 5E                                               pop     esi
.text:004010E0 C3                                               retn
.text:004010E0                                                 ; ---------------------
.text:004010E1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC      align 10h
.text:004010F0 33 C0                                            xor     eax, eax
.text:004010F2 C2 10 00                                         retn    10h
.text:004010F2                                                 ; ---------------------
```

Fig. 2. Assembly view.

4.1. Windows API function calls

The frequency of use of Application Programming Interfaces (API) and their function calls are regarded as very characterizing features. Literature has demonstrated that API calls can be analyzed to model the program behavior. API functions and system calls are related with services provided by operating systems. They support various key operations such as networks, security, system services, file management, and so on. In addition, they include various functions to utilize system resources, such as memory, file system, network or graphics. There is no other way for software to access system resources that are managed by operating systems without using API functions or system calls and thus, API function calls can provide key information to represent the behavior of the software. In this work, each API function and system call is treated as a feature. The feature range is [0,1]; 0 (or False) if the API function or system call hasn't been invoked by the program; 1 (or True) otherwise. Alternatively, one can count how many times each API function has been called during the execution of the program.

Because many malware programs are packed, leaving only the stub of the import table or perhaps even no import table at all, our approach will search for the name of the dynamic link library or function in the body of the suspected malware (by disassembling the executable).

The number of Windows OS API functions is extremely large. Considering all of them would bring little or no meaningful information for malware classification. Consequently, the analysis was restricted to a subset of API functions. The complete list of Windows API functions was reduced to only those functions that were invoked at least thrice in our training data. The remaining functions were not considered in the analysis. Among the most used functions we found the following: the *Sleep* function, which is used to evade dynamic analyzers, the *VirtualAlloc* function, which is used to allocate memory to store the unpacked code in the newly allocated block and perform a jump to run the code from there, and the *LoadLibraryA* and *GetProcAddress* functions, which are both used to resolve the addresses of the API calls made by the program.

The total number of functions invoked is 10670, almost equal to the number of training samples, which might lead to overfitting. High dimensionality results in increased cost and complexity for both feature extraction and classification. In practice, the algorithm might perform badly if the dimensionality is increased beyond a certain point when there is a finite number of training samples. This problem is known as the curse of dimensionality. Consequently, feature selection has been applied to select only a subset of the features. Feature selection is the process of selecting a subset of features that are more relevant to a predictive modeling problem.

This helps to remove unneeded, irrelevant and redundant attributes from the data that do not contribute to the accuracy of a predictive model. Afterwards, the subset of features is used to learn the predictive modeling problem. An overview of the proposed method is presented in Fig. 3. In our specific implementation, the classification algorithm is a feed-forward network whose hyper-parameters have been selected using a grid search. The experiments to select the optimal subset of features are described in detail in Section 6.1. A detailed description of our feed-forward network architecture is introduced below.

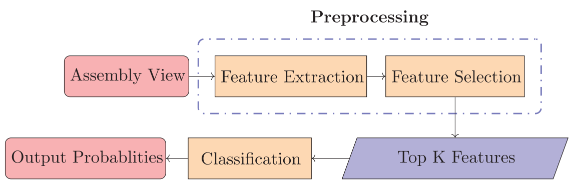


Fig. 3. Traditional pipeline of an API-based malware classification system.

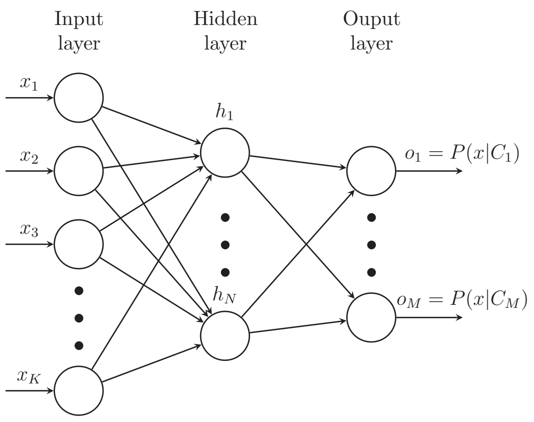


Fig. 4. API-based feed-forward network architecture. *N* and *M* are equal to 250 and 9, respectively.

4.1.1. API-Based feed-forward neural network

An overview of the architecture is described in Fig. 4. The input of the network is a vector containing the K most relevant API-based features according the feature selection technique of our choice. The output of the network is given by the function $f(x) = f^{(2)}(f^{(1)}(x))$, where $f^{(1)}$ refers to the first layer or hidden layer of the network and $f^{(2)}$ refers to the output layer. The mathematical formulation of $f^{(i)}$ is $f^{(i)} = \alpha(Wx + b)$, where α represents the activation function, x the input of the layer and W and b the weights and biases, respectively. In particular, the activation function of the hidden layer is the ELU function (Clevert et al., 2015) while the output layer has no activation function. Instead, it calculates the softmax function to generate the normalized output probabilities.

4.2. Mnemonics analysis

The most common approach to categorize a given sample of text in natural language processing is through n-gram analysis. N-gram analysis calculates the n-gram words distribution of a file as a means of solving a predictive modeling problem. In addition, n-grams have been one of the most popular features used for malware detection or classification (Santos et al., 2013; Shabtai et al., 2012). The simplest approach is to capture only the instruction used as the base. On encountering the instruction *mov eax, [esp+10h]* we simply reduce it to *mov*.

Specifically, mnemonic n-grams are extracted from the sequence of mnemonics included in the assembly language source code of malware. To give a specific example of the process, the mnemonics sequence in Fig. 2, from bytes 00,401,090 to 004010B0 would have the following 2-grams:

```
  [[mov,mov], [mov,mov], [mov,push],
[push,mov],
  [mov,push], [push,push], [push,push], [push,
push],
  [push,call], [call,add], [add,mov],
[mov,pop],
  [pop, retn]]
```

N-gram based methods construct a feature vector representation of malware where each element in the vector indicates the number of appearances of a particular n-gram in the instruction sequence.

The main drawback of n-gram based methods is that the number of unique n-grams depends on n, the number of mnemonics that each n-gram will contain. Since the number of unique n-grams is huge, it is difficult to run machine learning algorithms on the original data. One solution is to perform feature selection, i.e. a process of identifying the best features and filtering out less important features. Another solution was proposed by (Gibert et al., 2017), who presented an alternative to n-gram counts using convolutional neural networks to automatically learn the most discriminative features from a sequence of mnemonics without having to apply any feature selection technique to make the problem tractable.

4.2.1. Convolutional neural network as an alternative to N-grams

The main advantage of a convolutional neural network based approach is that it removes the need to manually enumerate the large number of n-grams during training, as n-gram based approaches do. Instead, it learns n-gram like signatures through the convolutional layers. The most notable implication of such an approach is the elimination of the traditional pipeline composed of feature extraction, feature selection and reduction and classification, as both procedures are optimized together during training.

Due to the advantages of a convolutional neural network based approach, the component responsible for addressing this modality of data has been constructed considering the architecture presented by Gibert et al. (2017) as baseline, with a few minor modifications. The network differs on: (i) the kernel sizes, (ii) the number of filters per size, and (iii) the size of the input layer. In addition, ours also has dropout applied to the input layer. The overall architecture is presented in Fig. 5. It comprises the following layers:

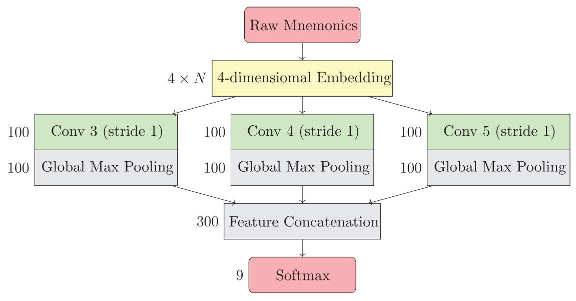


Fig. 5. Convolutional neural network for malware mlassification from sequences of mnemonics.

Input layer. The network takes as input an executable represented as a sequence of mnemonics. As the network cannot be fed with text just as strings, each mnemonic is converted to a one-hot vector. To form a one-hot vector, we associate each mnemonic with a numerical ID in the range 1 to I, where I is the vocabulary size. A one-hot vector is a vector of zeros of size I, with a '1' in the position of the mnemonics' ID.

Embedding layer. One-hot vectors cannot encode semantic information about similar operations or similar meaning. To address this issue, each mnemonic is represented as a low-dimensional vector of real values (word embedding) of size

M, where each value captures a dimension of the mnemonics' meaning.

Convolutional layer. This layer is responsible for convolving various filters over the mnemonic sequences and extracting the n-gram like features from it. The size of each filter is hxk where $h \in \{3, 4, 5\}$ and k is equal to the size of the mnemonic's embedding. Consequently, filters are applied to sequences containing from 3 to 5 mnemonics. The activation function adopted is the Exponential Linear Unit or ELU (Clevert et al., 2015). Having different filter' sizes allows the network to detect salient sub-sequences in the sequence of instructions that have variations in size.

Global max-pooling layer. The global max-pooling is applied to extract the maximum activation of each of the feature map activations passed from the convolutional layer.

Softmax layer. It linearly combines the features learned by the previous layers and applies the softmax function to output the normalized probability distribution over malware families.

Xavier's initialization (Glorot and Bengio, 2010) has been used to initialize the weights of the network with the exception of the embedding layer, in which the initial values of the embedding had been initialized with random values from a uniform distribution ranging from -1 to 1. Additionally, dropout (Hinton et al., 2012) was applied to the input, the convolutional and the output layers, with a percentage of 0.1, 0.1 and 0.5 dropped neurons. For a complete description of the experiments performed see Section 6.2.

4.3. *Byte analysis*

Similar to the mnemonics analysis counterpart, there had been attempts in the literature (Krčál et al., 2018; Raff et al., 2018) to build end-to-end malware detection systems from raw byte sequences. These approaches take as input the raw byte sequences from the hexadecimal representation of the malware's binary content and try to identify whether or not the executable is malicious. The main challenges that these approaches have to deal with are:

- The meaning of any byte is context-dependent and could encode any type of information such as binary code, human-readable text, images, etc. In addition, the same instruction can be encoded using different byte codes depending on its arguments such as the cmp instruction, whose binary code can begin with 0x3C, 0x3D, 0x3A, 0x3B, 0x80, 0x81, 0x38 or 0x39 depending on the arguments given.
- The content of a Portable Executable file exhibits various levels of spatial correlation. Nearby instructions in a function are spatially correlated. However, function calls and jump instructions produce discontinuities over code instructions and functions. Subsequently, these discontinuities are maintained through the binary content.
- By treating an executable as a sequence of bytes, we are dealing with sequences of millions of time steps, becoming one of the most challenging sequence classification problems with regard to the size of the time series.

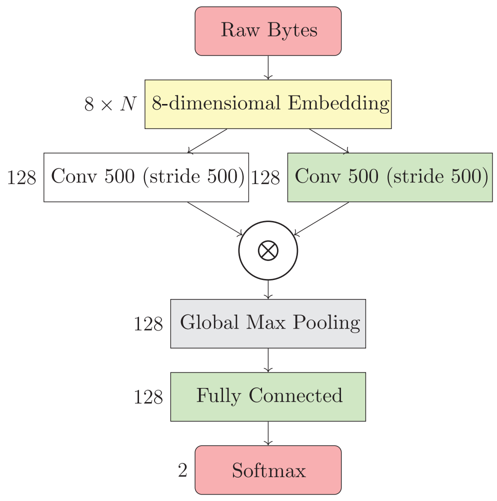


Fig. 6. MalConv architecture.

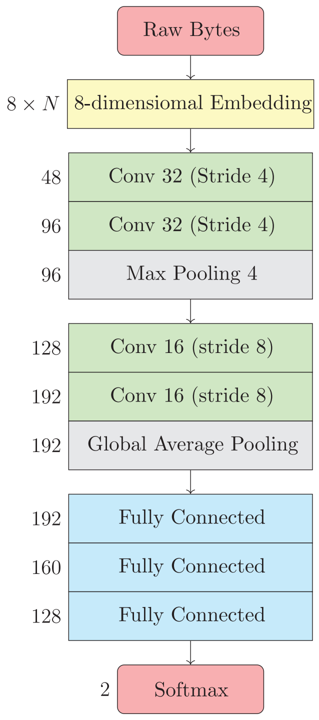


Fig. 7. DeepConv architecture.

4.3.1. *State-of-the-art methods*

Raff et al. (2018) presented a shallow convolutional neural network architecture consisting of an embedding layer, followed by a gated convolutional layer with filters of size 500 combined with a stride of 500, plus a global max-pooling layer and a softmax layer. This architecture will be called MalConv from now on. Cf. Fig. 6.

Krčál et al. (2018) proposed a deeper architecture that comprises an embedding layer followed by four convolutions with strides and max-pooling between the second and third convolutions. Afterwards, global average pooling is applied to generate the average features in the byte sequences. Finally, the features are non-linearly combined through various fully-connected layers and lastly a softmax layer. This architecture will be called DeepConv from now on. Cf. Fig. 7.

Performing convolutions on raw byte values implies interpreting that certain byte values are intrinsically closer to each other than others, which is known to be false, as the meaning of a particular byte is context-dependent. Consequently, both approaches represent bytes as a distributed vector representation of size K,

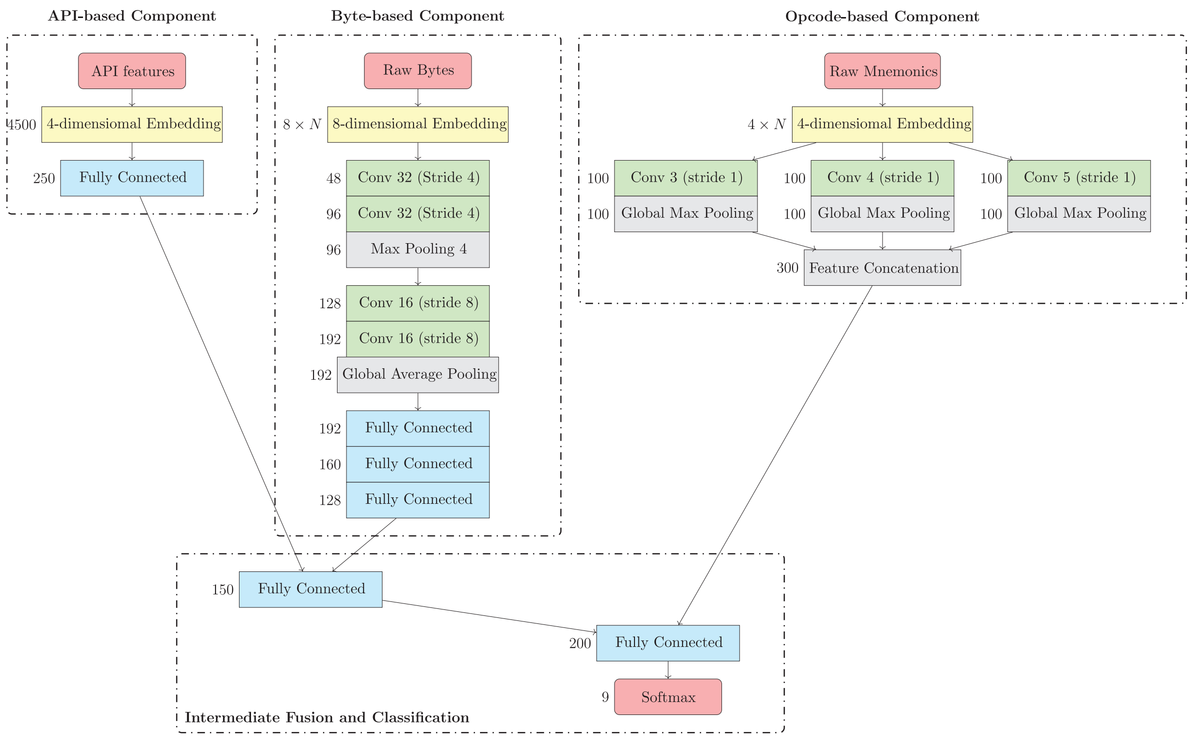


Fig. 8. Multimodal deep neural network architecture.

where each element contributes to the definition of each byte with the goal of capturing the context of a byte in the executable, its semantic similarity as well as its relation with other bytes.

The third component of the multimodal network is responsible for extracting features from the byte sequence representation, which has been constructed according to the network architecture presented by Krčál et al. (2018) as it achieves better performance than MalConv (Raff et al., 2018) in our experiments. See Section 6.3.

5. Multimodal deep learning framework

Fig. 8 shows the architecture of HYDRA, our multimodal deep learning framework for malware classification. This architecture aims to serve as a baseline for future implementations of malware detection systems. It only includes those feature types that the authors consider essential to any machine learning system for the task at hand. Nonetheless, new feature types can be added to the architecture as needed. It consists of 4 main components: (1) the API-based component, (2) the mnemonics-based component, (3) the byte-based component and (4) the feature fusion and classification component. The first three components are not connected to each other. Each one extracts features from a different abstract representation of malware (from a different data modality). The final component is responsible for fusing the features learned by each component into a shared representation and for producing the classification predictions. This architecture consists of both hand-engineered and end-to-end components. On the one hand, the hand-engineered component learns the complex relationships among the input API feature vector. On the contrary, the end-to-end components learn features from malware represented as a sequence of mnemonics and as a sequence of bytes. As end-to-end learning requires a great deal of labeled data to work properly, combining hand-engineered and end-to-end components helps to mitigate the limitations of the system and more specifically, of the byte-based component (See Section 6.3), and to build a stronger classifier by combining the strengths of both approaches.

5.1. Architecture

In this subsection, the architecture of the multimodal neural network is described, in particular the input layer, the three feature components and the fusing component.

Input layer. The dataset consists of a set of pairs x^i, y^i, where x^i is an executable and y^i is the category label or family to which it belongs. Each executable x^i is represented as an n-tuple, where n is equal to the number of different modalities of data. In our case $n = 3$, as each executable is represented as a list of API function calls x^i_A, a sequence of mnemonics x^i_M, and lastly a sequence of bytes x^i_B.

API-based component. Let $x^i_A \in \mathbb{Z}^K$ be the API feature vector of the ith executable in the training set. Each vector consists of K feature values, where each feature indicates whether or not a particular API function has been invoked. For instance, if $x^i_A(j)$ is equal to 1 it indicates that the ith executable has invocked API function j.
The API-based component takes as input the API feature vector and non-linearly combines the features into a low-dimensional feature vector A of size 250, where $A^i = h(x^i_A) = \sigma(W_A x^i_A + b_A)$ and W_A and b_A denote the weights and biases of the fully-connected layer.

Mnemonics-based component. Let x^i_M denote the sequence of mnemonics extracted from the assembly language source code of the *i*th executable. As already explained in Section 4.2.1, each mnemonic is mapped to a vector of real numbers of size 4. The mnemonics-based component outputs a feature vector *M* of size equal to 300 consisting of the concatenation of the 3-gram, 4-gram and 5-gram like signatures.

Byte-based component. Let x_B^i refers to the sequence of bytes extracted from the hexadecimal representation of the binary content of the *i*th executable. This component outputs a feature vector *B* containing a low-dimensional representation of the malware's binary content of size 128 (See Section 4.3.1).

Intermediate fusion and classification. The learned representations A, M and B of malware are merged iteratively across multiple fusion layers during training and combined into a shared multimodal representation. This process is known as intermediate fusion. First, vectors A and B are fused into vector C of size 150. Afterwards, vectors C and M are combined in a 1-D vector of size 200, called P. This joint multimodal representation P is later used to classify a malicious executable into the corresponding family, as follows:

$$p = softmax(W_c P + b_c)$$

where *p* is a vector of size C ($C = 9$), and W_c and b_c are the weights and biases of the layer. The softmax function outputs the probability of an executable belonging to any of the malware families in the training set. The size of vectors C and P were defined during the configuration of the network. Various values for the number of hidden units were tried and the ones yielding to better results were selected. In addition, fusing all features in the same layer yield to worse results even if these are statistically insignificant.

For consistency, all layers have been initialized using Xavier's weight initialization (Glorot and Bengio, 2010). The non-linear functions through all convolutional and fully connected layers are the ELU (Clevert et al., 2015) and the SELU (Klambauer et al., 2017) activation function.

Two aspects have been critical for the success of our multimodal setting: (1) per-modality pretraining and transfer learning, and (2) multimodal dropout.

5.2. Pretraining

In our experiments we observed that taking all modalities of information as input is suboptimal, since it leads to overfitting one subset of features belonging to one modality and underfitting the features belonging to the others. This issue has been addressed by separately pretraining each component and optimizing their hyperparameters for each subtask. Consequently, we randomly split the training data into two sets, 80% for training and 20% for the validation set and we trained three models to classify malware, one model taking one modality as input. Afterwards, the weights of each component in the multimodal neural network are initialized with the optimal pretrained weights that each network has learned for each task. The idea is to transfer the knowledge learned by each model into the multimodal neural network to save training time and help the network converge faster.

5.3. Regularization mechanisms

In a real-world scenario, although malicious and benign executables are given to be analyzed, it is not guaranteed that all of the features can be extracted from the given executables. The only modality that would always be available is the byte sequence. On the contrary, due to encryption and compression, the API function calls and the sequence of assembly language instructions might not be properly retrieved. For instance, there are some samples in the training set that have not been disassembled correctly or could not be disassembled and, consequently, their corresponding assembly language source file contains almost no instructions or no instructions at all (Hu et al., 2016). Thus, we have addressed this issue using modality dropout, which makes the network less sensitive to the loss of one or more channels of information. Modality dropout randomly drops one or more data modalities during training. Additionally, dropout has been applied to both fully-connected and convolutional layers, with a dropout rate equal to 0.5 and 0.1, respectively.

6. Evaluation

We deployed the proposed framework on a machine with an Intel Core i7-7700k CPU, 4xGeforce GTX 1080Ti GPUs and 64 Gb RAM. The GPUs are critical to accelerate the multimodal neural network algorithm. The modules of the framework and the machine learning algorithm have been implemented in Python and Tensorflow (Abadi et al., 2015). Due to the memory resource limitations we have reduced the mini-batch size to 8.

The generalization performance of our approach has been estimated using 10-fold cross validation. Two baseline classifiers were implemented, the Random Guess classifier and the Zero Rule classifier. The accuracy of a Random Guess classifier is calculated as follows:

$$acc = \frac{\sum_{i=1}^{c} p_i n_i}{\sum n_i}$$

where p_i is the probability to say "it is in the *i*th class" and n_i is the number of samples of class *i*. Thus, the accuracy of the Random Guess classifier is 0.1755. On the other hand, the Zero Rule classifier it simply outputs the majority class in the dataset. In particular, the accuracy of the Zero Rule classifier is $2942/10,868 = 0.2707$.

Instead of evaluating the model with accuracy alone, we selected the best model according to the macro F1-score. This is because accuracy can be a misleading measure in datasets were there exist a large class imbalance. For instance, a model can correctly predict the value of the majority class for all predictions and achieve a high classification accuracy while making mistakes on the minority and critical classes. The macro F1-score metric penalizes this kind of behavior by calculating the metrics for each label and finding their unweighted mean.

6.1. API-based component performance

Below is an in-depth analysis of the performance of various baseline algorithms to classify malware based on the use of Windows API function calls found in the assembly language source code. In particular,Tables 2–4 provide the 10-fold cross validation accuracy of various algorithms for different K values, where K refers to the K top features selected by either the $\tilde{\chi}^2$ or ANOVA-F feature selection algorithms. The baseline algorithms used in the experiment are logistic regression, support vector machines with linear or rbf kernel, random forests and lastly, gradient boosting. Table 2 presents their performance using as input a feature vector of 0s and 1s of size K, where K refers to the top K features according to the $\tilde{\chi}^2$ score and each value represents whether or not a particular API function has been invoked by the program. Table 3 shows the performance of the algorithms taking as input a feature vector of size K (top K features according to the $\tilde{\chi}^2$ score), where each value indicates the number of times a particular API function has been invoked. Table 4 presents the performance of the baseline algorithms taking as input a feature vector of size K, where K refers to the top K features according to the ANOVA-F

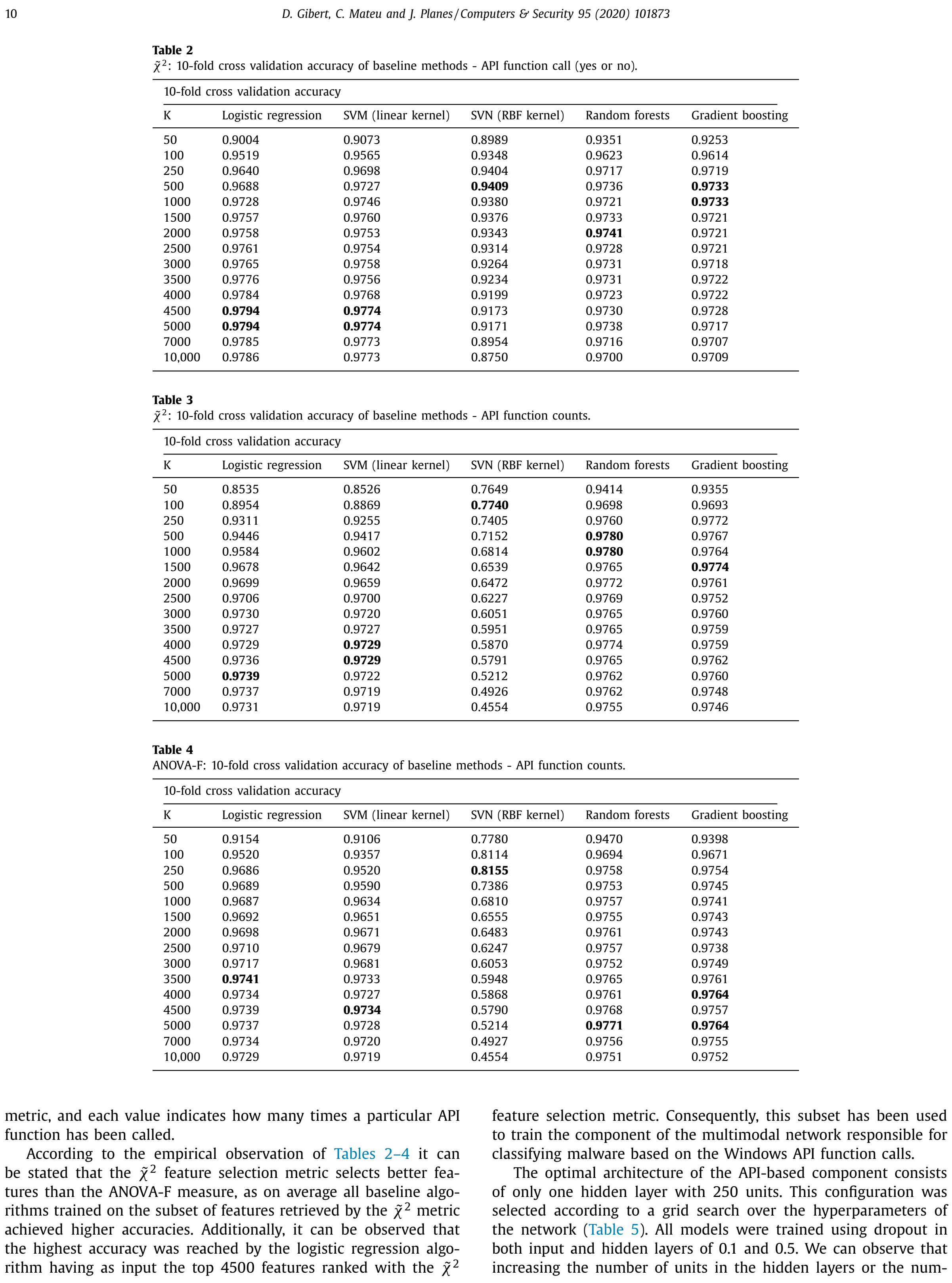


Table 2
$\tilde{\chi}^2$: 10-fold cross validation accuracy of baseline methods - API function call (yes or no).

10-fold cross validation accuracy					
K	Logistic regression	SVM (linear kernel)	SVN (RBF kernel)	Random forests	Gradient boosting
50	0.9004	0.9073	0.8989	0.9351	0.9253
100	0.9519	0.9565	0.9348	0.9623	0.9614
250	0.9640	0.9698	0.9404	0.9717	0.9719
500	0.9688	0.9727	**0.9409**	0.9736	**0.9733**
1000	0.9728	0.9746	0.9380	0.9721	**0.9733**
1500	0.9757	0.9760	0.9376	0.9733	0.9721
2000	0.9758	0.9753	0.9343	**0.9741**	0.9721
2500	0.9761	0.9754	0.9314	0.9728	0.9721
3000	0.9765	0.9758	0.9264	0.9731	0.9718
3500	0.9776	0.9756	0.9234	0.9731	0.9722
4000	0.9784	0.9768	0.9199	0.9723	0.9722
4500	**0.9794**	**0.9774**	0.9173	0.9730	0.9728
5000	**0.9794**	**0.9774**	0.9171	0.9738	0.9717
7000	0.9785	0.9773	0.8954	0.9716	0.9707
10,000	0.9786	0.9773	0.8750	0.9700	0.9709

Table 3
$\tilde{\chi}^2$: 10-fold cross validation accuracy of baseline methods - API function counts.

10-fold cross validation accuracy					
K	Logistic regression	SVM (linear kernel)	SVN (RBF kernel)	Random forests	Gradient boosting
50	0.8535	0.8526	0.7649	0.9414	0.9355
100	0.8954	0.8869	**0.7740**	0.9698	0.9693
250	0.9311	0.9255	0.7405	0.9760	0.9772
500	0.9446	0.9417	0.7152	**0.9780**	0.9767
1000	0.9584	0.9602	0.6814	**0.9780**	0.9764
1500	0.9678	0.9642	0.6539	0.9765	**0.9774**
2000	0.9699	0.9659	0.6472	0.9772	0.9761
2500	0.9706	0.9700	0.6227	0.9769	0.9752
3000	0.9730	0.9720	0.6051	0.9765	0.9760
3500	0.9727	0.9727	0.5951	0.9765	0.9759
4000	0.9729	**0.9729**	0.5870	0.9774	0.9759
4500	0.9736	**0.9729**	0.5791	0.9765	0.9762
5000	**0.9739**	0.9722	0.5212	0.9762	0.9760
7000	0.9737	0.9719	0.4926	0.9762	0.9748
10,000	0.9731	0.9719	0.4554	0.9755	0.9746

Table 4
ANOVA-F: 10-fold cross validation accuracy of baseline methods - API function counts.

10-fold cross validation accuracy					
K	Logistic regression	SVM (linear kernel)	SVN (RBF kernel)	Random forests	Gradient boosting
50	0.9154	0.9106	0.7780	0.9470	0.9398
100	0.9520	0.9357	0.8114	0.9694	0.9671
250	0.9686	0.9520	**0.8155**	0.9758	0.9754
500	0.9689	0.9590	0.7386	0.9753	0.9745
1000	0.9687	0.9634	0.6810	0.9757	0.9741
1500	0.9692	0.9651	0.6555	0.9755	0.9743
2000	0.9698	0.9671	0.6483	0.9761	0.9743
2500	0.9710	0.9679	0.6247	0.9757	0.9738
3000	0.9717	0.9681	0.6053	0.9752	0.9749
3500	**0.9741**	0.9733	0.5948	0.9765	0.9761
4000	0.9734	0.9727	0.5868	0.9761	**0.9764**
4500	0.9739	**0.9734**	0.5790	0.9768	0.9757
5000	0.9737	0.9728	0.5214	**0.9771**	**0.9764**
7000	0.9734	0.9720	0.4927	0.9756	0.9755
10,000	0.9729	0.9719	0.4554	0.9751	0.9752

metric, and each value indicates how many times a particular API function has been called.

According to the empirical observation of Tables 2–4 it can be stated that the $\tilde{\chi}^2$ feature selection metric selects better features than the ANOVA-F measure, as on average all baseline algorithms trained on the subset of features retrieved by the $\tilde{\chi}^2$ metric achieved higher accuracies. Additionally, it can be observed that the highest accuracy was reached by the logistic regression algorithm having as input the top 4500 features ranked with the $\tilde{\chi}^2$ feature selection metric. Consequently, this subset has been used to train the component of the multimodal network responsible for classifying malware based on the Windows API function calls.

The optimal architecture of the API-based component consists of only one hidden layer with 250 units. This configuration was selected according to a grid search over the hyperparameters of the network (Table 5). All models were trained using dropout in both input and hidden layers of 0.1 and 0.5. We can observe that increasing the number of units in the hidden layers or the num-

Table 5
Feed-forward neural network grid search. The architecture of a feed forward neural network is defined as follows: NN $F \times H_1 \times H_2$, where F is the size of the input feature vector, H_1 and H_2 is the number of neurons in the first and second hidden layers, respectively.

Algorithm	Accuracy	Macro F1-score
Random guess	0.1755	–
Zero rule	0.2707	–
Logistic regression	0.9810	0.9573
NN 4500 × 30	0.9824	0.9566
NN 4500 × 50	0.9826	0.9570
NN 4500 × 100	0.9829	0.9602
NN 4500 × 250	**0.9833**	**0.9621**
NN 4500 × 500	0.9829	0.9617
NN 4500 × 2000	0.9828	0.9602
NN 4500 × 2500	0.9825	0.9603
NN 4500 × 30 × 20	0.9823	0.9612
NN 4500 × 50 × 20	0.9822	0.9564
NN 4500 × 50 × 30	0.9824	0.9580
NN 4500 × 1000 × 100	0.9820	0.9603

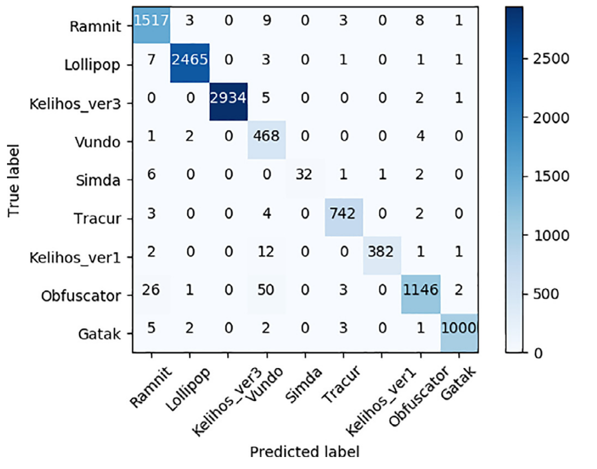


Fig. 9. NN 4500x250 confusion matrix.

ber of layers does not significantly improve the performance of the model and in particular, the highest accuracy and macro F1-score was achieved by a network of only 250 units.

Fig. 9 shows the confusion matrix of the 10-fold cross validation procedure. The major source of errors comes from the misclassification of samples belonging to the Obfuscator.ACY family. In particular, 82 out of the 1228 Obfuscator.ACY samples have been incorrectly classified.

6.2. Mnemonic-based component performance

Previously to training, the vocabulary (number of distinct mnemonics) was reduced to only consist of those mnemonics that appeared in at least three different executables. Those mnemonics that appeared less than three times in the training set were converted to the UNK token.

Initializing the word vectors with those vectors learned from an unsupervised learning model it has been a common practice in the literature (Collobert et al., 2011), as it improves the performance in tasks where there is no large training set available. Consequently, we initialized the mnemonic vectors using vectors of dimensionality 4 trained using either CBOW or Skip-Gram architecture (Mikolov et al., 2013). However, we did not observe any relevant improvement in either case. Figs. 12 and 11 show the confusion matrices for such architectures.

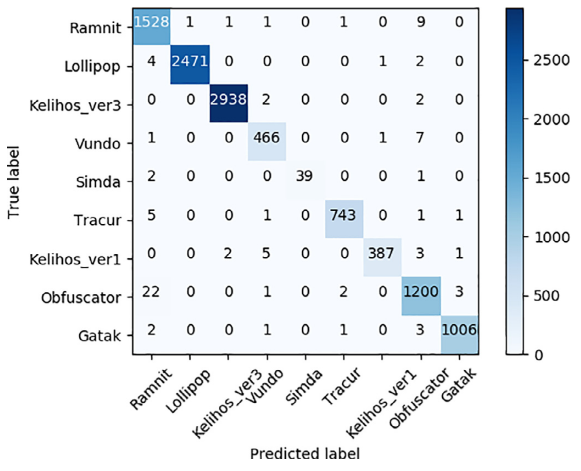


Fig. 10. CNN-rand confusion matrix.

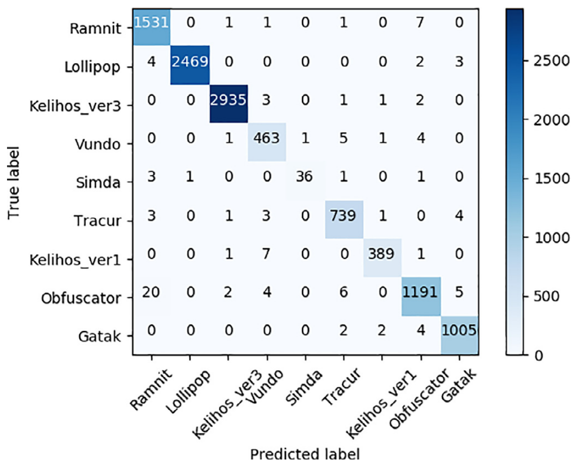


Fig. 11. CNN-skipgram confusion matrix.

In the next experiment, we test three different initialization settings for the mnemonic embeddings by computing the 10-fold cross validation accuracy and macro F1-score achieved by our convolutional neural network. The three settings are the following:

- CNN-rand. Baseline model where all mnemonic vectors are randomly initialized and then modified during training. See Fig. 10.
- CNN-skipgram. Baseline model with the mnemonic vectors initialized using the pretrained embeddings generated using the Skip-gram model. See Fig. 11.
- CNN-cbow. Baseline model with the mnemonic vectors initialized using the pretrained embeddings generated using the CBOW model. See Fig. 12.

According to the experiment, the model that achieved the highest cross-validation accuracy and macro F1-score is the one whose weights were randomly initialized from a uniform distribution (See Table 6). Therefore, to construct the multimodal network we decided not to initialize the embeddings with pretrained vectors. Nevertheless, all models achieved comparable results.

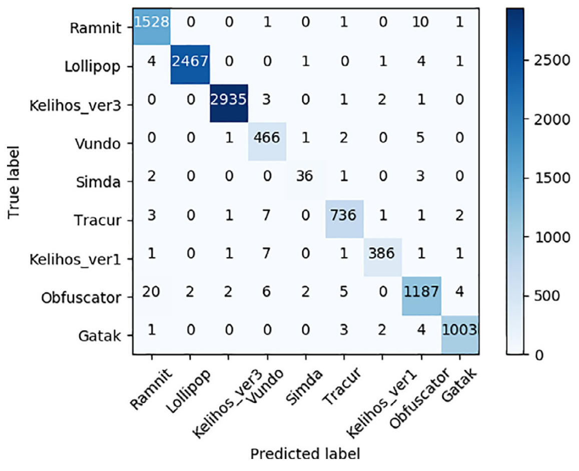


Fig. 12. CNN-cbow confusion matrix.

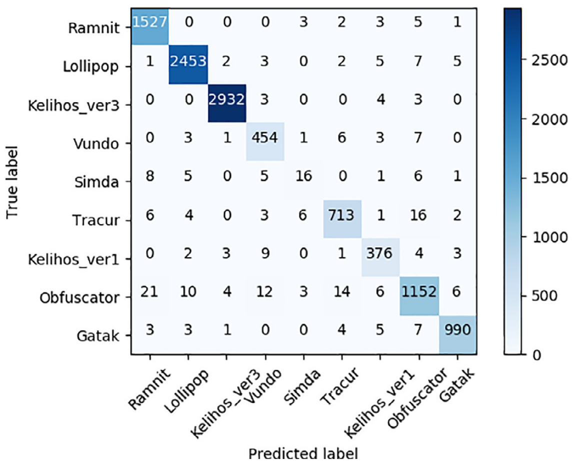


Fig. 14. DeepConv confusion matrix.

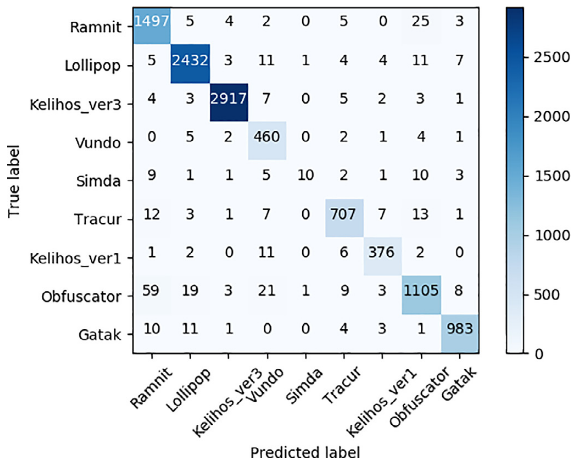


Fig. 13. MalConv confusion matrix.

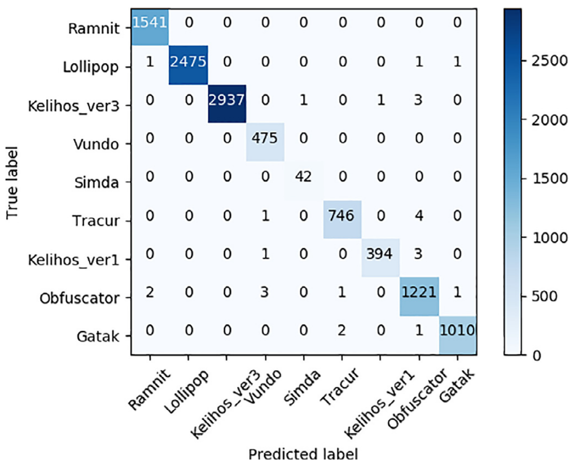


Fig. 15. HYDRA (pretraining, modality dropout) confusion matrix.

6.3. Byte-based component performance

In this section we compare the performance of the MalConv and DeepConv's model on the Microsoft Malware Classification Challenge dataset (Ronen et al., 2018). Figs. 13 and 14 display the confusion matrices reported from 10-fold cross-validation, whose macro F1-score is 0.8902 and 0.9071 for the MalConv and DeepConv model's, respectively (See Table 7). As a result, the byte-based component of our multimodal system would consist of the DeepConv architecture, given its superior performance.

The performance of these approaches is slightly worse than that of the approaches based on the assembly language source code. This is partially due to the high-dimensionality of the input sequence and the reduced training set, which make byte-based approaches suffer severely from overfitting. However, the hexadecimal representation of the malware's binary content is a very important source of information for any classifier as it is the minimal type of representation that can obtained from an executable. Depending on the obfuscation techniques employed by malware authors, the assembly language source code might not be retrieved correctly. Under these circumstances, the only information available is provided by the hexadecimal representation of the malware's binary content. Consequently, the information extracted from this modality of information is crucial to be able to correctly classify those malicious executables.

Table 6
Opcode-based CNN approaches comparison.

Approach	Accuracy	Macro F1-score
Random guess	0.1755	–
Zero rule	0.2707	–
CNN-rand	0.9917	0.9856
CNN-skipgram	0.9899	0.9770
CNN-cbow	0.9886	0.9717

Table 7
Bytes-based approaches comparison.

Approaches	Accuracy	Macro F1-score
Random guess	0.1755	–
Zero rule	0.2707	–
MalConv	0.9641	0.8902
DeepConv	0.9756	0.9071

Table 8
10-fold cross validation accuracy and macro F1-score comparison of modality-based and HYDRA models.

Model	Accuracy	Macro F1-score
API-based Feedforward Network	0.9833	0.9621
Assembly-based Shallow CNN	0.9917	0.9856
Bytes-based DeepConv	0.9756	0.8902
HYDRA	0.9871	0.9695
HYDRA (Pretraining, Modality Dropout)	**0.9975**	**0.9954**

6.4. Effectiveness of the multimodal deep learning model

In this section, we demonstrate the effectiveness of HYDRA by comparing it to the models trained on each modality independently. Fig. 15 shows the jointly reported accuracies of the 10-fold cross validation procedure to estimate the performance of HYDRA. The results of all models are shown in Table 8. We can observe that HYDRA's overall accuracy is 0.9975 and the macro F1-score is 0.9954, outperforming the modality-specific model's. Per-modality pretraining and multimodal dropout have been critical for the success of HYDRA. On the one hand, per-modality pretraining avoids overfitting one subset of features belonging to one modality and underfitting the features belonging to the others. On the other hand, multimodal dropout prevents the co-adaptation of the sub-networks to a specific feature type or modality of data.

6.4.1. Comparison with the state-of-the-art

To further evaluate the performance of our multimodal approach, we compared HYDRA with state-of-the-art methods in the literature that have evaluated their models on the dataset provided for the Kaggle's Microsoft Malware Classification Challenge. The results are shown in Table 9.

Methods in the literature are divided into various groups depending on the feature types used as input for the training algorithms. The groups are as follows:

- IMG-based approaches. This group consists of methods that take as input a grayscale image representing the malware's binary content (Gibert et al., 2018c) or a set of features extracted from it using any feature extractor technique (Ahmadi et al., 2016; Narayanan et al., 2016).
- Entropy-based approaches (Ahmadi et al., 2016; Gibert et al., 2018b) analyze the entropy and structural entropy representation of malware.
- Opcode-based approaches are split into two groups, (1) traditional approaches that extract n-gram features (Ahmadi et al., 2016) and (2) deep learning approaches (Gibert et al., 2017; 2019; McLaughlin et al., 2017), that take as input a sequence of

Table 9
10-fold cross validation accuracy of methods evaluated on the microsoft malware classification challenge.

Approach	Feature Type	Classification Algorithm	Accuracy	Macro F1
Random guess	–	–	0.1755	–
Zero rule	–	–	0.2707	–
Grayscale image				
Gibert et al. (2018c)	128 × 128 Grayscale Image	CNN	0.9750	0.9400
Ahmadi et al. (2016)	Haralick features	XGBoost	0.9690	0.9282
Ahmadi et al. (2016)	Local Binary Pattern features	XGBoost	0.9724	0.9530
Narayanan et al. (2016)	PCA features	1-NN	0.9660	0.9102
Entropy				
Gibert et al. (2018b)	Structural Entropy	Dynamic Time Warping +K-NN	0.9894	0.9813
Gibert et al. (2018b)	Structural Entropy	CNN	0.9828	0.9636
Ahmadi et al. (2016)	Entropy Statistical Measures	XGBoost	0.9900	0.9766
Sequence of opcodes				
Ahmadi et al. (2016)	1-Gram	XGBoost	0.9929	0.9906
McLaughlin et al. (2017)	Opcode sequence	CNN	0.9903	0.9743
Gibert et al. (2019)	Opcode sequence	Hierarchical CNN	0.9913	0.9830
OPCODE-based component	Opcode sequence	CNN	0.9917	0.9856
Sequence of bytes				
Ahmadi et al. (2016)	1-Gram	XGBoost	0.9850	0.9678
Raff et al. (2018)	Bytes sequence	MalConv	0.9641	0.8894
BYTE-based component (Krčál et al., 2018)	Bytes sequence	DeepConv	0.9756	0.9089
Le et al. (2018)*	Scaled bytes sequence	CNN	0.9647	0.9341
Le et al. (2018)*	Scaled bytes sequence	CNN+Unidirectional LSTM	0.9800	0.9577
Le et al. (2018)*	Scaled bytes sequence	CNN+Bidirectional LSTM	0.9814	0.9662
Gibert et al. (2018a)	Bytes sequence	Denoising Autoencoder + Dilated Residual Network	0.9861	0.9719
Yousefi-Azar et al. (2017)	1-Gram	Autoencoder + XGBoost	0.9309	0.8664
API invocations				
Ahmadi et al. (2016)	API feature vector (796)	XGBoost	0.9868	0.9638
API-based component	API feature vector (4500)	Feed-forward network	0.9833	0.9621
Multiple features				
Zhang et al. (2016)	Total lines of each Section, Operation Code Count, API Usage, Special Symbols Count, Asm File Pixel Intensity Feature, Bytes File Block Size Distribution, Bytes File N-Gram	Ensemble Learning (XGBoost)	0.9974	0.9938
Ahmadi et al. (2016)	ENT, Bytes 1-G, STR, IMG1, IMG2, MD1, MISC, OPC, SEC, REG, DP, API, SYM, MD2	Ensemble Learning (XGBoost)	0.9976	0.9931
Mays et al. (2017)	IMG and Opcode N-Grams	Ensemble Learning (CNN and NN)	0.9724	0.9618
HYDRA	**APIs, Bytes sequence, Opcode sequence**	**Multimodal Deep Neural Network**	**0.9975**	**0.9951**

opcodes representing the malware's assembly language source code.

- Byte-based approaches are split similarly to opcode-based approaches. On the one hand, there are those approaches that extract n-gram features from the bytes sequence (Ahmadi et al., 2016). On the other hand, deep learning approaches jointly learn to extract features and classify malware during training (Krčál et al., 2018; Raff et al., 2018). Furthermore, it includes methods that learn an encoded representation of malware using autoencoders (Gibert et al., 2018a; Yousefi-Azar et al., 2017).
- API-based approaches (Ahmadi et al., 2016) generate a feature set by mining the API calls that a classifier uses to make predictions.
- Multimodal learning refers to those approaches that learn to detect malware using information from multiple modalities (Ahmadi et al., 2016; Mays et al., 2017; Zhang et al., 2016).

Furthermore, note that the most used classification algorithms are either neural networks or gradient boosting. On the one hand, neural networks and in particular convolutional neural networks have recently attracted the academic community due to their advantages on processing raw data and their ability to learn features by themselves. On the other hand, gradient boosting and, in particular, the XGBoost library have until recently provided unmatched performance in tasks that rely on feature engineering and domain-specific knowledge. This trend might change in the near future due to the availability of bigger training feeds for the research community and developments and improvements in the multimodal learning field. As observed in Table 9, HYDRA achieved comparable results to (Zhang et al., 2016) and (Ahmadi et al., 2016) with fewer input modalities and achieved a higher detection rate and macro F1-score than (Mays et al., 2017) and the remaining deep learning and feature engineering based approaches in the literature. Thus, we demonstrate that end-to-end learning systems can be successfully complemented with hand-engineered features to achieve state-of-the-art results in the malware classification task, where domain-specific knowledge has been the way-to-go for building systems.

7. Conclusions

In this paper, we present a novel malware classification framework that combines both hand-engineered features and end-to-end components in a modular architecture. To the best of our knowledge, this research is the first application of multimodal deep learning for malware classification, and in particular Portable Executable files. The multimodal approach learns and combines characteristics of malware from various sources of information, yielding to higher classification performance than those classifiers that take as input only a single modality of data. Three kinds of modalities are extracted by analyzing the hexadecimal representation of malware's binary content and its disassembly counterpart, (1) the list of API functions invoked, (2) the sequence of mnemonics representing malware's assembly language source code, and (3) the sequence of bytes representing malware's binary content. This architecture can be enriched with many more feature types to express executables' characteristics and it serves as a baseline model for future improvements. Furthermore, by fusing hand-crafted features with the end-to-end components we are able to mix the strengths of both approaches, characterizing domain-specific features and the ability of deep learning to automatically extract a set of descriptive features without relying on domains' knowledge.

Reported results allow the effectiveness of our approach to be assessed with respect to state-of-the-art techniques. The detection accuracy and macro f1-score of our multimodal deep learning architecture is comparable to gradient boosting methods based on feature engineering, with ours only relying on three basic types of features from Portable Executables, and far more accurate than deep learning approaches in the literature.

7.1. The problem of concept drift

Machine learning techniques were originally designed for stationary environments in which the training and test sets are assumed to be generated from the same statistical distribution. In a stationary environment, a model will approximate a mapping function $f(x)$ given input data x to predict an output value y, $y = f(x)$, and it is assumed that the mapping learned from the data will be valid in the future and the relationship between input and output do not change over time. However, this assumption is not valid in the malware domain. Software applications, including malware, evolve over time due to changes resulting from adding features and capabilities, fixing bugs, porting to new platforms, etc. Additionally, versions of the same software are expected to be similar to previous versions with few exceptions. Thus, the similarity between previous and future versions will degrade slowly over time. This is known as the problem of concept drift. Concept drift is the problem of the changing underlying relationships in the data. This will result in the decay of the prediction quality of malware detectors and classifiers over time as malware evolves and new variants appear (Pendlebury et al., 2019).

Furthermore, malware is constantly pushed to evolve in order to avoid detection by anti-malware engines and be able to infect new hosts. Thus, malware authors are well-motivated to intentionally craft adverarial examples (Huang et al., 2011) using a wide range of obfuscation techniques (You and Yim, 2010). As a result, the aforementioned issues have to be taken into account in the process of building a sustainable model for malware detection and classification (Jordaney et al., 2017).

7.2. Future work

One future line of research could be the implementation of new architectures to detect and classify malware from their binary content represented as a sequence of bytes, as current methods perform below average in comparison with the rest of approaches in the literature. A second line of research could be to study the incorporation of more data modalities or feature types into the current multimodal architecture and analyze its impact on the performance of the system. A third line of research could be the study of explainable artificial intelligence (XAI) techniques to interpret the results of machine learning models for malware detection and classification in order to help security researchers in the process of malware analysis.

Declaration of Competing Interest

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

CRediT authorship contribution statement

Daniel Gibert: Conceptualization, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing. **Carles Mateu:** Supervision, Writing - original draft, Writing - review & editing, Resources, Funding acquisition. **Jordi Planes:** Supervision, Writing - original draft, Writing - review & editing, Resources, Funding acquisition.

Acknowledgments

This research has been partially funded by the Spanish MICINN Projects TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, and is supported by the University of Lleida. This research article has received a grant (2019 call) from the University of Lleida Language Institute to review the English.

References

Aafer, Y., Du, W., Yin, H., 2013. DroidAPIMiner: mining API-level features for robust malware detection in android. In: Zia, T., Zomaya, A., Varadharajan, V., Mao, M. (Eds.), Security and Privacy in Communication Networks. Springer International Publishing, Cham, pp. 86–103.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X., 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., Giacinto, G., 2016. Novel feature extraction, selection and fusion for effective malware family classification. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 183–194. doi:10.1145/2857705.2857713.

Bidoki, S.M., Jalili, S., Tajoddin, A., 2017. PbMMD: a novel policy based multi-process malware detection. Eng. Appl. Artif. Intell. 60, 57–70. doi:10.1016/j.engappai.2016.12.008.

Chandrasekar, K., Cleary, G., Cox, O., Lau, H., Nahorney, B., Gorman, B.O., O'Brien, D., Wallace, S., Wood, P., Wueest, C., 2017. Internet Security Threat Report. Technical Report. Symantec Corporation.

Cleary, G., Corpin, M., Cox, O., Lau, H., Nahorney, B., O'Brien, D., O'Gorman, B., Power, J.-P., Wallace, S., Wood, P., Wueest, C., 2018. Internet Security Threat Report. Technical Report. Symantec Corporation.

Clevert, D., Unterthiner, T., Hochreiter, S., 2015. Fast and accurate deep network learning by exponential linear units (ELUS). CoRR arXiv:1511.07289

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P., 2011. Natural language processing (almost) from scratch. J. Mach. Learn. Res. 12, 2493–2537.

Daniely, Y., Clayton, R., Johnson, S., Eisner, T., Fishman, Y., Kaye, J., 2018. Security Report. Technical Report. Check Point Software Technologies Ltd.

Ghiasi, M., Sami, A., Salehi, Z., 2015. Dynamic VSA: a framework for malware detection based on register contents. Eng. Appl. Artif. Intell. 44, 111–122. doi:10.1016/j.engappai.2015.05.008.

Gibert, D., Bejar, J., Mateu, C., Planes, J., Solis, D., Vicens, R., 2017. Convolutional neural networks for classification of malware assembly code. In: International Conference of the Catalan Association for Artificial Intelligence, pp. 221–226. doi:10.3233/978-1-61499-806-8-221.

Gibert, D., Mateu, C., Planes, J., 2018. An end-to-end deep learning architecture for classification of malware's binary content. In: Artificial Neural Networks and Machine Learning - ICANN 2018 - 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4–7, 2018, Proceedings, Part III, pp. 383–391. doi:10.1007/978-3-030-01424-7_38.

Gibert, D., Mateu, C., Planes, J., 2019. A hierarchical convolutional neural network for malware classification. In: The International Joint Conference on Neural Networks 2019. IEEE, pp. 1–8.

Gibert, D., Mateu, C., Planes, J., Vicens, R., 2018. Classification of malware by using structural entropy on convolutional neural networks. In: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2–7, 2018, pp. 7759–7764.

Gibert, D., Mateu, C., Planes, J., Vicens, R., 2018. Using convolutional neural networks for classification of malware represented as images. J. Comput. Virol. Hacking Tech. doi:10.1007/s11416-018-0323-0.

Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. In: In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics.

Hassen, M., Carvalho, M.M., Chan, P.K., 2017. Malware classification using static analysis based features. In: 2017 IEEE Symposium Series on Computational Intelligence (SSCI), pp. 1–7. doi:10.1109/SSCI.2017.8285426.

Hassen, M., Chan, P.K., 2017. Scalable function call graph-based malware classification. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 239–248. doi:10.1145/3029806.3029824.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2012. Improving neural networks by preventing co-adaptation of feature detectors. CoRR arXiv:1207.0580

Hu, X., Jang, J., Wang, T., Ashraf, Z., Stoecklin, M.P., Kirat, D., 2016. Scalable malware classification with multifaceted content features and threat intelligence. IBM J. Res. Dev. 60 (4), 6:1–6:11. doi:10.1147/JRD.2016.2559378.

Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I., Tygar, J.D., 2011. Adversarial machine learning. In: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence. Association for Computing Machinery, New York, NY, USA, pp. 43–58. doi:10.1145/2046684.2046692.

Jain, S., Meena, Y.K., 2011. Byte level n–gram analysis for malware detection. In: Venugopal, K.R., Patnaik, L.M. (Eds.), Computer Networks and Intelligent Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 51–59.

Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L., 2017. Transcend: detecting concept drift in malware classification models. In: 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, BC, pp. 625–642.

Khan, R.U., Zhang, X., Kumar, R., 2018. Analysis of ResNet and GoogleNet models for malware detection. J. Comput. Virol. Hacking Tech. doi:10.1007/s11416-018-0324-z.

Kinable, J., Kostakis, O., 2011. Malware classification based on call graph clustering. J. Comput. Virol. 7 (4), 233–245. doi:10.1007/s11416-011-0151-y.

Klambauer, G., Unterthiner, T., Mayr, A., Hochreiter, S., 2017. Self-normalizing neural networks. CoRR arXiv:1706.02515

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

Le, Q., Boydell, O., Namee, B.M., Scanlon, M., 2018. Deep learning at the shallow end: malware classification for non-domain experts. Digit. Invest. 26, S118–S126. doi:10.1016/j.diin.2018.04.024.

Lyda, R., Hamrock, J., 2007. Using entropy analysis to find encrypted and packed malware. IEEE Secur. Priv. 5 (2), 40–45. doi:10.1109/MSP.2007.48.

Mays, M., Drabinsky, N., Brandle, S., 2017. Feature selection for malware classification. In: Proceedings of the 28th Modern Artificial Intelligence and Cognitive Science Conference 2017, Fort Wayne, IN, USA, April 28–29, 2017., pp. 165–170.

McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., Joon Ahn, G., 2017. Deep android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. ACM, New York, NY, USA, pp. 301–308. doi:10.1145/3029806.3029823.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2. Curran Associates Inc., USA, pp. 3111–3119.

Moskovitch, R., Stopel, D., Feher, C., Nissim, N., Elovici, Y., 2008. Unknown malcode detection via text categorization and the imbalance problem. In: 2008 IEEE International Conference on Intelligence and Security Informatics, pp. 156–161. doi:10.1109/ISI.2008.4565046.

Narayanan, B.N., Djaneye-Boundjou, O., Kebede, T.M., 2016. Performance analysis of machine learning and pattern recognition algorithms for malware classification. In: 2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS), pp. 338–342. doi:10.1109/NAECON.2016.7856826.

Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S., 2011. Malware images: Visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security. ACM, New York, NY, USA, pp. 4:1–4:7. doi:10.1145/2016904.2016908.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., 2019. TESSERACT: eliminating experimental bias in malware classification across space and time. In: 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA, pp. 729–746.

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.K., 2018. Malware detection by eating a whole EXE. In: The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2–7, 2018., pp. 268–276.

Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M., 2018. Microsoft malware classification challenge. CoRR arXiv:1802.10135

Salehi, Z., Sami, A., Ghiasi, M., 2017. Maar: robust features to detect malicious activity based on API calls, their arguments and return values. Eng. Appl. Artif. Intell. 59, 93–102. doi:10.1016/j.engappai.2016.12.016.

Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., Hamze, A., 2010. Malware detection based on mining API calls. In: Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 1020–1025. doi:10.1145/1774088.1774303.

Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G., 2013. Opcode sequences as representation of executables for data-mining-based unknown malware detection. Information Sciences 231, 64–82. doi:10.1016/j.ins.2011.08.020. Data Mining for Information Security

Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., Elovici, Y., 2012. Detecting unknown malicious code by applying classification techniques on opcode patterns. Secur. Inform. 1 (1), 1. doi:10.1186/2190-8532-1-1.

Sorokin, I., 2011. Comparing files using structural entropy. J. Comput. Virol. 7 (4), 259. doi:10.1007/s11416-011-0153-9.

Souri, A., Hosseini, R., 2018. A state-of-the-art survey of malware detection approaches using data mining techniques. Hum. Centric Comput. Inf. Sci. 8 (1), 3. doi:10.1186/s13673-018-0125-x.

Ucci, D., Aniello, L., Baldoni, R., 2019. Survey of machine learning techniques for malware analysis. Comput. Secur. 81, 123–147. doi:10.1016/j.cose.2018.11.001.

Vulnerabilities, C., Exposures, 2016. CVE-2017-0143.Available from MITRE, CVE-ID CVE-2017-0143.

Wojnowicz, M., Chisholm, G., Wolff, M., Zhao, X., 2016. Wavelet decomposition of software entropy reveals symptoms of malicious code. J. Innov. Digit. Ecosyst. 3 (2), 130–140. doi:10.1016/j.jides.2016.10.009.

You, I., Yim, K., 2010. Malware obfuscation techniques: a brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, pp. 297–300. doi:10.1109/BWCCA.2010.85.

Yousefi-Azar, M., Varadharajan, V., Hamey, L., Tupakula, U., 2017. Autoencoder-based feature learning for cyber security applications. In: 2017 International Joint Conference on Neural Networks (IJCNN), pp. 3854–3861. doi:10.1109/IJCNN.2017.7966342.

Yuxin, D., Siyi, Z., 2017. Malware detection based on deep learning algorithm. Neural Comput. Appl. doi:10.1007/s00521-017-3077-6.

Zhang, Y., Huang, Q., Ma, X., Yang, Z., Jiang, J., 2016. Using multi-features and ensemble learning method for imbalanced malware classification. In: 2016 IEEE Trustcom/BigDataSE/ISPA, pp. 965–973. doi:10.1109/TrustCom.2016.0163.

Daniel Gibert is a Ph.D. student at the Department of Computer Science and Industrial Engineering of the University of Lleida, in Spain. He graduated in 2014 and received a Masters degree in Artificial Intelligence in 2016 from the Polytechnic University of Catalonia. His research interests include intrusion detection and machine learning.

Carles Mateu received his B.Sc. from Universitat de Lleida, M.Sc. from Open University of Catalonia, and Ph.D. from University of Lleida in 2009. He is currently an Associate Professor at the University of Lleida. His research interests include Security, Energy Storage, Energy Efficiency and Artificial Intelligence.

Jordi Planes received his B.Sc. from Universitat de Lleida, M.Sc. from University Rovira i Virgili, and Ph.D. from University of Lleida in 2007. He is currently an Associate Professor at the University of Lleida. His research interests include Applications of Neural Networks.

Chapter 9

Discussion

This section introduces the main issues and challenges faced by security researchers. It is structured as follows: Section 9.1 reviews the availability of public benchmarks of malware for research. Section 9.2 presents the problem of class imbalance. Section 9.3 presents the problem of concept drift. Section 9.4 introduces the problem of adversarial learning and presents an exhaustive evaluation of machine learning approaches to common obfuscation techniques. Lastly, Section 9.5 gives some insights about the interpretability of the models.

9.1 Open and Public Benchmarks

Through the development of the thesis, the performance of the approaches presented were mainly evaluated on the dataset [66] provided by Microsoft for the Big Data Innovators Gathering Challenge. This is not a mere coincidence. The task of malware detection and classification has not received the same attention in the research community as other applications, where rich labeled datasets exist, including image classification, speech recognition, etc. Due to legal restrictions, benign binaries are not shared, as they are often protected by copyright laws and thus, researchers cannot share the binaries used in their research. On the other hand, malicious binaries are shared through web sites such as VirusShare. Nevertheless, unlike other domains where data may be labeled very quickly and in many cases by a non-expert, determining whether a file or its corresponding family or class is malicious can be a time-consuming process, even for security experts. Furthermore, services like VirusTotal [1] specifically restrict sharing the vendor anti-malware labels to the public.

The aforementioned issues render it impossible to meaningfully compare accuracy numbers across works, as different datasets are used with different labeling procedures. At the present time, the only standard benchmark available to the research community regarding Windows Portable Executables that includes the raw binaries is the one provided by Microsoft [66] for the Big Data Innovators Gathering Anti-Malware Prediction Challenge. There exist other datasets such as MalIMG [59] and EMBER [4] but none of them include the raw binaries and thus, experiments using featureless deep-learning malware detectors are precluded.

[1] `https://www.virustotal.com/en`

9.2 Class Imbalance

Obtaining good training data is one of the most challenging aspects of any machine learning problem. Machine learning classifiers are only as good as the data used to train them, and reliable labeled data are especially important for the task of malware detection, where the process of labeling a file can be a very time-consuming process.

Additionally, there are various disciplines including fraud detection, malware detection, malware classification, medical diagnosis, etc, where it is common to have a disproportional number of samples per class. For instance, the number of benign samples might not be proportionally equal to the number of malicious samples, or the number of samples belonging to one family might far exceed the number of samples from other families. Data with highly imbalanced class distributions are said to suffer the class imbalance problem [42, 35]. That is, this kind of distribution, where one class is much larger that the other(s) can lead to a model that predicts the value of the majority classes for all predictions and still achieve high classification accuracy while lacking predictive power. In other words, the classifier might be biased towards the majority classes and achieve very poor classification rates in the minority classes. It might happen that the classifier predicts everything as the major class and ends up ignoring the minor classes. This is called the accuracy paradox. In these cases, accuracy is a misleading measure. It may be desirable to select a less accurate model but with greater predictive power. For problems like this, additional measures are required to evaluate a classifier such as precision 9.1, recall 9.2 and the F1 score 9.3. Alternatively, the Receiver Operating Characteristic (ROC) curve graphically illustrates the discriminative ability of a binary classifier.

Precision (P) is the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p).

$$P = \frac{T_p}{T_p + F_p}. \tag{9.1}$$

Recall (R) is the number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n).

$$R = \frac{T_p}{T_p + F_n}. \tag{9.2}$$

The F1 score is the weighted average of precision, defined as following:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}. \tag{9.3}$$

Finally, the ROC curve is created by plotting the True Positive Rate (TPR) or recall against the False Positive Rate. The FPR is also known as the probability of false alarm and can be calculated as (1-Specificity) where Specificity is equal to $\frac{TN}{TN+FP}$. The higher the AUC, the better the model is at predicting the correct label of classes.

9.3 Concept Drift

In the machine learning literature, the term "concept drift" has been used to describe the problem of the changing underlying relationships in the data. Supervised learning is the machine learning task of learning a function that maps an input to an output based on a set of input-output samples. Technically speaking, it is the problem of approximating a mapping function (f) given input data (x) to predict an output value (y), $y = f(x)$. Traditional machine learning applications such as digit classification, text categorization or speech recognition, assume that training data is sampled from a stationary population. In other words, they assume that the mapping learning from historical data will be valid for new data in the future and that the relationships between input and output do not change over time. This is not true for the problem of malware detection and classification.

Software applications, including malware, naturally evolve over time due to changes resulting from adding features, fixing bugs, porting to new environments and platforms [53]. These changes are expected to be introduced relatively infrequently. Additionally, successive versions of the software are expected to be highly similar to previous versions, with few exceptions such as when the code base undergoes significant refactoring and there are changes in the compilers or libraries linked to the software. Moreover, the similarity between previous and future versions is expected to degrade slowly over time. Consequently, the prediction quality decays over time as malware evolves and new variants and families appear [43]. Thus, in order to build high-quality models for malware detection and classification, it is important to identify when the model shows signs of degradation and thereby it fails to recognize new malware. Existing solutions [44, 22] aim at periodically retrain the model with the hope that it will automatically adapt to changes in malware over time. The process of retraining the model can be done from scratch, partially and incrementally, where incremental retraining refers to the process of retraining a given model with new labeled malware samples and all previous training samples without forgetting the knowledge obtained from prior datasets.

9.4 Adversarial Learning

Malware is pushed to evolve in order to survive and operate. That is, malicious software has to constantly evolve to avoid detection by anti-malware engines. Consequently, malware writers are well-motivated to intentionally seek evasion by employing a wide range of obfuscation techniques [75, 60].

To put it in the machine learning context, an attacker's aim is to fool the machine learning detector by camouflaging a piece of malware in feature space by inducing a feature representation highly correlated to benign behavior. The ability of the attacker to bypass machine learning solutions is related to their knowledge about features and machine learning models to target. For instance, consider a machine learning approach that relies on the program's invocations of API functions or the DLLs dynamically loaded by the executable. An attacker might use this information to conceal the usage of any suspicious API function by packing the executable and leaving only the stub of the import table or perhaps even no import table at all. These modifications to the feature space may or may not be performed manually.

Adversarial machine learning [40] is a technique employed to attempt to fool machine learning by automatically crafting adversarial examples, that is, samples with small, intentional feature perturbations that cause a machine learning model to make an incorrect prediction. Machine learning-based detectors are vulnerable to adversarial examples, and the application of machine learning to the cybersecurity domain does not constitute an exception. For a detailed overview of the evolution of adversarial machine learning over the past decade we refer to [9]. They reviewed the work done in the context of various applications, including computer security and its arms race notion and proposed a comprehensive threat model that accounts for the presence of the attacker during the system design. Recent classifiers proposed for malware detection, have indeed been shown to be easily fooled by well-crafted adversarial manipulations [17, 11, 39, 70, 55]. [11] explored adversarial machine learning to attack a malware detector based on the input of Windows Application Programming Interface (API) calls extracted from the PE files. [70] analyzed various append-based strategies to generate adversarial examples to conceal malware and bypass the MalConv [63] model. Furthermore, [17] proposed a novel attack algorithm to generate adversarial malware binaries which only change a few tens of bytes of the file header. Their algorithm was evaluated against MalConv. They found that MalConv learns discriminative features mostly from the characteristics of the file header and used their findings to exploit and bypass the model. Contrarily, [55] explored the types of adversarial attacks that have exploited the vulnerabilities of the components of PDFs to bypass malware detectors, including JavaScript-based attacks, ActionScript-based attacks and file embedding-based attacks.

9.4.1 Auditing Static Machine Learning Anti-Malware Tools against Metamorphic Attacks

However, most state-of-the-art approaches [17, 48, 70] investigated how slight permutations generated by appending bytes at the end of sections or at the end of the file can produce misclassifications. Unfortunately, these approaches are based on modifications that only affect the structure of the Portable Executable files. None of them modify the actual source code of the executables. Thus, they greatly differ from the modifications performed by real-word malware to generate variants of itself.

To fill this gap, the following manuscript provides an extensive evaluation of state-of-the-art detectors powered by machine learning (M.L.) against common obfuscation techniques. More specifically, the performance of the M.L. approaches is assessed against the modifications performed by the following metamorphic techniques:

- The dead code insertion technique.
- The registers reassignment technique.
- The subroutine reordering technique.
- The code reordering through jumps technique.

The manuscript aims to address the limitations found in deep learning approaches in the literature [63, 49, 31] by proposing a shallow architecture that improves 14.95% and 16.38% with respect to MalConv [63] and DeepConv [49] architectures. Instead of learning complex and deep patterns, our architecture learns N-gram like features from the malware's binary content represented as a sequence of bytes. This is achieved through a convolutional layer with filters of various sizes that act as feature extractors. Furthermore, the usage of the aforementioned metamoorphic techniques to augment the dataset and reduce class imbalance is investigated. The generalization performance of the M.L. approaches has been evaluated on a standard public benchmark provided by Microsoft for the Big Data Innovators Gathering Anti-Malware Prediction Challenge [66] for reproducibility purposes.

The manuscript has been submitted to a journal belonging to the First Quartile (Q1) as classified by Scimago Journal Rank and it is currently under revision.

Auditing Static Machine Learning Anti-Malware Tools Against Metamorphic Attacks

Daniel Gibert[a,*], Carles Mateu[a], Jordi Planes[a], Joao Marques-Silva[b]

[a] *University of Lleida, Jaume II, 69, Lleida, Spain*
[b] *Université Fédérale Toulouse Midi-Pyrénées, 41 Allées Jules Guesde, Toulouse, France*

Abstract

Malicious software is one of the most serious cyber threats on the Internet today. Traditional malware detection has proven unable to keep pace with the sheer number of malware because of their growing complexity, new attacks and variants. Most malware implement various metamorphic techniques in order to disguise themselves, therefore preventing successful analysis and thwarting the detection by signature-based anti-malware engines. During the past decade, there has been an increase in the research and deployment of anti-malware engines powered by machine learning, and in particular deep learning, due to their ability to handle huge volumes of malware and generalize to never-before-seen samples. However, there is little research about the vulnerability of these models to adversarial examples. To fill this gap, this paper presents an exhaustive evaluation of the state-of-the-art approaches for malware classification against common metamorphic attacks. Given the limitations found in deep learning approaches, we present a simple architecture that increases 14.95% the classification performance with respect to MalConv's architecture. Furthermore, the use of the metamorphic techniques to augment the training set is investigated and results show that it significantly improves the classification of malware belonging to families with few samples.

Keywords: Malware Analysis, Malware Classification, Software Obfuscation, N-gram Extraction, Machine Learning, Deep Learning

1. Introduction

In today's ever-connected society, cyber-attacks have been dramatically increasing in number and damage up to the point that cyberthreats are ranked as a consistent and persistent threat among the top global risks[1], along with weather extremes, climate change and natural disasters. Some estimates [2] predict that the cost of cyber-crime to the world would be 6 trillion an-

*I am corresponding author
Email addresses:
`daniel.gibert@diei.udl.cat` (Daniel Gibert), `carlesm@diei.udl.cat` (Carles Mateu), `jplanes@diei.udl.cat` (Jordi Planes), `joao.marques-silva@univ-toulouse.fr` (Joao Marques-Silva)

[1] `http://www3.weforum.org/docs/WEF_Global_Risks_Report_2019.pdf`
[2] `https://cybersecurityventures.com/`

Preprint submitted to Computers & Security *November 15, 2020*

nually by 2021, rising from the 3 trillion in 2015. This estimate includes damage and destruction of data, stolen money, lost productivity, theft of personal, financial data and intellectual property, fraud, disruption of the normal course of business, forensic investigation and reputational harm. According to MalwareBytes [3], there has been an increase of 13% in the business threat detections in 2019 with a dramatic spike in detections of the malware Emotet at the beginning of the year.

Malicious software is one of the most common methods employed by cybercriminals to launch a cyberattack. Other methods include, but are not limited to, phishing, man-in-the-middle attack, SQL injection, etc. Every day, the AV-TEST Institute registers over 350000 new malicious programs and potentially unwanted applications (PUA). In fact, the number of total malware has more than doubled from the 470.01m in 2015 to 1065.61m in 2020[4]. However, this is not due to an increase in new malware but to the reuse of well-established families through the usage of code obfuscation techniques to bypass detection engines. Thus, to keep up with malware and be able to reduce its impact, it is necessary to improve the computer systems' cyberdefenses and in particular, anti-malware engines, the last layer of defense against a cyberattack and the defensive layer responsible of preventing, detecting and removing malicious software.

The fastest and most reliable method employed by anti-malware engines to detect known malware is by means of unique signatures. Signatures are composed by sequences of bytes or data, to provide an identifier for each malicious software or group of samples with similar capabilities or behavior. However, signatures cannot detect against unknown malware because a new signature has to be developed previously. Consequently, signature-based detection only protects against known malware. Another problem is that malware can alter its signature to avoid detection by simply modifying the code while preserving its functionality and behavior. Furthermore, as new malware appears every day, it is necessary to store large amounts of signatures, demanding considerable storage, making slow to search a particular signature, and affecting system performance (Amro and Alkhalifah, 2015).

Due to the sheer volumes of new malware variants being deployed every day, anti-malware solutions that rely solely on signatures have become obsolete. During the past decade, there has been an increase in the research and deployment of anti-malware engines powered by machine learning (Gibert et al., 2020b; Ucci et al., 2019; Souri and Hosseini, 2018) to complement signature-based detection due to their ability to handle huge volumes of data and generalize to never-before-seen malware. This is achieved by summarizing complex relationships among features that are discriminative between malware and goodware or between malware families, allowing the detection engine to adapt to the modifications in malware's code. However, there is little research about the susceptibility of these models to adversarial samples. Most state-of-the-art approaches (Demetrio et al., 2019; Kolosnjaji et al., 2018; Suciu et al., 2019) investigated how slight permutations

[3]`https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf`

[4]`https://www.av-test.org/en/statistics/malware/`

2

generated by appending bytes at the end of sections or at the end of the file can produce misclassifications. Unfortunately, these approaches are based on modifications that only affect the structure of the Portable Executable files. None of them modify the actual source code of the executables. Thus, they greatly differ from the modifications performed by real-word malware to generate variants of itself.

The aim of this paper is to fill this gap. To this end, this work provides an extensive evaluation of state-of-the-art detectors powered by machine learning (ML) against common metamorphic techniques. More specifically, the performance of the ML approaches is assessed against the modifications performed by the following metamorphic techniques:

- The dead code insertion technique.
- The registers reassignment technique.
- The subroutine reordering technique.
- The code reordering through jumps technique.

Given the poor performance of the byte-based deep learning approaches in the literature (Raff et al., 2018a; Krčál et al., 2018; Gibert et al., 2018) in comparison to the opcode-based approaches (Gibert et al., 2017) due to their greater complexity and the length of the input data, we propose a shallow architecture that improves 14.95% and 16.38% with respect to MalConv (Raff et al., 2018a) and DeepConv (Krčál et al., 2018) architectures. Instead of learning complex and deep patterns, our architecture learns n-gram like features from the malware's binary content represented as a sequence of bytes. This is achieved through a convolutional layer with filters of various sizes that act as feature extractors. Furthermore, we investigate the usage of the aforementioned metamorphic techniques to augment the dataset and reduce class imbalance. The generalization performance of the ML approaches has been evaluated on a standard public benchmark provided by Microsoft for the Big Data Innovators Gathering Anti-Malware Prediction Challenge (Ronen et al., 2018) for reproducibility purposes.

The rest of the paper is organized as follows. Section 2 provides the background. Section 3 introduces state-of-the-art approaches to bypass ML malware detectors. Section 4 describes the metamorphic techniques employed by malware authors to modify the executables. Section 5 presents the ML approaches evaluated in this work. Section 6 presents the results of the experimentation. Finally, Section 7 summarizes the concluding remarks and presents some future lines of research.

2. Background

This section introduces the task of malware classification, it presents an overview of malware, the Portable Executable (PE) file format and the methods employed by malware authors to evolve malware.

2.1. The Task of Malware Detection and Classification

Malware detection refers to the task of identifying whether or not a given file is malicious to a computer system. By malicious we refer to code that is harmful to the system. Malware might seek to invade, damage or disable partially or completely the computer system, often taking control of it. Depending on their purpose, malware can be divided into various, not mutually exclusive

3

general categories including, but not limited to adware, spyware, trojans, rootkit, virus, worms, ransomware, etc. These categories provide a general overview of the functionality and behavior of malware. Typically, malware is also identified by a family name. A malware family refers to a collection of malware that has been generated from the same code base. Furthermore, malware families are divided into variants or strains, that is, malware built from an existing code base that have different signatures that are not included in the list of signatures used by anti-malware solutions. Distinguishing and classifying different types of malware is known as the task of malware classification and it provides information to better understand how the malware has infected the computers or devices, their threat level and how to protect against them.

2.2. The Portable Executable File Format

Malicious software targeting the Windows operating system is commonly written using the Portable Executable (PE) format, a file format for executable, object code, DLLs and others used in 32-bit and 64-bit versions of the Windows operations system. Portable Executables contain the information necessary for the Windows operating system to run the executable code including dynamic library references for linking, API export and import tables, etc. A Portable Executable file consists of headers and sections that tell the dynamic linker how to map the file into memory. An overview of the PE file is shown in Figure 1. The PE Header includes information regarding to the number of sections, their sizes, characteristics of the file, the import address table (IAT), etc. Furthermore, the PE file is divided into sections that contain the code

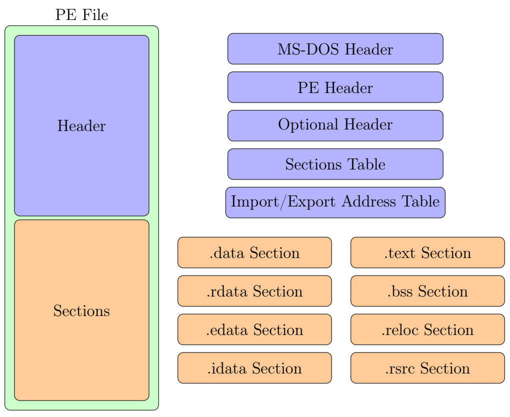


Figure 1: Portable Executable File Format

and data of the executable, including, but not limited to:

- the .text section. This section keeps the actual code of the computer program although the code can be written in any other section.
- the .data section. This section is used to declare initialized data or constants that do not change at runtime.
- the .rsrc section. This section contains all the resources of the program.

Detailed information of the PE file format can be found in the documentation provided by Microsoft[5].

2.3. Malware Evolution

Malware is constantly evolving and seeking new ways to bypass detection engines. The proliferation of malware has increased mainly due to the use of polymorphic and metamorphic techniques employed by malware authors to evade detection and hide

[5]`https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`

the true behavior of the executables. In polymorphic malware, a polymorphic engine is used to mutate the code while keeping the original functionality intact. The two most common methods to hide code are packing and encryption. On the one hand, packers employ one or more layers of compression to hide the real code of the program. Then, the original code is restored into memory at runtime through the unpacking routines and it is executed. On the other hand, crypters encrypt malware or part of its code to make it harder to analyze. A crypter typically contains a stub used to encrypt and decrypt malicious code. On the contrary, metamorphic malware rewrites its code to an equivalent version each time it is propagated. Malware authors may employ various transformation techniques including, but not limited to, register renaming, subroutine reordering and garbage code insertion. Thanks to the combination of the aforementioned techniques, malware volumes rapidly grown, making forensic investigations of malware cases time-consuming, costly and difficult even for security analysts and experts.

The aforementioned circumstances force security analysts and researchers to continually improve their cyberdefenses to keep pace with the evolution of malware. This caused the following problems with traditional antivirus solutions that relied on signature-based and heuristic/behavioral methods. First, signatures cannot be used to detect unknown malware variants because a new signature has to be developed previously. Second, although behavior-based detection is an effective approach to analyze the file's characteristics and behavior to determine if the file is indeed malware, the scanning and analysis is very time-consuming and can't be applied to every suspicious sample. Thus, researchers started adopting machine learning to complement their solutions and overcome the prior pitfalls of traditional signature-based engines and to provide an initial screening of the samples that exhibit malicious traits, as machine learning is well suited for processing large volumes of data.

3. Related Work

Machine learning has become an appealing tool for anti-malware vendors for either primary detection engines or as complementary detection heuristics. This is due to the ability of machine learning models to generalize to new samples, if the models are properly regularized. Furthermore, machine learning models allow to automatically summarize complex relationships among features that are discriminative between malware and goodware or between malware families, depending on the task, which allows the detection engine to adapt to the modifications in the malicious samples. For a complete review of machine learning solutions to detect and classify malware the reader is referred to the following articles (Souri and Hosseini, 2018; Ucci et al., 2019; Gibert et al., 2020b).

Although over the past decade there has been an increase in the research and deployment of machine learning solutions to tackle the problem of malware detection and classification, there is little research about the vulnerability of these models to adversarial attacks (Pitropakis et al., 2019; Mcgraw et al., 2019). Most state-of-the-art approaches (Demetrio et al., 2019; Kolosnjaji et al., 2018; Suciu et al., 2019) investigated how to slightly perturbate Portable Executable files by appending carefully-selected bytes at the end of the PE header

5

or at the end of the file, to evade detection by a shallow CNN architecture based on the raw bytes of the executable (Raff et al., 2018b). For instance, Demetrio et al. (2019) perturbed the bytes in the PE header that are expected to maximally increase the probability of evasion. On the other hand, Kolosnjaji et al. (2018) appended carefully-selected bytes at the end of the file, where these bytes were selected in a way that the resulting file minimizes the confidence associated to the malicious class. Unfortunately, for both approaches to work they need to have access to the machine learning model's gradients, which is very unlikely, if not impossible, to happen in a real-world scenario. Furthermore, some approaches in the literature tried to automatically learn which changes to perform to a feature vector (Hu and Tan, 2017) or to the actual executable (Anderson et al., 2018) in order to bypass black-box detectors. More specifically, Hu and Tan (2017) proposed a GAN to generate adversarial examples by modifying a binary feature vector, whose features refer to the API functions added to the import address table of the PE header of an executable. For example, if M APIs are used as features, an M-dimensional feature vector is constructed, with all the features corresponding to the imported API functions set to 1, and the rest set to 0. Contrarily, Anderson et al. (2018) proposed a reinforcement learning agent equipped with a set of functionality preserving operations like adding a function to the import address, manipulate the names of the sections, append bytes at the end of the file or between sections, etc. However, these modifications only affect the structure of the Portable Executables. None of them modify the actual source code of the executables. Thus, they greatly differ from the modifications performed by real-world malware to generate new variants of themselves.

The address the limitations of the aforementioned adversarial attacks, this paper performs an extensive investigation of the vulnerability of state-of-the-art machine learning approaches to realistic attacks already being employed by malware authors to bypass detection. For a complete description of the attacks see Section 4.

4. Metamorphic Attacks

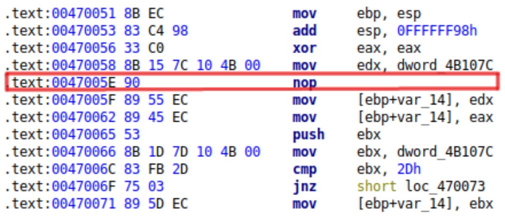
Malware authors usually employ metamorphic and polymorphic techniques to change the form of each instance from generation to generation in order to evade signature-based and pattern-matching detection. On the one hand, polymorphic malware pairs with a polymorphic engine with self-propagating code to continually change its appearance by using encryption or packaging algorithms to hide its code. On the other hand, metamorphic malware rewrites its code so that the newly propagated version of itself no longer matches its previous iterations. Metamorphic malware may use multiple transformation techniques that include, but are not limited to, garbage code or dead code insertion, register reassignment, subroutine reordering and code reordering through jumps. In this work we focus on the transformations performed by metamorphic techniques because they are the ones that make alterations to the actual source code of malware. Following, the most common metamorphic techniques are described in more detail.

4.1. Dead Code Insertion

The insertion of dead code or do-nothing instructions change the appearance of a program while not affecting the execution of the original code. Examples of malware that added dead code instructions on each generation are Evol and MetaPHOR. Cf. Figure 2. Following are described the dead code instructions implemented for our research purposes.

- **NOP**. The NOP or no-op instruction (short for **no operation**) is an assembly language instruction that does nothing.
- **MOV Reg, Reg**. The MOV instruction copies the contents of the register referred by its second operand into the register referred to by its firs operand.
- **PUSH Reg; POP Reg**. The PUSH instruction places the register referred to by its operand onto the top of the stack in memory while the POP instruction removes the element from the top of the stack.
- **ADD Reg, 0**. The ADD instruction adds the first and second operands, storing the result in its first operand.
- **SUB Reg, 0**. The SUB instruction subtracts the second operand from the first operand and stores the result in its first operand.
- **INC Reg; DEC Reg**. The INC instruction increments the content of the register referred by its operand by one while the DEC instruction decrements the contents of the register by one.
- **SHL Reg, 0**. The SHL instruction shifts the bits in its first operand's contents left, padding the resulting empty bit positions with zero. The number of bits to shift is specified by the second operand.
- **SHR Reg, 0**. The SHR instruction shifts the bits in its first operand's contents right, padding the resulting empty bit positions with zero. The number of bits to shift is specified by the second operand.

Notice that the dead code instructions **PUSH Reg; POP Reg** and **INC Reg; DEC Reg** do not necessarily need to be executed sequentially (one after the other). The instructions can be alternated with other instructions from the original code that do not modify the same registers as the do-nothing instructions.



```
.text:00470051 8B EC                 mov     ebp, esp
.text:00470053 83 C4 98              add     esp, 0FFFFFF98h
.text:00470056 33 C0                 xor     eax, eax
.text:00470058 8B 15 7C 10 4B 00     mov     edx, dword_4B107C
.text:0047005E 90                    nop
.text:0047005F 89 55 EC              mov     [ebp+var_14], edx
.text:00470062 89 45 EC              mov     [ebp+var_14], eax
.text:00470065 53                    push    ebx
.text:00470066 8B 1D 7D 10 4B 00     mov     ebx, dword_4B107C
.text:0047006C 83 FB 2D              cmp     ebx, 2Dh
.text:0047006F 75 03                 jnz     short loc_470073
.text:00470071 89 5D EC              mov     [ebp+var_14], ebx
```

Figure 2: Assembly language source code after the insertion of a NOP instruction.

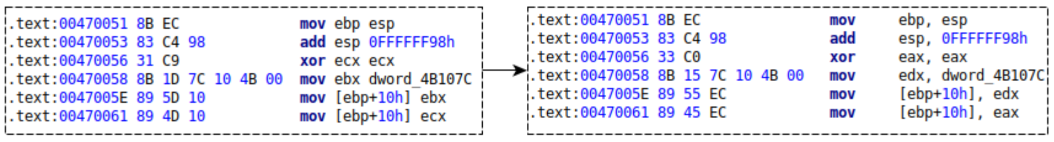
4.2. Register's Reassignment

Register reassignment switches registers from generation to generation while keeping the program behavior unaltered. Cf. Figure 3. This technique was first used by Win95/RegSwap virus. Traditional antivirus engines detect viruses that employ this technique by a wildcard string search (Ször and Ferrie, 2001). Wildcard strings allow to skip particular bytes in regular expressions. For instance, in "89 ?? 7C 10 4B 00", the wildcard is indicated by '??'.

4.3. Subroutine Reordering

The subroutine reordering technique employs permutations to reorder the subroutines of a malware executable. Cf. Figure 4. With n different subroutines, this

7

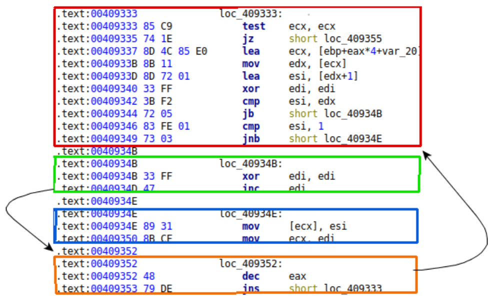


```
.text:00470051 8B EC                mov ebp esp
.text:00470053 83 C4 98             add esp 0FFFFFF98h
.text:00470056 31 C9                xor ecx ecx
.text:00470058 8B 1D 7C 10 4B 00    mov ebx dword_4B107C
.text:0047005E 89 5D 10             mov [ebp+10h] ebx
.text:00470061 89 4D 10             mov [ebp+10h] ecx
```

```
.text:00470051 8B EC                mov     ebp, esp
.text:00470053 83 C4 98             add     esp, 0FFFFFF98h
.text:00470056 33 C0                xor     eax, eax
.text:00470058 8B 15 7C 10 4B 00    mov     edx, dword_4B107C
.text:0047005E 89 55 EC             mov     [ebp+10h], edx
.text:00470061 89 45 EC             mov     [ebp+10h], eax
```

Figure 3: Register reassignment example. Registers ecx and ebx are switched to eax and edx, respectively.

technique can generate up to $n!$ different variants. The technique was initially employed by Win32/Ghost virus, which had ten subroutines. Thus, it could generate $10! = 3,628,800$ variants.



```
.text:00409333                  loc_409333:
.text:00409333 85 C9                    test    ecx, ecx
.text:00409335 74 1E                    jz      short loc_409355
.text:00409337 8D 4C 85 E0              lea     ecx, [ebp+eax*4+var_20]
.text:0040933B 8B 11                    mov     edx, [ecx]
.text:0040933D 8D 72 01                 lea     esi, [edx+1]
.text:00409340 33 FF                    xor     edi, edi
.text:00409342 3B F2                    cmp     esi, edx
.text:00409344 72 05                    jb      short loc_40934B
.text:00409346 83 FE 01                 cmp     esi, 1
.text:00409349 73 03                    jnb     short loc_40934E
.text:0040934B
.text:0040934B                  loc_40934B:
.text:0040934B 33 FF                    xor     edi, edi
.text:0040934D 47                       inc     edi
.text:0040934E
.text:0040934E                  loc_40934E:
.text:0040934E 89 31                    mov     [ecx], esi
.text:00409350 8B CF                    mov     ecx, edi
.text:00409352
.text:00409352                  loc_409352:
.text:00409352 48                       dec     eax
.text:00409353 79 DE                    jns     short loc_409333
```

Figure 4: Subroutine reordering technique example.

4.4. *Code Reordering through Jumps*

Code reordering through jumps is based on the insertion of conditional or unconditional jumps to split a subroutine into two blocks of instructions. Afterwards, these blocks generated by the branching instruction are permuted to change the control flow. Cf. Figure 5. This technique was first employed by Win95/ZPerm family to generate new variants jointly with the insertion of garbage or dead code instructions.

5. Static Machine Learning Anti-Malware Tools

There are multiple ways in which malware can be represented from a static analysis point of view. Typically, features are manually-engineered to capture some specific characteristics of the executable that can help distinguishing malware families or malware from benign software, e.g. API function calls, byte and opcode n-grams, etc (Souri and Hosseini, 2018; Ucci et al., 2019; Gibert et al., 2020b). In the present study, the machine learning approaches are limited to those whose performance would be affected by the functionality-preserving changes performed by the metamorphic attacks described in Section 4. Accordingly, the approaches assessed in this study can be divided in two groups: (1) n-gram based approaches or (2) deep learning approaches, depending on whether they take as input a feature vector containing an abstract representation of the executable or directly raw data.

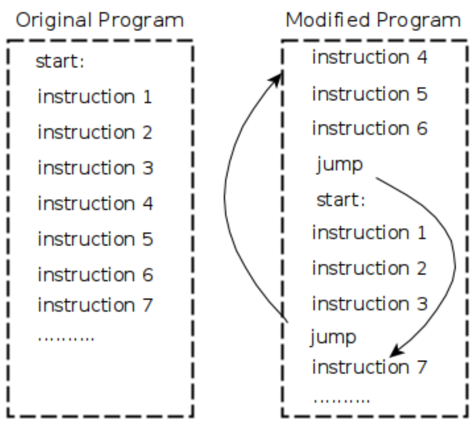


Figure 5: Code reordering through jumps example.

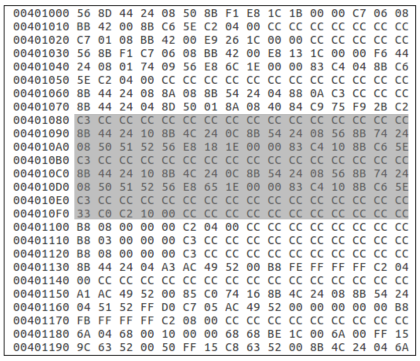
5.1. N-gram based Approaches

An n-gram is a contiguous sequence of n items from a given sequence of text. N-grams can be extracted from the hexadecimal representation of malware's binary content and from the assembly language source code. On the one hand, the hexadecimal representation represents the binary content of an executable as a sequence of bytes (base-16 number representation with digits [0-9] and [A-F]). Cf. Figure 6a. Alternatively, the assembly language source code contains the symbolic machine code of an executable with metadata information as function calls, memory allocation and variable information. Cf. Figure 6b. In consequence, byte n-grams (Zhang and Zhao, 2017; Raff and Nicholas, 2018) and opcode n-grams (Santos et al., 2013; Hu et al., 2013) refer to the unique combination of every n consecutive bytes and opcodes as individual features, respectively. An opcode refers to the name of a specific instruction, i.e. *ADD*, *MUL*, *PUSH*, etc, without its arguments.

N-gram based approaches construct a feature vector containing an abstract representation of malware, where each element in the vector indicates the number of appearances of a particular n-gram in the sequence of text. In consequence, the length of the feature vector depends on the number of unique n-grams, which increases with n. For instance, if we want to extract byte n-grams with $n = 3$, the number of possible n-grams is $256^3 = 16.777.216$. This leads to two main problems. First, the resulting feature vector is too large to keep in memory, even if malware n-grams do not increase exponentially with n but follow a Zipfian distribution (Raff et al., 2018c). Second, the machine learning model will be affected by the curse of dimensionality (Bellman, 2015; Chen, 2009) which means that the number of samples in the dataset that need to be accessed to estimate a function with a given level of accuracy grows exponentially with the underlying dimensionality. As a result, methods in the literature reduced the high dimensional input space using feature selection techniques (Santos et al., 2013; Zhang and Zhao, 2017) or the hashing trick (Raff and Nicholas, 2018; Hu et al., 2013).

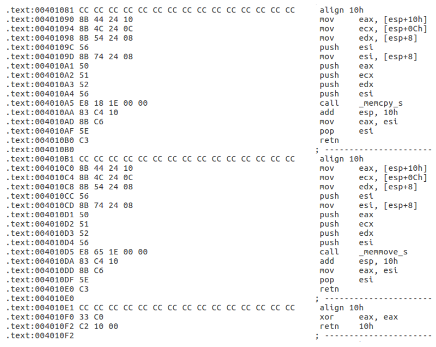
1. Feature selection is the process of selecting a subset of relevant features from the initial input space for use in model construction. A common approach is to rank the features based on the mutual information index in decreasing order (Santos et al., 2013; Zhang and Zhao, 2017). Mutual information, is an index of statistical dependence between two variables (Vergara and Estévez, 2014). In the case of a classification task, it measures the dependence between a feature X and the target variable Y. This is done by measuring how much knowing one of these variables reduces uncertainty about the other. The mutual information between two variables is a non-negative value, which measures the dependency between the variables. For two independent variables, their mutual information will be 0. Otherwise, for dependent variables, higher mutual information mean higher dependency.
2. Feature hashing, also known as the hashing trick, is a method for handling sparse, high-dimensional feature vectors by using a hash function to determine the feature's location in a lower-dimensional vector. It can be seen as a random projection of the input space $A \in \mathbb{R}^n$ to a low dimensional space $B \in \mathbb{R}^m$, where $m \ll n$. More specif-

9



```
00401000 56 8D 44 24 08 50 8B F1 E8 1C 1B 00 00 C7 06 08
00401010 BB 42 00 8B C6 5E C2 04 00 CC CC CC CC CC CC CC
00401020 C7 01 08 BB 42 00 E9 26 1C 00 00 CC CC CC CC CC
00401030 56 8B F1 C7 06 08 BB 42 00 E8 13 1C 00 00 F6 44
00401040 24 08 01 74 09 56 E8 6C 1E 00 00 83 C4 04 8B C6
00401050 5E C2 04 00 CC CC CC CC CC CC CC CC CC CC CC CC
00401060 8B 44 24 08 8A 08 8B 54 24 04 88 0A C3 CC CC CC
00401070 8B 44 24 04 8D 50 01 8A 08 40 84 C9 75 F9 2B C2
00401080 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401090 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010A0 08 50 51 52 56 E8 18 1E 00 00 83 C4 10 8B C6 5E
004010B0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010C0 8B 44 24 10 8B 4C 24 0C 8B 54 24 08 56 8B 74 24
004010D0 08 50 51 52 56 E8 65 1E 00 00 83 C4 10 8B C6 5E
004010E0 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
004010F0 33 C0 C2 10 00 CC CC CC CC CC CC CC CC CC CC CC
00401100 B8 08 00 00 00 C2 04 00 CC CC CC CC CC CC CC CC
00401110 B8 03 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401120 B8 08 00 00 00 C3 CC CC CC CC CC CC CC CC CC CC
00401130 8B 44 24 04 A3 AC 49 52 00 B8 FE FF FF FF C2 04
00401140 00 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
00401150 A1 AC 49 52 00 85 C0 74 16 8B 4C 24 08 8B 54 24
00401160 04 51 52 FF D0 C7 05 AC 49 52 00 00 00 00 00 B8
00401170 FB FF FF FF C2 08 00 CC CC CC CC CC CC CC CC CC
00401180 6A 04 68 00 10 00 00 68 68 BE 1C 00 6A 00 FF 15
00401190 9C 63 52 00 50 FF 15 C8 63 52 00 8B 4C 24 04 6A
```

(a) Hexadecimal view of a PE file. Each line is composed of the starting address of the machine codes in the memory and an accumulation of consecutive 16 byte values.



```
.text:00401081 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:00401090 8B 44 24 10                                      mov     eax, [esp+10h]
.text:00401094 8B 4C 24 0C                                      mov     ecx, [esp+0Ch]
.text:00401098 8B 54 24 08                                      mov     edx, [esp+8]
.text:0040109C 56                                               push    esi
.text:0040109D 8B 74 24 08                                      mov     esi, [esp+8]
.text:004010A1 50                                               push    eax
.text:004010A2 51                                               push    ecx
.text:004010A3 52                                               push    edx
.text:004010A4 56                                               push    esi
.text:004010A5 E8 18 1E 00 00                                   call    _memcpy_s
.text:004010AA 83 C4 10                                         add     esp, 10h
.text:004010AD 8B C6                                            mov     eax, esi
.text:004010AF 5E                                               pop     esi
.text:004010B0 C3                                               retn
.text:004010B0                                                 ; ---------------------
.text:004010B1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010C0 8B 44 24 10                                      mov     eax, [esp+10h]
.text:004010C4 8B 4C 24 0C                                      mov     ecx, [esp+0Ch]
.text:004010C8 8B 54 24 08                                      mov     edx, [esp+8]
.text:004010CC 56                                               push    esi
.text:004010CD 8B 74 24 08                                      mov     esi, [esp+8]
.text:004010D1 50                                               push    eax
.text:004010D2 51                                               push    ecx
.text:004010D3 52                                               push    edx
.text:004010D4 56                                               push    esi
.text:004010D5 E8 65 1E 00 00                                   call    _memmove_s
.text:004010DA 83 C4 10                                         add     esp, 10h
.text:004010DD 8B C6                                            mov     eax, esi
.text:004010DF 5E                                               pop     esi
.text:004010E0 C3                                               retn
.text:004010E0                                                 ; ---------------------
.text:004010E1 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC     align 10h
.text:004010F0 33 C0                                            xor     eax, eax
.text:004010F2 C2 10 00                                         retn    10h
.text:004010F2                                                 ; ---------------------
```

(b) Assembly view of the grayed part in Figure 6a. The first column represents the address, the second column the byte sequence and the third column the mnemonics sequence.

Figure 6: Hexadecimal and assembly view of a Portable Executable file.

ically, given an array of size N that counts the number of times each n-gram occurred, and a hash function, the hashing trick maps each n-gram to a location in the lower dimensional array.

Afterwards, the resulting low-dimensional feature vector is used for training a classification algorithm. In our experiments, we extracted 3-gram features from both the hexadecimal view and the assembly view. The high-dimensional feature vector was reduced using feature selection or the hashing trick. The size of the resulting low-dimensional vector is set to $K = 5000$ (different sizes for $K \in [500, 2000, 5000]$ were tried but $K = 5000$ provides the best performance). Afterwards, we trained various classifiers, including a logistic regression classifier and feed-forward neural networks consisting of 1 to 3 hidden layers. The number of neurons in the hidden layers was set to 2048, 1024 and 512 for the first, second and third hidden layer, respectively. The results presented in Section 6 show the performance of the best classifiers, which are referred using the following notation for the rest of the paper:

- *NN opcodes (hashing trick)* refers to a neural network with two hidden layers trained with the opcode-based feature vector reduced with the hashing trick technique. Cf. Figure 7.

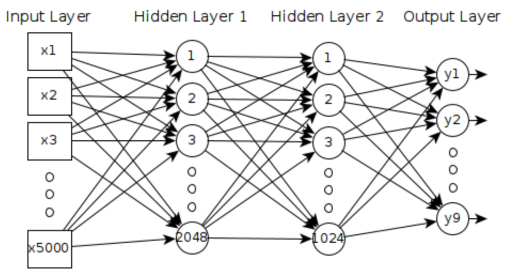


Figure 7: NN opcodes (hashing trick) architecture.

- *LR opcodes (mutual information)* refers to a logistic regression classifier trained with the top K opcode-based features selected using the mutual information index. Cf. Figure 8.

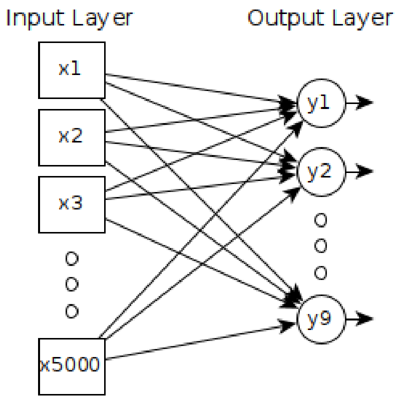


Figure 8: LR opcodes (mutual information) architecture.

- *NN bytes (hashing trick)* refers to a neural network with one hidden layer trained with the byte-based feature vector reduced with the hashing trick technique. Cf. Figure 9.

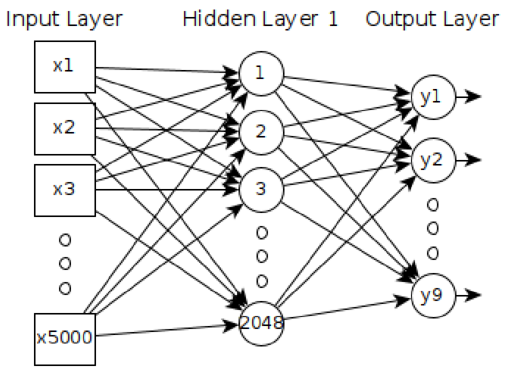


Figure 9: NN bytes (hashing trick) architecture.

Notice that feature selection using the mutual information metric has been only applied to the opcode-based 3-grams. This is because in the dataset used for training there are 55136 and 9514156 unique opcode and byte 3-grams, respectively, and in consequence, the memory requirements needed for selecting 5000 byte-based 3-gram features far exceeds the memory capacity of our system.

Instead of taking the number of appearances of n-grams as features, data normalization is applied to normalize the range of the features. The motivation behind data normalization is that the range of values of each feature varies widely. Machine learning models learn a mapping from input variables to an output variable. In consequence, the scale and distribution of the data drawn from the domain may be different for each variable or feature. For example, the number of times the 3-gram [mov, push, mov] or the 3-gram [pop, sub, and] may greatly differ. These differences in the scales across input variables may increase the difficulty of the problem being modeled. For instance, large input values can result in a model that learns large weight values, which are known to be often unstable, i.e. it may suffer from poor performance during learning and sensitivity to input values resulting in higher generalization error. Therefore, the features have been standarized so that they have zero-mean and unit-variance. The formula for standarization is as follows:

$$x' = \frac{x - \mu}{\delta}$$

where x is the original feature vector, μ is the mean of the feature values and δ is the standard deviation of the feature values.

11

5.2. Deep Learning Approaches

The need for manual feature engineering can be obviated by automated feature learning. Deep learning replaces the feature engineering process by an underlying system which typically consists of a neural network with multiple layers, that performs both feature learning and classification. With deep learning, one can start with raw data as features will be automatically created by the neural network when it learns. The main distinction between deep learning approaches for malware detection and classification lean on what they use as raw data.

5.2.1. Opcode-based Approaches

Opcode-based approaches (Gibert et al., 2017) take as input a sequence of assembly language instructions extracted from the assembly language source code of an executable. Gibert et al. (2017) proposed a shallow convolutional neural network to extract n-gram like features from malware's instructions. This is achieved by a convolutional layer with filters of various sizes. In their work, an assembly program is represented as a concatenation of mnemonics

$$x_{1:n} = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

where n is the length of the program and $x_i \in \mathbb{R}^k$ corresponds to the i-th mnemonic in the program. Instead of representing the mnemonics as one-hot vectors, each mnemonic is represented as a word embedding. Following, a convolutional layer extracts n-gram like features. This is achieved by the convolution operator, which involves a filter $w \in \mathbb{R}^{hk}$ where h is the number of mnemonics to which is applied and k is the size of the word embedding. In particular, filters are applied to sequences containing from 3, 5 and 7 mnemonics. Cf. Figure A.18

5.2.2. Byte-based Approaches

Byte-based approaches (Raff et al., 2018a; Krčál et al., 2018; Gibert et al., 2018) are those that take as input a sequence of bytes extracted from the hexadecimal representation of the malware's binary content. Cf. Figure 6a. These approaches face the following challenges:

- By treating an executable as a sequence of bytes, we are dealing with sequences of millions of time steps, which turns the task of malware detection and classification as one of the most challenging sequence classification problems with regard to the size of the time series (sequence of bytes).
- The meaning of any byte is dependent on its context and could encode any type of information, from binary code to human-readable text, images, etc.
- The same instruction could be encoded using different bytes depending on its arguments. For instance, the bytes sequence corresponding to the cmp instruction can begin with 0x3C, 0x3D, 0x3A, 0x3B, 0x80, 0x81, 0x38 or 0x39 depending on the arguments given.
- The content of a Portable Executable (PE) file exhibits various levels of spatial correlation. Nearby instructions within the same funtion are spatially correlated, but function calls and jcc instructions produce discontinuities over the code instructions and functions. As a result, this discontinuities are maintained through the bytes sequences.

Following it is provided a brief description of the architectures evaluated in the present study:

Raff et al. (2018a) proposed an architecture that consists of an embedding layer, a gated convolutional layer, a global max-pooling layer to produce its activations regardless of the location of the detected features, followed by fully-connected layers. This architecture will be called *MalConv* from now on. Cf. Figure A.19.

Krčál et al. (2018) presented a deep convolutional neural network architecture that consists of an embedding layer, four convolutions with strides and max-pooling between the second and third convolutions. Afterwards, it follows a global average pooling layer and various fully-connected layers. This architecture will be called *DeepConv* from now on. Cf. Figure A.20.

Gibert et al. (2018) presented a convolutional neural network architecture to categorize malware based on their structural entropy. The structural entropy of an executable is the representation of a file as a stream of entropy values, where each value describes the amount of entropy over a small chunk of code in a specific location of the file. Additionally, they proposed a multiresolution CNN to classify malware based on the approximation and details coefficients generated by the Haar wavelet transform over the entropy time series. The architectures will be called *Structural entropy CNN* and *Multiresolution CNN* from now on. Cf. Figures A.21, A.22.

5.2.3. Byte-based Shallow Convolutional Neural Network

Byte-based approaches presented in the literature (Raff et al., 2018a; Krčál et al., 2018; Gibert et al., 2018) tend to underperform in comparison to the opcode-based approaches (Gibert et al., 2017) as it can be observed in Section 6.3. Our intuition is that the size of their filters and the complexity of the network architectures played a deep role. To check this premise, a shallow convolutional neural network architecture similar to the one presented by Gibert et al. (2017) has been proposed based on the raw byte sequences.

The architecture differs in the input of the network and in the size of the convolutional filters. Instead of receiving the assembly language instructions, the network takes as input the hexadecimal representation of the malware's binary content represented as a concatenation of bytes:

$$x_{1:n} = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

where n is the length of the program and $x_i \in \mathbb{R}^k$ corresponds to the i-th byte in the program. The overall architecture is presented in Figure 10.

The network comprises the following layers:

- **Input layer.** The network takes as input a sequence of bytes, of size N, representing malware's binary content.

- **Embedding layer.** Rather than perform convolutions on the raw byte values, each byte is mapped to a fixed length feature vector (word embedding) that is learnt during training. Take into account that using raw byte values would imply that certain byte values are intrinsically closer to each other than other byte values, which is known a priori to be false, as the meaning of the byte value is dependent on the context.

- **Convolutional layer.** This layer convolves various filters over the byte sequences and extracts n-gram like features from it. A convolution operation

13

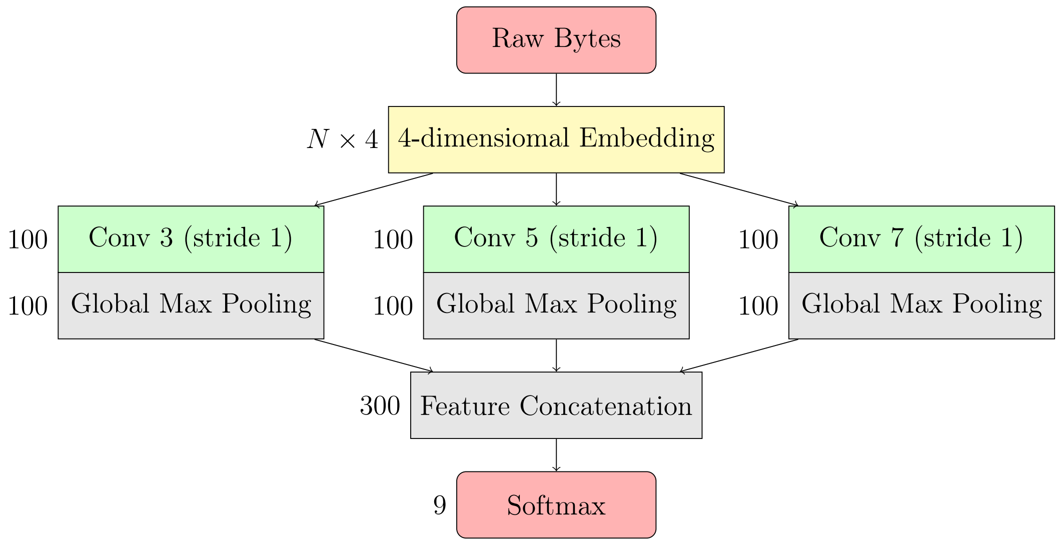


Figure 10: Convolutional neural network for malware classification from sequences of bytes.

involves a filter $w \in \mathbb{R}^{hk}$ where h is the number of bytes to which is applied and k is the size of the word embedding. In particular, filters are applied to sequences containing 3,5 and 7 bytes.

A feature c_i is generated from a window of bytes $x_{i:i+h-1}$ (it comprises all bytes between position i and $i+h-1$) and is defined as follows:

$$c_i = f(w \cdot x_{i:i+h-1} + b),$$

where f is a rectifier linear unit (ReLU) function and b the bias term.

- **Pooling layer.** Global max-pooling is applied to extract the maximum activation of each of the feature map activations generated by the convolutional layer.

- **Softmax layer.** It linearly combines the features learned by the previous layers and applies the softmax function to generate a vector containing the normalized probability distribution over malware families.

Other variants of this architecture including dilated convolutions (Yu and Koltun, 2016) and gated linear units (Dauphin et al., 2017) have been evaluated but as it can be observed in Section 6.3, their performance is slightly worse than the standard convolution. This variants are named *Aatrous CNN* and *CNN GLU*, respectively.

6. Evaluation

This section presents an extensive evaluation of the robustness of state-of-the-art detectors powered by ML against the metamorphic techniques presented in Section 4. The experiments are carried out using the data provided by Microsoft for the Big Data Innovators Gathering Challenge of 2015 (Ronen et al., 2018). Furthermore, we investigate the utility of the aforementioned metamorphic techniques to augment the dataset and reduce class imbalance.

6.1. The Microsoft Malware Classification Challenge

Unlike other applications, the task of malware detection and classification has not received much attention in the research community and unfortunately, there are not available rich labeled datasets. Due to legal restrictions, benign binaries (e.g. executables of common Windows applications, utilities, etc) can not be shared, as they are often protected by copyright laws. Contrarily, there are websites such as VirusShare and VXHeaven that share malicious executables. However, unlike other domains where data may be labeled very rapidly by a non-expert, determining whether a file is malicious and its corresponding family or class is a very time-consuming process, even for security experts. In consequence, for reproducibility purposes the research conducted in this paper has been evaluated on the data provided by Microsoft for the Big Data Innovators Gathering Challenge of 2015 (Ronen et al., 2018), which over the years has become the de facto benchmark for evaluating approaches on the task of malware classification. Microsoft provided a high-quality public labeled benchmark of almost half a terabyte of malware. Nowadays, the dataset is hosted on Kaggle [6] and is publicly accessible. The dataset contains samples of malware representing a mix of 9 malware families (See Table 1): (1) Ramnit, (2) Lollipop, (3) Kelihos_ver3, (4) Vundo, (5) Simda, (6) Tracur, (7) Kelihos_ver1, (8) Obfuscator.ACY and (9) Gatak.

[6]https://www.kaggle.com/c/malware-classification/

Table 1: Class distribution in the Microsoft Malware Classification Challenge dataset.

Family Name	#Samples	Type
Ramnit	1541	Worm
Lollipop	2478	Adware
Kelihos_ver3	2942	Backdoor
Vundo	475	Trojan
Simda	42	Backdoor
Tracur	751	TrojanDownloader
Kelihos_ver1	398	Backdoor
Obfuscator.ACY	1228	Any kind of obfuscated malware
Gatak	1013	Backdoor

6.2. Experimental Setup

The experiments have been carried out on a machine with an Intel Core i7-7700K CPU, 4xGeforce GTX 1080Ti and 64Gb of RAM. All algorithms have been implemented using TensorFlow (Abadi et al., 2015).

The experimentation has been divided into two phases. The first phase analyses the individual impact of the metamorphic techniques on the performance of the machine learning models. Oppositely, the second phase analyzes the usage of the metamorphic techniques for augmenting the training set and boosting the performance of the machine learning classifiers.

6.3. Analysis of the Performance of ML Classifiers against Metamorphic Techniques

To analyze the performance of the ML classifiers, the dataset has been divided into three sets: (1) the training set, (2) the validation set and (3) the test set, containing 70%, 15% and 15% of the samples, respectively.

Instead of the accuracy, the macro F1-score has been used to evaluate the models. This is due to the fact that accuracy alone can be a misleading measure, specially in datasets with large class imbalance. For instance, a machine learning model might correctly predict the value of the majority class for all predictions and achieve a high classification accuracy while failing to correctly predict the class of samples belonging to the minority and critical classes. In this situation the macro F1-score metric is more adequate because it penalizes this kind of behavior by calculating the unweighted mean of the precision and recall for each label or class. The mathematical formulation of the macro f1-score is as follows:

$$macro\ f1\ score = \frac{1}{q}\sum_{i=1}^{q} F_1^i$$

where F_1^i is the weighted average of precision and recall in class i.

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}$$

When evaluating the performance of the classifiers, the test set is used to generate three obfuscated test sets (A, B and C) and the average macro f1-score achieved on the three sets is provided as outcome.

6.3.1. Dead Code Insertion

To evaluate the resilience of ML models to the changes performed on the code by the dead code insertion technique, the test set has been obfuscated by inserting 10, 50, 100, 200 and 500 dead code instructions in random positions within the assembly language source code of executables (See Section 4.1). This includes all sections that contain assembly language instructions, and not only the .text section. Given P = [0.5, 0.071, ..., 0.071] and X = [NOP, $MOV\ Reg\ Reg$, ...], where $|P|$ and $|X|$ equals N, $P(n)$ is the probability to select the $X(n)$ element. Accordingly, the probability to insert the NOP instruction is 0.5 while the probability to insert any other instruction is 0.071. The probability values have been set as described above to put more focus on the NOP instruction.

Table 2: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated by the dead code insertion technique.

			Dead code insertion				
	Method	Test Set	10	50	100	200	500
Opcode-based	(Gibert et al., 2017)	0.9852	0.9806	0.9794	0.9711	0.9672	0.8754
	NN opcodes (hashing trick)	0.9765	0.9754	0.9738	0.9704	0.9458	0.7031
	NN opcodes (mutual information)	0.9876	0.9873	0.9888	0.9887	0.9879	0.9728
Byte-based	MalConv (Raff et al., 2018a)	0.8480	0.8543	0.8502	0.8497	0.8300	0.8328
	DeepConv (Krčál et al., 2018)	0.8376	0.8030	0.7913	0.7567	0.6944	0.6765
	Structural entropy CNN (Gibert et al., 2018)	0.8809	0.8952	0.9015	0.8880	0.8348	0.7818
	Multiresolution CNN (Gibert et al., 2018)	0.8972	0.9074	0.9008	0.8880	0.8297	0.7995
	NN bytes (hashing trick)	0.8858	0.8863	0.8854	0.8864	0.8863	0.8861
	Shallow CNN	0.9748	0.9733	0.9731	0.9694	0.9674	0.9577
	Dilated CNN	0.9622	0.9369	0.9362	0.9427	0.9317	0.9280
	CNN GLU	0.9627	0.9627	0.9627	0.9623	0.9606	0.9638

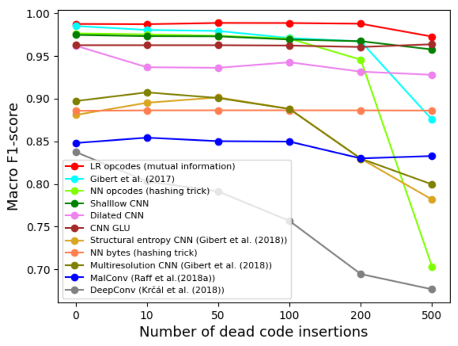


Figure 11: Macro f1-score of ML classifiers on the test set obfuscated by the dead code insertion technique.

Table 2 and Figure 11 present the classification performance of the ML classifiers against the aforementioned obfuscated test set. It can be observed that the performance of the opcode-based methods (Gibert et al. (2017), NN opcodes (hashing trick)) degrade considerably when more than 200 dead code instructions are inserted per sample. On the contrary, byte-based classifiers (Raff et al. (2018a); Krčál et al. (2018); Gibert et al. (2018), NN bytes (hashing trick)) remain more stable to the changes in the executable. There are two reasons for this occurrence: (1) First, the size of the executables significantly differ between families (Gibert et al., 2020a). The less opcodes has the sample the easy is to obfuscate the patterns learned by the ML model; (2) Second, samples belonging to some families in the dataset do not contain any NOP instruction in their assembly language source code such as samples belonging to the Kelihos_ver1 family. This have caused the opcode-based models to learn that if there exist a n-gram containing the NOP instruction in the assembly language source code, the corresponding executable might belong to any other family but not to Kelihos_ver1. For instance, as it can be observed in Figure 12 the opcode-based CNN (Gibert et al., 2017) only classified correctly 27 out of 67 samples of the Kelihos_ver1 families. To check this premise, in Section 6.5 it is augmented the training data by using a combination of the metamorphic techniques presented in Section 4, including the dead code insertion technique. Results show that the degradation in the performance of opcode-based classifiers is due to data bias rather than to any weaknesses of the opcode-based classifiers with respect to the byte-based classifiers and could be resolved by augmenting the training set with some samples of those families containing some NOP instructions.

6.3.2. Register's Reassignment

The second metamorphic technique evaluated is the register's reassignment technique (See Section 4.2). To evaluate the resilience of ML models against the register's reassignment technique, two or more data registers (i.e. *EAX*, *EBX*, *ECX*, *EDX*) of the samples in the test set have been swapped. More specifically, two experiments have been performed:

- Experiment A: Swap two randomly selected data registers.
- Experiment B: Swap all data registers.

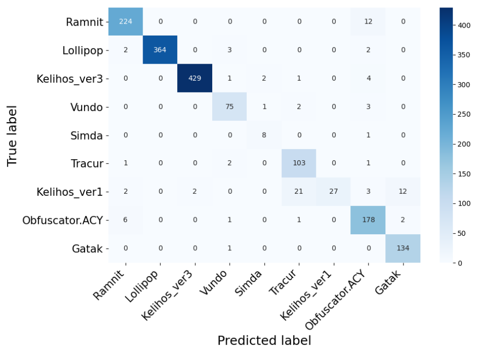


Figure 12: Opcode-based CNN confusion matrix (500 dead code insertions).

Table 3: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated by the register's reassignment technique.

		Register's Reassignment	
Method	Test Set	A	B
MalConv (Raff et al., 2018a)	0.8480	0.8465	0.8294
DeepConv citepkrcal2018deep	0.8376	0.8136	0.7768
Structural entropy CNN (Gibert et al., 2018)	0.8809	0.8850	0.8954
Multiresolution CNN (Gibert et al., 2018)	0.8972	0.8973	0.8996
NN bytes (hashing trick)	0.8858	0.8846	0.8764
Shallow CNN	0.9748	0.9708	0.9565
Dilated CNN	0.9622	0.9410	0.9308
CNN GLU	0.9627	0.9638	0.9570

Table 3 and Figure 13 present the performance of the ML classifiers against the samples of the test set obfuscated with the register's reassignment technique. Notice that opcode-based approaches are not evaluated as they are not affected by this technique. On the contrary, Malconv's (Raff et al., 2018a), DeepConv's (Krčál et al., 2018), the shallow CNN and the n-gram based classifier's performance degraded 2.19%, 7.26%, 1.88% and 1.06% respectively while the

17

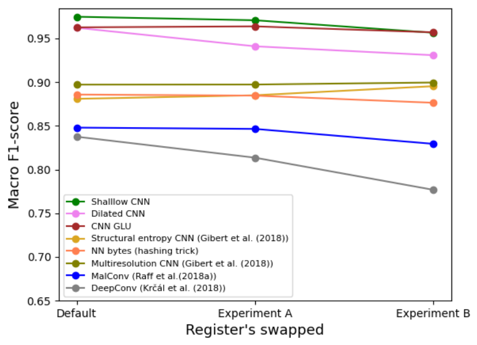


Figure 13: Macro f1-score of ML classifiers on the test set obfuscated by the register's reassignment technique.

macro f1-score of the methods presented in Gibert et al. (2018) slightly increased. Our intuition is that both classifiers (Gibert et al., 2018) as they are based on the structural entropy of an executable, even that the register reassignment technique replaces some bytes in the executable, the entropy time series remain mostly unaltered. In addition, it can be observed that the performance of the classifiers decreases as the number of registers swapped augments.

6.3.3. Subroutine Reordering

To evaluate the resilience of ML models to the modifications performed on the code by the subroutine reordering technique, the samples on the test set have been obfuscated by performing 5, 10, 20 and 50 random subroutine permutations (See Section 4.3).

Table 4 and Figure 14 display the performance of the ML models over the obfuscated test set. It can be observed that the classification performance of all models remain mostly constant. This is because most neural network architectures evaluated contain a global max-pooling (Raff et al., 2018a; Gibert et al., 2017) or global

Table 4: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated by the subroutine reordering technique.

	Method	Test Set	Subroutine reorderings			
			5	10	20	50
Opcode-based	(Gibert et al., 2017)	0.9852	0.9807	0.9801	0.9798	0.9784
	NN opcodes (hashing trick)	0.9765	0.9784	0.9781	0.9785	0.9783
	NN opcodes (mutual information)	0.9876	0.9876	0.9876	0.9874	0.9874
Byte-based	MalConv (Raff et al., 2018a)	0.8480	0.8558	0.8391	0.8645	0.8552
	DeepConv (Krčál et al., 2018)	0.8376	0.8025	0.8150	0.7955	0.7903
	Structural entropy CNN (Gibert et al., 2018)	0.8809	0.8976	0.8929	0.9026	0.8990
	Multiresolution CNN (Gibert et al., 2018)	0.8972	0.9044	0.8998	0.9013	0.9008
	NN bytes (hashing trick)	0.8858	0.8859	0.8860	0.8852	0.8851
	Shallow CNN	0.9748	0.9732	0.9733	0.9731	0.9734
	Dilated CNN	0.9622	0.9465	0.9418	0.9474	0.9560
	CNN GLU	0.9627	0.9623	0.9627	0.9626	0.9623

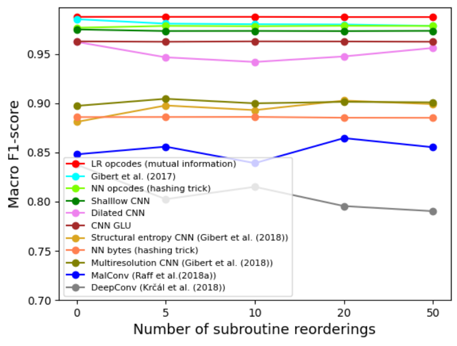


Figure 14: Macro f1-score of ML classifiers on the test set obfuscated by the subroutine reordering technique.

avg-pooling layer (Krčál et al., 2018) at the end of the convolutional layers which allowed the detection of patterns independently of their position in the raw input sequences.

6.3.4. Code Reordering through Jumps

The resilience of the ML models against the code reordering through jumps technique (See Section 4.4) is assessed by reordering the code with the insertion of 5, 10, 20 and 50 jumps randomly within the assembly language source code of the executables.

Table 5 and Figure 15 present the performance of the ML classifiers against the samples of the test set obfuscated with the code reordering through jumps technique.

Table 5: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated by the code reordering through jumps technique.

	Method	Test Set	Code reordering through jumps			
			5	10	20	50
Opcode-based	(Gibert et al., 2017)	0.9852	0.9789	0.9776	0.9768	0.9729
	NN opcodes (hashing trick)	0.9765	0.9784	0.9777	0.9765	0.9667
	NN opcodes (mutual information)	0.9876	0.9874	0.9876	0.9876	0.9876
Byte-based	MalConv (Raff et al., 2018a)	0.8480	0.8482	0.8573	0.8374	0.8481
	DeepConv (Krčál et al., 2018)	0.8376	0.7547	0.7243	0.7136	0.6749
	Structural entropy CNN (Gibert et al., 2018)	0.8809	0.8416	0.8162	0.7671	0.7319
	Multiresolution CNN (Gibert et al., 2018)	0.8972	0.8453	0.8195	0.7798	0.7457
	NN bytes (hashing trick)	0.8858	0.8841	0.8954	0.8907	0.8955
	Shallow CNN	0.9748	0.9732	0.9739	0.9733	0.9743
	Dilated CNN	0.9622	0.9503	0.9535	0.9605	0.9609
	CNN GLU	0.9627	0.9625	0.9628	0.9627	0.9607

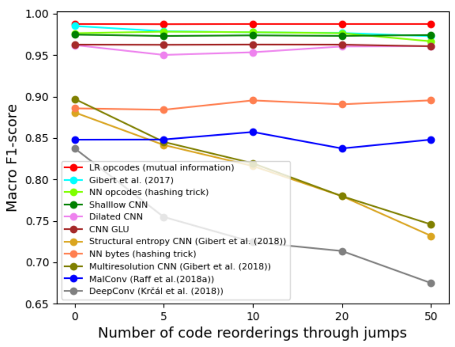


Figure 15: Macro f1-score of ML classifiers on the test set obfuscated by the code reordering through jumps technique.

Results are similar to Section 6.3.3 with the byte-based approaches performing poorly in comparison to opcode-based approaches, and additionally, the performance of DeepConv (Krčál et al., 2018), structural entropy and haar approximation & coefficients (Gibert et al., 2018) degraded considerably from 0.8376, 0.8809, 0.8972 to 0.6749, 0.7319, 0.7457, respectively. At the present moment we don't have an explanation for the behavior of the aforementioned models but it might be the case that the code reordering through jumps technique modifies the initial byte sequences in such a way that the resulting time series look completely different from the non-obfuscated versions. For instance, in Figure 16 it can be observed that the structural entropy visualization of the malicious sample with ID *bxED6RSpmnWV03kyMLoK* diverges from the entropy representation of the version obfuscated by reordering the source code with the random insertion of 50 jumps. Thus, if the modifications alter the byte sequence in a way that the patterns learned by the ML classifiers do not occur, then the sample will be misclassified. In addition, the higher complexity of the aforementioned architectures have negatively affected its performance. On the contrary, it can be observed in Figure 15 that the shallow-based CNN presented in Section 5.2.3 is resilient to the changes performed by the code reordering through jumps technique.

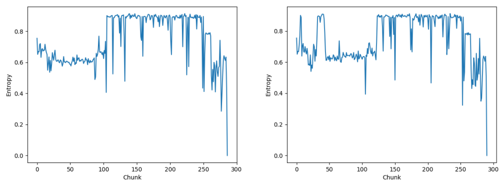


Figure 16: Structural entropy comparison between the non-obfuscated and obfuscated versions of the malicious sample with ID *bxED6RSpmnWV03kyMLoK*.

6.3.5. Mixed Obfuscation

Finally, the samples in the test set have been altered by various combinations of the four metamorphic techniques to mimic the changes performed by a metamorphic engine. To this end, four experiments have been performed:

- Experiment A: Each sample in the test set has been modified by inserting 10 dead code insertions, performing 10 subroutine reorderings, 10 code reorderings through jumps, and by swapping all four data registers.

19

- Experiment B: Each sample in the test set has been modified by inserting 50 dead code insertions, performing 20 subroutine reorderings, 20 code reorderings through jumps, and by swapping all four data registers.
- Experiment C: Each sample in the test set has been modified by inserting 100 dead code insertions, performing 30 subroutine reorderings, 30 code reorderings through jumps, and by swapping all four data registers.
- Experiment D: Each sample in the test set has been modified by inserting 200 dead code insertions, performing 40 subroutine reorderings, 40 code reorderings through jumps, and by swapping all four data registers.

Table 6: Comparison of the macro f1-score achieved by the ML classifiers on the test set obfuscated with various metamorphic techniques.

			Experiment			
	Method	Test Set	A	B	C	D
Opcode-based	(Gibert et al., 2017)	0.9852	0.9754	0.9718	0.9632	0.9529
	NN opcodes (hashing trick)	0.9765	0.9776	0.9756	0.9618	0.9159
	NN opcodes (mutual information)	0.9876	0.9889	0.9889	0.9878	0.9821
Byte-based	MalConv (Raff et al., 2018a)	0.8480	0.8496	0.8340	0.8353	0.8323
	DeepConv (Krčál et al., 2018)	0.8376	0.6981	0.6746	0.6738	0.6322
	Structural entropy CNN (Gibert et al., 2018)	0.8809	0.7988	0.7674	0.7360	0.7115
	Multiresolution CNN (Gibert et al., 2018)	0.8972	0.8215	0.7872	0.7533	0.7254
	NN bytes (hashing trick)	0.8858	0.8715	0.8744	0.8842	0.8727
	Shallow CNN	0.9748	0.9576	0.9499	0.9652	0.9461
	Dilated CNN	0.9622	0.9291	0.9376	0.9331	0.9216
	CNN GLU	0.9627	0.9578	0.9569	0.9567	0.9570

Table 6 and Figure 17 show the performance of the ML classifiers on the obfuscated test sets. Similarly to the previous experiments, opcode-based approaches achieve better results than byte-based approaches. In addition, it can be observed that approaches based on the manual extraction of n-gram features are more resilient to metamorphic techniques as they are mostly unaffected by them. On the other hand, from those approaches based on deep learning, the opcode-based CNN (Gibert et al., 2017) is the one that achieved

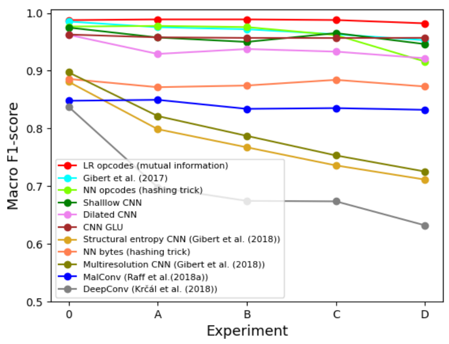


Figure 17: Macro f1-score of ML classifiers on the test set obfuscated with various metamorphic techniques.

the highest macro f1-score while DeepConv (Krčál et al., 2018) is the ML model that was most negatively affected by the modifications in the malware's code. Our intuition is that the size of its filters and the complexity of the network, jointly with the imbalanced data played a deep role.

6.4. Summary

Table 7 compares the performance of the ML classifiers on the obfuscated test sets. To sum up, opcode-based approaches (Gibert et al. (2017), NN opcodes (hashing trick), LR opcodes (mutual information)) perform considerable better than byte-based approaches (Raff et al. (2018a); Krčál et al. (2018); Gibert et al. (2018), NN bytes (hashing trick)), with the logistic regression classifier achieving the highest macro f1-score in all the obfuscated sets. The strong performance of n-gram features have already been investigated in the literature, e.g. Zhang et al. (2016) and the Winner's solution of the Microsoft Malware Classification Challenge [7], in which proved

[7]`https://github.com/xiaozhouwang/kaggle_Microsoft_Malware/tree/master/`

Table 7: Macro F1-score achieved by ML classifiers on the obfuscated test set.

			Dead code insertion					Register's reassignment		Subroutine reorderings				Code reordering through jumps				Mixed			
	Method	Test Set	10	50	100	200	500	A	B	5	10	20	50	5	10	20	50	A	B	C	D
Opcode-based	(Gibert et al., 2017)	0.9852	0.9806	0.9794	0.9711	0.9672	0.8754	0.9852	0.9852	0.9807	0.9801	0.9798	0.9784	0.9789	0.9776	0.9768	0.9729	0.9754	0.9718	0.9632	0.9529
	NN opcodes (hashing trick)	0.9765	0.9754	0.9738	0.9704	0.9458	0.7031	0.9765	0.9765	0.9784	0.9781	0.9785	0.9783	0.9784	0.9777	0.9765	0.9667	0.9776	0.9756	0.9618	0.9159
	NN opcodes (mutual information)	0.9876	0.9873	0.9888	0.9887	0.9879	0.9728	0.9876	0.9876	0.9876	0.9876	0.9874	0.9874	0.9874	0.9876	0.9876	0.9876	0.9889	0.9889	0.9878	0.9821
Byte-based	MalConv (Raff et al., 2018a)	0.8480	0.8543	0.8502	0.8497	0.8300	0.8328	0.8465	0.8294	0.8558	0.8391	0.8645	0.8552	0.8482	0.8573	0.8374	0.8481	0.8496	0.8340	0.8353	0.8323
	DeepConv (Krčál et al., 2018)	0.8376	0.8030	0.7913	0.7567	0.6944	0.6765	0.8136	0.7768	0.8025	0.8150	0.7955	0.7903	0.7547	0.7243	0.7136	0.6749	0.6981	0.6746	0.6738	0.6322
	Structural entropy CNN (Gibert et al., 2018)	0.8809	0.8952	0.9015	0.8880	0.8348	0.7818	0.8850	0.8954	0.8976	0.8929	0.9026	0.8990	0.8416	0.8162	0.7671	0.7319	0.7988	0.7674	0.7360	0.7115
	Multiresolution CNN (Gibert et al., 2018)	0.8972	0.9074	0.9008	0.8880	0.8297	0.7995	0.8973	0.8996	0.9044	0.8998	0.9013	0.9008	0.8453	0.8195	0.7798	0.7457	0.8215	0.7872	0.7533	0.7254
	NN bytes (hashing trick)	0.8858	0.8863	0.8854	0.8864	0.8863	0.8861	0.8846	0.8764	0.8859	0.8860	0.8852	0.8851	0.8841	0.8954	0.8907	0.8955	0.8715	0.8744	0.8842	0.8727
	Shallow CNN	**0.9748**	**0.9733**	**0.9731**	**0.9694**	**0.9674**	**0.9577**	**0.9708**	**0.9565**	**0.9732**	**0.9733**	**0.9731**	**0.9734**	**0.9732**	**0.9739**	**0.9733**	**0.9743**	**0.9576**	**0.9499**	**0.9652**	**0.9461**
	Dilated CNN	**0.9622**	**0.9369**	**0.9362**	**0.9427**	**0.9317**	**0.9280**	**0.9410**	**0.9308**	**0.9465**	**0.9418**	**0.9474**	**0.9560**	**0.9503**	**0.9535**	**0.9605**	**0.9609**	**0.9291**	**0.9376**	**0.9331**	**0.9216**
	CNN GLU	**0.9627**	**0.9627**	**0.9627**	**0.9623**	**0.9606**	**0.9638**	**0.9638**	**0.9570**	**0.9623**	**0.9627**	**0.9626**	**0.9623**	**0.9625**	**0.9628**	**0.9627**	**0.9607**	**0.9579**	**0.9569**	**0.9567**	**0.9570**

to be decisive features for the construction of their classifier.

On the other hand, although deep learning approaches have shown great adoption in recent years by the cybersecurity industry, they are still in an early stage and there still exist margin for improvement. As observed in Table 7, the performance of the deep learning approaches varies greatly depending on the input of the network. For instance, the opcode-based shallow model (Gibert et al., 2017) performance degraded considerably when 500 random dead code instructions were inserted within the source code of the samples in the test set mainly because some bias in the dataset. Regarding byte-based approaches, those that take as input the raw byte sequences (Raff et al., 2018a; Krčál et al., 2018) are mostly affected by the register reassignment technique while entropy-based approaches (Gibert et al., 2018) demonstrated robustness against it. Furthermore, the performance of all deep learning approaches, indistinctly of their input, show some degradation with respect to the changes performed by the code reordering through jumps technique. In addition, it can be observed that there exist a huge gap in the performance of opcode-based (Gibert et al., 2017) and byte-based approaches (Raff et al., 2018a; Krčál et al., 2018; Gibert et al., 2018) deep learning methods in the literature, achieving a macro f1-score on the test set equals to 0.9852, 0.8480, 0.8376, 0.8972, respectively. This is because byte-based approaches failed to correctly classify samples belonging to the minority classes, e.g. Simda, Obfuscator.ACY, etc. This is attributable to several factors: (1) the complexity and depth of the network architectures; (2) the size of the filters and (3) class imbalance. As it is shown in Table 7, the proposed shallow architecture trained on the raw bytes sequences with filters of various sizes ranging from $k \in \{3, 5, 7\}$ achieves a macro f1-score comparable to the opcode-based approaches and 14.95% and 16.38% higher than MalConv's (Raff et al., 2018a) and DeepConv's (Krčál et al., 2018) architectures, respectively. Thus, it demonstrates that the complexity of the network architectures played an important role in the low output achieved by the byte-based models. In addition, the byte-based shallow CNN architecture has shown greater robustness against the dead code insertion technique than their opcode-based counterpart although it is still affected by the register's reassignment technique as all byte-based approaches.

`kaggle_Microsoft_malware_full`

6.5. Data Augmentation with Adversarial Examples

Recent advances in deep learning have been largely attributed to the quantity and diversity of data. The more data and the

21

more variation possible in the data the better the generalization of the model will be. However, in some cases it is not possible to collect thousands or millions of samples. In such cases, more data can be generated from a given dataset. This process is known as data augmentation. As far as we know, no data augmentation scheme has been proposed in the literature for the malware domain. Following, the use of metamorphic techniques for augmenting the dataset is investigated. The main idea behind using the aforementioned metamorphic techniques to augment the dataset is that the modifications preserve the functionality of the executables and are commonly used by malware authors and thus, it may help build robust ML models. As observed in Table 1 the training set is very imbalanced, with the majority class (Kelihos_ver3) containing 70 times more samples than the minority class (Simda). Subsequently, the number of samples generated for each family varied considerably, with the samples of the minority families reused more than the samples in the majority families to expand the training set. The total number of samples in the training set is shown in Table 8. The augmented training set contains the original sample and one or more obfuscated versions of it. This obfuscated versions have been generated using the following parameters:

- A total of 10 dead code instructions have been inserted.
- 5 subroutines have been randomly permuted.
- The source code has been reordered by inserting 5 jumps to split the subroutines.
- The registers of a given sample have been randomly swapped with probability $p = 0.2$.

Table 8: Class distribution in augmented training set.

Family Name	Training set	Augmented training set
Ramnit	1084	3249
Lollipop	1736	3472
Kelihos_ver3	2057	4114
Vundo	327	2289
Simda	26	546
Tracur	532	2660
Kelihos_ver1	279	2511
Obfuscator.ACY	845	3380
Gatak	721	2884

Table 9 presents the performance of the models trained using the augmented training set on the obfuscated test set. In general, all deep learning approaches gained some robustness against the changes performed by the metamorphic techniques. It can be observed that byte-based approaches improved considerably thanks to the augmented training set. For instance, MalConv's (Raff et al., 2018a) and DeepConv's (Krčál et al., 2018) performance improved 9.98% and 5.03%, respectively. However, their models continue to perform poorly in comparison to the byte-based shallow CNN (0.9748 macro f1-score on the test set), the dominant byte-based classifier. On the other hand, the robustness of the opcode-based CNN (Gibert et al., 2017) improved considerably with respect to the modifications performed by the dead code insertion technique. The macro f1-score achieved by the opcode-based CNN on the test set obfuscated by inserting 500 dead code instructions improved from 0.8754 to 0.9796, an 11.9% increase.

Notice that the macro f1-score achieved by the shallow CNN approaches, indistinctly of whether they take as input the bytes or opcode sequences, is close to those

Table 9: Macro F1-score achieved by ML classifiers on the obfuscated test set trained on the augmented dataset (AD).

			Dead code insertion					Register's Reassignment		Subroutine reorderings				Code reordering through jumps				Mixed			
	Method	Test Set	10	50	100	200	500	A	B	5	10	20	50	5	10	20	50	A	B	C	D
Opcode-based	(Gibert et al., 2017)	0.9852	0.9806	0.9794	0.9711	0.9672	0.8754	0.9852	0.9852	0.9807	0.9801	0.9798	0.9784	0.9789	0.9776	0.9768	0.9729	0.9754	0.9718	0.9632	0.9529
	(Gibert et al., 2017), AD	0.9850	0.9854	0.9835	0.9855	0.9836	0.9796	0.9850	0.9850	0.9849	0.9848	0.9845	0.9850	0.9835	0.9821	0.9811	0.9781	0,9821	0,9791	0,9787	0,9767
	NN opcodes (hashing trick)	0.9765	0.9754	0.9738	0.9704	0.9458	0.7031	0.9765	0.9765	0.9784	0.9781	0.9785	0.9783	0.9784	0.9777	0.9765	0.9667	0.9776	0.9756	0.9618	0.9159
	NN opcodes (hashing trick, AD)	0.9862	0.9827	0.9833	0.9734	0.9298	0.7659	0.9862	0.9862	0.9862	0.9862	0.9862	0.9862	0.9859	0.9855	0.9859	0.9820	0,9857	0,9820	0,9684	0,9103
	LR opcodes (mutual information)	0.9876	0.9873	0.9888	0.9887	0.9879	0.9728	0.9876	0.9876	0.9876	0.9876	0.9874	0.9874	0.9874	0.9876	0.9876	0.9876	0.9889	0.9889	0.9878	0.9821
	LR opcodes (mutual information, AD)	0.9873	0.9875	0.9876	0.9852	0.9858	0.9836	0.9873	0.9873	0.9878	0.9875	0.9875	0.9880	0.9873	0.9875	0.9873	0.9870	0.9875	0,9875	0,9865	0,9865
Byte-based	MalConv (Raff et al., 2018a)	0.8480	0.8543	0.8502	0.8497	0.8300	0.8328	0.8465	0.8294	0.8558	0.8391	0.8645	0.8552	0.8482	0.8573	0.8374	0.8481	0.8496	0.8340	0.8353	0.8323
	MalConv (Raff et al., 2018a), AD	0.9326	0.9483	0.9399	0.9434	0.9398	0.9422	0,9339	0.9388	0.9427	0.9415	0.9401	0.9396	0.9341	0.9387	0.9404	0.9453	0.9367	0.9346	0,9335	0.9280
	DeepConv (Krčál et al., 2018)	0.8376	0.8030	0.7913	0.7567	0.6944	0.6765	0.8136	0.7768	0.8025	0.8150	0.7955	0.7903	0.7547	0.7243	0.7136	0.6749	0.6981	0.6746	0.6738	0.6322
	DeepConv (Krčál et al., 2018), AD	0.8797	0.8812	0.8809	0.8680	0.8585	0.8252	0,8762	0.8712	0.8804	0.8802	0.8760	0.8741	0.8653	0.8650	0.8513	0.8523	0.8541	0,8474	0,8409	0,8388
	Structural entropy CNN (Gibert et al., 2018)	0.8809	0.8952	0.9015	0.888	0.8348	0.7818	0.8850	0.8954	0.8976	0.8929	0.9026	0.8990	0.8416	0.8162	0.7671	0.7319	0.7988	0.7674	0.7360	0.7115
	Structural entropy CNN (Gibert et al., 2018), AD	0.8904	0.8996	0.9022	0.8941	0.9223	0.8904	0,8988	0.8941	0.8987	0.8970	0.9012	0.9019	0.9023	0.8911	0.8796	0.8759	0.8806	0.8887	0,8744	0,8678
	Multiresolution CNN (Gibert et al., 2018)	0.8972	0.9074	0.9008	0.8880	0.8297	0.7995	0.8973	0.8996	0.9044	0.8998	0.9013	0.9008	0.8453	0.8195	0.7798	0.7457	0.8215	0.7872	0.7533	0.7254
	Multiresolution CNN (Gibert et al., 2018), AD	0.9261	0.9244	0.9256	0.9305	0.9286	0.8335	0,9264	0.9319	0.9241	0.9233	0.9282	0.9244	0.9182	0.9215	0.897	0.8954	0.9179	0,9052	0.8945	0.8784
	NN bytes (hashing trick)	0.8858	0.8863	0.8854	0.8864	0.8863	0.8861	0.8846	0.8764	0.8859	0.8860	0.8852	0.8851	0.8841	0.8954	0.8907	0.8955	0.8715	0.8744	0.8842	0.8727
	NN bytes (hashing trick, AD)	0.9539	0.9498	0.9474	0.9492	0.9492	0.9465	0,9410	0.9363	0.9491	0.9522	0.9494	0.9489	0.9491	0.9437	0.9464	0.9403	0.9439	0.9349	0.9448	0,9360
	Shallow CNN	0.9748	0.9733	0.9731	0.9694	0.9674	0.9577	0.9708	0.9565	0.9732	0.9733	0.9731	0.9734	0.9732	0.9739	0.9733	0.9743	0.9576	0.9499	0.9652	0.9461
	Shallow CNN, AD	0.9765	0.9762	0.9740	0.9744	0.9745	0.9609	0,9723	0.9748	0.9763	0.9762	0.9760	0.9712	0.9765	0.9769	0.976	0.9753	0.9690	0,9734	0,9718	0,9618

obtained by the n-gram based approaches. In the case of the byte-based CNN it achieves higher macro f1-score than the n-gram based approach while the opcode-based CNN macro f1-score is marginally lower than the n-gram based counterpart. Thus, demonstrating that deep learning approaches are a good alternative to n-gram based approaches without the computational and memory costs of having to exhaustively enumerate millions of features and manually perform feature extraction and selection during training.

7. Conclusions & Future Work

This paper provides an exhaustive evaluation of the vulnerability of state-of-the-art anti-malware engines to the changes in the source code generated by the following metamorphic techniques: (1) the dead code insertion technique, (2) the register's reassignment technique, (3) the subroutine reordering technique and (4) the code reordering through jumps technique. Results show that byte-based approaches perform poorly in comparison with opcode-based approaches and are not robust to the changes caused by the register's reassignment technique. Their lower yield is attributable to several factors: (1) the size of the filters, (2) the complexity of the network and (3) the data imbalance. On the other hand, the shallow architecture presented in this paper has achieved an improvement of 14.95% and 16.38% with respect to MalConv's (Raff et al., 2018a) and DeepConv's (Krčál et al., 2018) architectures, respectively, and attains similar classification performance in comparison to opcode-based approaches (Gibert et al., 2017). Furthermore, the usage of metamorphic techniques to augment the training set has been investigated. Results show that the classification performance of deep learning approaches improves considerably and gain robustness against metamorphism independently of their input data. Thus, we demonstrate the feasibility of augmenting the training data, and in particular the number of samples belonging to the minority classes, by employing metamorphic techniques for the malware classification task.

A future line of research is the exploration of encryption and compression techniques, and the investigation of their effects in ML classifiers.

Acknowledgements

This research has been partially funded by the Spanish MICINN Projects TIN2015-71799-C2-2-P, ENE2015-64117-C5-1-R, PID2019-111544GB-C22, and supported by

the University of Lleida.

Conflicts of Interest

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

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X., 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. URL: `http://tensorflow.org/`. software available from tensorflow.org.

Amro, S.A., Alkhalifah, A., 2015. A comparative study of virus detection techniques. International Journal of Computer and Information Engineering 9, 1559 – 1566. URL: `https://publications.waset.org/vol/102`.

Anderson, H.S., Kharkar, A., Filar, B., Evans, D., Roth, P., 2018. Learning to evade static PE machine learning malware models via reinforcement learning. CoRR abs/1801.08917. URL: `http://arxiv.org/abs/1801.08917`, `arXiv:1801.08917`.

Bellman, R.E., 2015. Adaptive control processes: a guided tour. Princeton university press.

Chen, L., 2009. Curse of Dimensionality. Springer US, Boston, MA. pp. 545–546. doi:`10.1007/978-0-387-39940-9_133`.

Dauphin, Y.N., Fan, A., Auli, M., Grangier, D., 2017. Language modeling with gated convolutional networks, in: Proceedings of the 34th International Conference on Machine Learning - Volume 70, JMLR.org. p. 933–941.

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., 2019. Explaining vulnerabilities of deep learning to adversarial malware binaries, in: 3rd Italian Conference on Cyber Security, ITASEC 2019, CEUR Workshop Proceedings. CEUR Workshop Proceedings, Pisa, Italy. URL: `https://arxiv.org/abs/1901.03583`.

Gibert, D., Bejar, J., Mateu, C., Planes, J., Solis, D., Vicens, R., 2017. Convolutional neural networks for classification of malware assembly code, in: International Conference of the Catalan Association for Artificial Intelligence, pp. 221–226. doi:`10.3233/978-1-61499-806-8-221`.

Gibert, D., Mateu, C., Planes, J., 2020a. Orthrus: A bimodal learning architecture for malware classification, in: Proceedings of the International Joint Conference on Neural Networks (IJCNN) 2020, Glassgow (UK), IEEE.

Gibert, D., Mateu, C., Planes, J., 2020b. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. Journal of Network and Computer Applications 153, 102526. URL: `http://www.sciencedirect.com/science/article/pii/S1084804519303868`, doi:`https://doi.org/10.1016/j.jnca.2019.102526`.

Gibert, D., Mateu, C., Planes, J., Vicens, R., 2018. Classification of malware by using structural entropy on convolutional neural networks, in: Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, pp. 7759–7764.

Hu, W., Tan, Y., 2017. Generating adversarial malware examples for black-box attacks based on GAN. CoRR abs/1702.05983. URL: `http://arxiv.org/abs/1702.05983`, `arXiv:1702.05983`.

Hu, X., Shin, K.G., Bhatkar, S., Griffin, K., 2013. Mutantx-s: Scalable malware clustering based on static features, in: Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), USENIX, San Jose, CA. pp. 187–198. URL: `https://www.usenix.org/conference/atc13/technical-sessions/presentation/hu`.

Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F., 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables, in: 26th European Signal Processing Conference, EUSIPCO 2018, Roma, Italy, September

3-7, 2018, IEEE. pp. 533–537. URL: https://doi.org/10.23919/EUSIPCO.2018.8553214, doi:10.23919/EUSIPCO.2018.8553214.

Krčál, M., Švec, O., Bálek, M., Jašek, O., 2018. Deep convolutional malware classifiers can learn from raw executables and labels only. URL: https://openreview.net/pdf?id=HkHrmM1PM.

Mcgraw, G., Bonett, R., Figueroa, H., Shepardson, V., 2019. Security engineering for machine learning. Computer 52, 54–57.

Pitropakis, N., Panaousis, E., Giannetsos, T., Anastasiadis, E., Loukas, G., 2019. A taxonomy and survey of attacks against machine learning. Computer Science Review 34, 100199. URL: http://www.sciencedirect.com/science/article/pii/S1574013718303289, doi:https://doi.org/10.1016/j.cosrev.2019.100199.

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.K., 2018a. Malware detection by eating a whole EXE, in: The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018., pp. 268–276.

Raff, E., Nicholas, C., 2018. Hash-grams: Faster n-gram features for classification and malware detection, in: Proceedings of the ACM Symposium on Document Engineering 2018, Association for Computing Machinery, New York, NY, USA. URL: https://doi.org/10.1145/3209280.3229085, doi:10.1145/3209280.3229085.

Raff, E., Zak, R., Cox, R., Sylvester, J., Yacci, P., Ward, R., Tracy, A., McLean, M., Nicholas, C., 2018b. An investigation of byte n-gram features for malware classification. Journal of Computer Virology and Hacking Techniques 14, 1–20. URL: https://doi.org/10.1007/s11416-016-0283-1, doi:10.1007/s11416-016-0283-1.

Raff, E., Zak, R., Cox, R., Sylvester, J., Yacci, P., Ward, R., Tracy, A., McLean, M., Nicholas, C., 2018c. An investigation of byte n-gram features for malware classification. Journal of Computer Virology and Hacking Techniques 14, 1–20. URL: https://doi.org/10.1007/s11416-016-0283-1, doi:10.1007/s11416-016-0283-1.

Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M., 2018. Microsoft malware classification challenge. CoRR abs/1802.10135. URL: http://arxiv.org/abs/1802.10135, arXiv:1802.10135.

Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G., 2013. Opcode sequences as representation of executables for data-mining-based unknown malware detection. Information Sciences 231, 64 – 82. URL: http://www.sciencedirect.com/science/article/pii/S0020025511004336, doi:https://doi.org/10.1016/j.ins.2011.08.020. data Mining for Information Security.

Souri, A., Hosseini, R., 2018. A state-of-the-art survey of malware detection approaches using data mining techniques. Human-centric Computing and Information Sciences 8, 3. URL: https://doi.org/10.1186/s13673-018-0125-x, doi:10.1186/s13673-018-0125-x.

Suciu, O., Coull, S.E., Johns, J., 2019. Exploring adversarial examples in malware detection, in: 2019 IEEE Security and Privacy Workshops, SP Workshops 2019, San Francisco, CA, USA, May 19-23, 2019, IEEE. pp. 8–14. URL: https://doi.org/10.1109/SPW.2019.00015, doi:10.1109/SPW.2019.00015.

Ször, P., Ferrie, P., 2001. Hunting for metamorphic, in: In Virus Bulletin Conference, pp. 123–144.

Ucci, D., Aniello, L., Baldoni, R., 2019. Survey of machine learning techniques for malware analysis. Computers & Security 81, 123 – 147. URL: http://www.sciencedirect.com/science/article/pii/S0167404818303808, doi:https://doi.org/10.1016/j.cose.2018.11.001.

Vergara, J.R., Estévez, P.A., 2014. A review of feature selection methods based on mutual information. Neural Computing and Applications 24, 175–186. URL: https://doi.org/10.1007/s00521-013-1368-0, doi:10.1007/s00521-013-1368-0.

Yu, F., Koltun, V., 2016. Multi-scale context aggregation by dilated convolutions. CoRR abs/1511.07122.

Zhang, F., Zhao, T., 2017. Malware detection and classification based on n-grams attribute similarity, in: 2017 IEEE International Conference on Computational Science and Engineering, CSE 2017, and IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2017, Guangzhou, China, July 21-24, 2017, Volume 1, IEEE Computer Society. pp. 793–796. URL: https://doi.org/10.1109/CSE-EUC.2017.157, doi:10.

`1109/CSE-EUC.2017.157.`

Zhang, Y., Huang, Q., Ma, X., Yang, Z., Jiang, J., 2016. Using multi-features and ensemble learning method for imbalanced malware classification, in: 2016 IEEE Trustcom/BigDataSE/ISPA, pp. 965–973.

Appendix A. Neural Network Architectures.

Appendix A.1. Shallow CNN (Gibert et al., 2017).

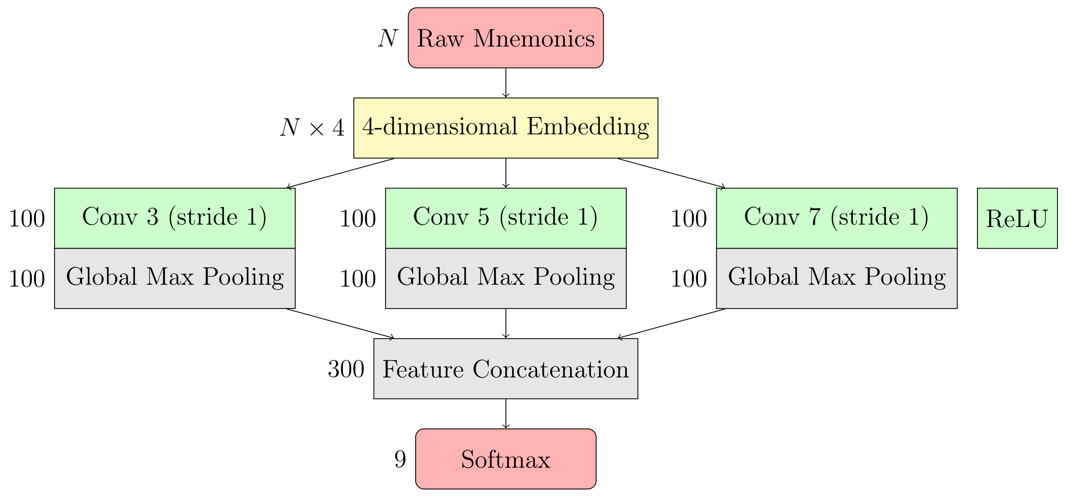


Figure A.18: Opcode-based shallow convolutional neural network (Gibert et al., 2017).

27

Appendix A.2. MalConv (Raff et al., 2018a).

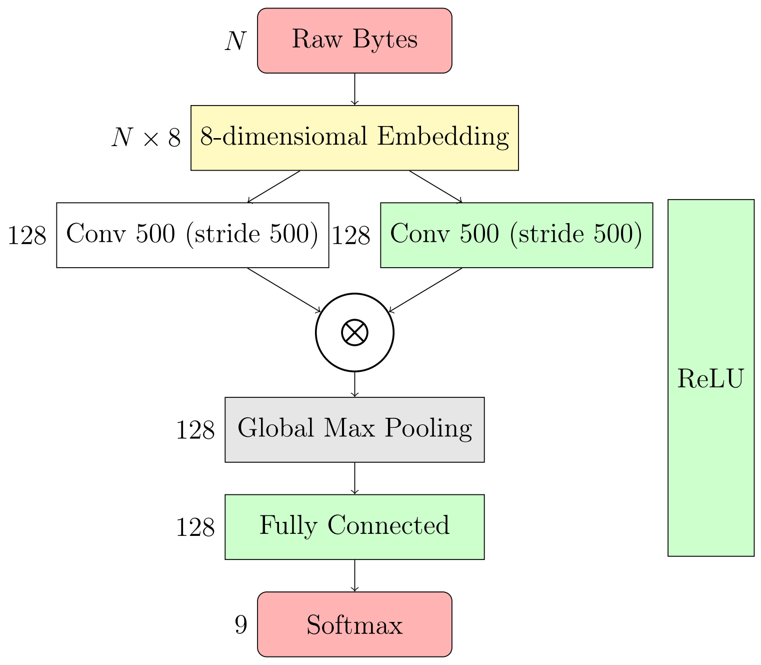


Figure A.19: MalConv architecture (Raff et al., 2018a).

28

Appendix A.3. DeepConv (Krčál et al., 2018).

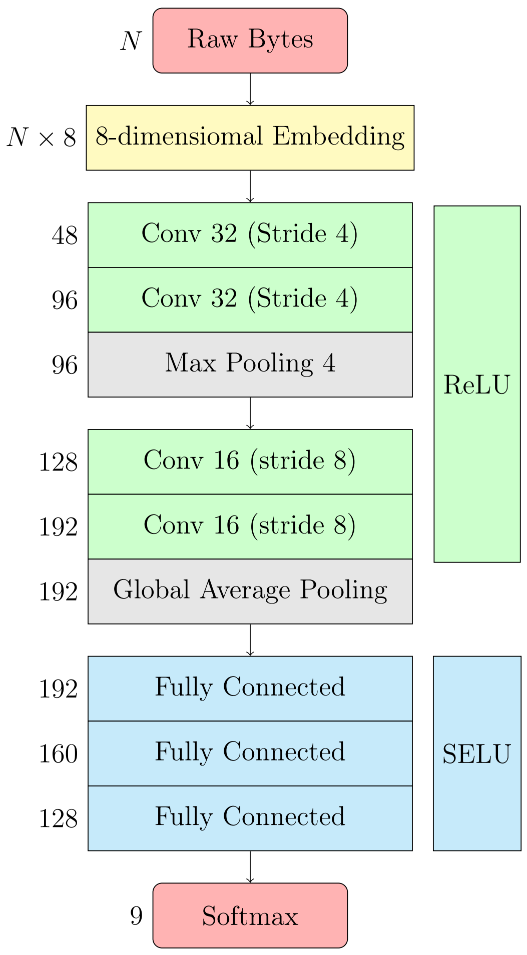


Figure A.20: DeepConv architecture (Krčál et al., 2018).

Appendix A.4. Structural entropy CNN and Multiresolution CNN (Gibert et al., 2018)

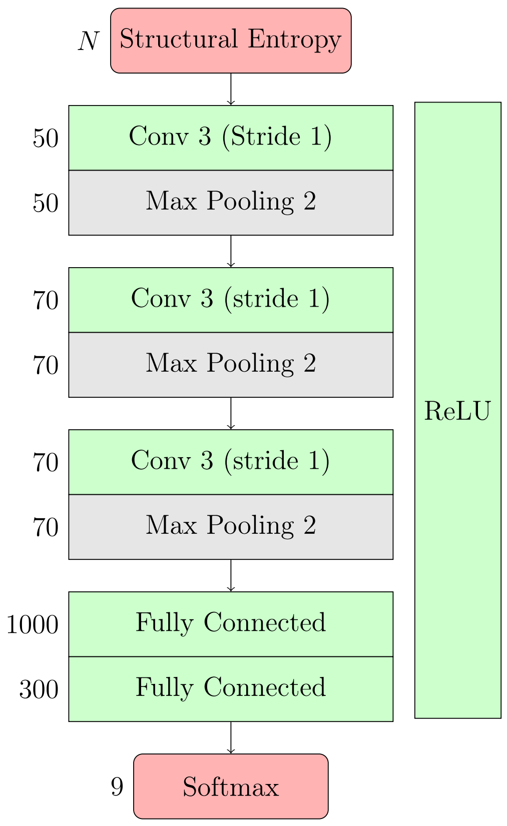


Figure A.21: Structural Entropy CNN (Gibert et al., 2018).

30

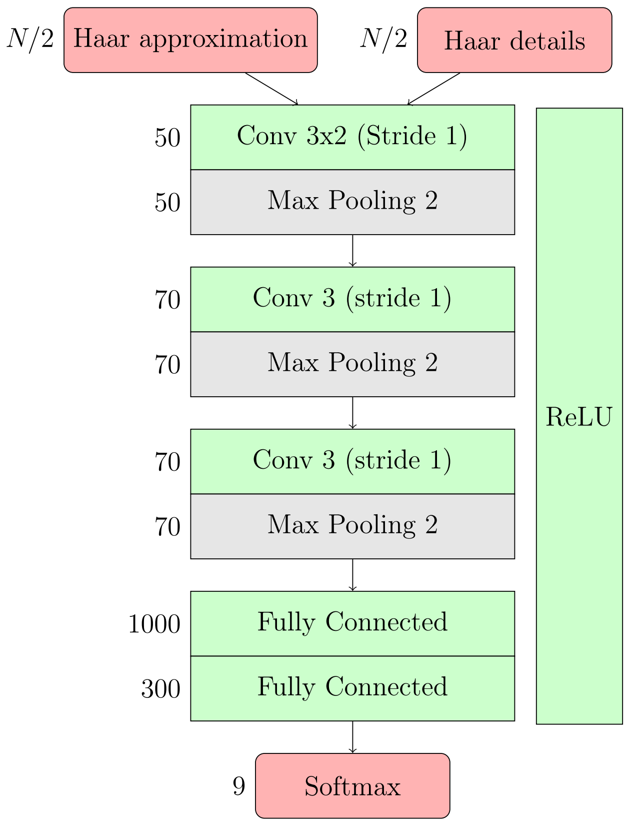


Figure A.22: Multiresolution CNN (Gibert et al., 2018).

31

9.5 Interpretability of the Models

The interpretation of machine learning models is a new and open challenge [68, 33]. Most of the models used at the present time are treated as a black box. This black box is given an input X and produces an output Y through a sequence of operations hardly understandable to a human. This could pose a problem in cybersecurity applications when a false alarm occurs as analysts would like to understand why it happened. The interpretability of the model determines how easily the analysts can manage and assess the quality and correct the operation of a given model. For this reason, cybersecurity analysts have preferred solutions that are more interpretable and understandable such as rule-based and signature-based systems rather than neural-based methods because they are easier to tune and optimize to mitigate and control the effect of false positives and false negatives. However, there is no work in the literature that investigates the interpretability of machine learning models for malware detection and classification.

Chapter 10

Other Research Activities

10.1 Contributions to Conferences

The author of this thesis has contributed with the following conference papers:

- D.Gibert, C.Mateu, J.Planes, D.Solis, R.Vicens, Convolutional Neural Networks for Classification of Malware Assembly Code, Recent Advances in Artificial Intelligence Research and Development: Proceedings of the 20th International Conference of the Catalan Association for Artificial Intelligence, Deltebre, Terres de L'Ebre, Spain, October 25-27, 2017.

- D. Gibert, C.Mateu, J.Planes, R.Vicens, Classification of Malware by Using Structural Entropy on Convolutional Neural Networks, Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018.

- D. Gibert, C. Mateu, J.Planes, An End-to-End Deep Learning Architecture for Classification of Malware's Binary Content, Artificial Neural Networks and Machine Learning - ICANN 2018 - 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part III

- D.Gibert, C.Mateu, J.Planes, A Hierarchical Convolutional Neural Network for Malware Classification, International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019.

- D.Gibert, A.Lamas, R.Martins, C.Mateu, J.Planes, An Android Malware Detection Framework Using Graph Embeddings and Convolutional Neural Networks, Artificial Intelligence Research and Development - Proceedings of the 22nd International Conference of the Catalan Association for Artificial Intelligence, CCIA 2019, Mallorca, Spain, 23-25 October 2019.

- D.Gibert, C.Mateu, J.Planes, Orthrus: A Bimodal Learning Architecture for Malware Classification, IEEE World Congress on Computational Intelligence (WCCI 2020), Proceedings of the International Joint Conference on Neural Networks, IJCNN 2020 Glassgow, United Kingdom, July 19-24, 2020.

10.2 An Android Malware Detection Framework Using Graph Embeddings and Convolutional Neural Networks

The article *An Android Malware Detection Framework Using Graph Embeddings and Convolutional Neural Networks* [25] has been published in the Proceedings of the 22nd International Conference of the Catalan Association for Artificial Intelligence (CCIA 2019). This research article has been included in Section 10 because it presents an approach to tackle the problem of Android malware identification and it is out of scope of the main line of research of this dissertation.

This article proposes a novel framework to identify malware targeting the Android operative system (OS) based on the function call graph representation of applications (APKs). The framework is composed by three main components: (1) the construction of the function call graph, (2) the generation of the graph embedding representation and (3) the extraction of patterns and the identification of malware.

1. Function call graph extraction. The first component is responsible of extracting the function call graph (FCG) of a given APK. A call graph is a directed graph G = (V, E) whose vertices V , denote the functions a program or application is composed of, and the edges E denote the function calls be- tween them.

2. The second component's aim is to generate a fixed size representation for every call graph. To do so, we represent every graph G as a sequence of random walks. As a result, a given graph G is represented as a matrix $x^{m,k}$, where m denotes the number of random walks and k the size of each random walk.

3. Learning and Identification. After the embedding of an APK is generated, the detection process is conducted to determine whether a given application is malicious or not. Therefore, a convolutional neural network architecture is presented to get as input the graph embedding and extract characterizing patterns that are indicative of maliciousness from it. Afterwards, the model produces the estimated label for a given application.

The presented method has been evaluated against DREBIN, a standard benchmark for Android malware identification and results show that the detection results achieved by our approach slightly higher than D. Arp et al. [5] while not relying on expertise knowledge of the domain to extract a discriminant set of features. However, the increase in accuracy came at the expense of an increase in the overall computational time required to employ our approach in comparison to D. Arp et al. [5], as their approach only extracts features from the manifest and the disassembled dex code of Android applications and thus, it avoids time-consuming extraction of the call graph. Nevertheless, our approach demonstrates that an accurate featureless approach for Android malware detection is possible.

An Android Malware Detection Framework Using Graph Embeddings and Convolutional Neural Networks

Daniel GIBERT [a,1], Alba LAMAS [a], Ruben MARTINS [b], Carles MATEU [a] and Jordi PLANES [a]

[a] *University of Lleida, Spain*
[b] *Carnegie Mellon University, USA*

Abstract. With the widespread use of mobile phones, the number of malware targeting smart devices has increased exponentially. In particular, the number of malware targeting Android devices, as it is the most popular operative system among smartphones. This paper proposes a novel framework for android malware detection based on the function call graph representation of an application. Our method generates an embedding of the function call graph using random walks and then, a convolutional neural network extracts features from their embedded matrix representation and labels a given application as benign or malicious considering the learned features. The method has been evaluated on a dataset of 3871 APKs and compared against DREBIN, a baseline benchmark. Experiments show that the method achieves competitive results without relying on the manual extraction of features.

Keywords. malware detection, android security, graph representation, convolutional neural networks

1. Introduction

Nowadays, Android is one of the most popular O.S. for smartphones. According to various sources [2] [3], Android has a market share of approximately 87.5% with respect to other mobile operating systems. In consequence, the widespread adoption of Android devices has resulted in an increase of mobile malware, more specifically, in Android malware, as it is much more profitable than targeting any other operative system. In particular, McAfee [4] detected an all-time high of 57.6 million new samples at the end of 2017, with mobile malware jumping by 60% in Q3, mainly due to an increase in Android screen-locking ransomware. In 2018 there were fewer infections due to better native Android security protections. This resulted in a decline in drop in ”rooter” attacks, drop in ”clickers” and downloaders. However, the number of infections produced by most other cate-

[1]Corresponding Author: Daniel Gibert, University of Lleida, Spain; E-mail: daniel.gibert@diei.udl.cat
[2]https://www.cnbc.com/2016/11/03/google-android-hits-market-share-record-with-nearly-9-in-every-10-smartphones-using-it.html
[3]https://qz.com/826672/android-goog-just-hit-a-record-88-market-share-of-all-smartphones/
[4]https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-dec-2017.pdf

gories, such as ad-based malware, banking trojans and fake apps, increased. According to Avast [5], adverstising, phishing and fake apps are expected to dominate the landscape in the near future.

Android applications are distributed through what is known as an app store or marketplace. The largest distribution channel is the Google Play Store, but many alternatives exist. For example, in China, Google Play is not officially available (although it can still be reached) and users usually download applications from alternative app stores such as Tencent's MyApp, 360 Mobile Assistant or Baidu's App Store. To ensure the quality of the apps, Google Play Store removes the applications it considers violate their policies periodically [9]. However, from time to time malicious apps pass the security filters and become available for download.

Android's main defense against malware is a permission-based mechanism, which warn users about the permissions an application requires previously to its installation or at runtime, depending on the version of the operative system. Permissions are divided into several protection levels: normal, signature, and dangerous permissions, depending on the risk to user's privacy or the operation of other apps. This approach is ineffective as it does not provide enough information to conclude whether an application is indeed malicious. Furthermore, it requires too much technical knowledge for a standard user to differentiate between malicious and benign applications, as most of the time these permissions are ignored or not understood by users [5,7].

In recent years, machine learning algorithms have been used to design systems for malware detection. The typical workflow of these systems include the following components: (1) feature extraction, (2) feature selection/reduction, (3) model learning and (4) deployment. More specifically, the features used for detecting Android malware range from simple features, such as string-, permission-, opcode- or API-based features, to more complex features based on the function call graph (FCG) representation of the applications.

In this paper we present a novel malware detection method that automatically learns to extract discriminative features from an embedded representation of an Android application's function call graph using convolutional neural networks. The method can be decomposed into three main components. First, the function call graph of the application is extracted. Second, the call graph is embedded as a matrix using random walks. Finally, a convolutional neural network takes as input the embedded representation of an application and determines its maliciousness. To demonstrate the suitability of our approach we evaluate our model on a dataset of 3871 APKs and we provide a comparison with DREBIN, a baseline benchmark. Experiments show that the method achieves competitive results without relying on the extraction of hand-engineered features.

The rest of the paper is organized as follows. Section 2 discusses past research. Section 3 provides a description of our method. Section 4 evaluates the performance of our approach. Finally, Section 5 concludes with our remarks and future work suggestions.

2. Related Work

Various machine learning based approaches have been proposed during the last decade to tackle the problem of malware detection. Many of these approaches mainly rely on

[5]https://blog.avast.com/avast-mobile-threat-predictions

feature engineering using either static or dynamic analysis of the APKs. These features range from simple ones, such as string-, permission-, opcode- or API-based features, to more complex features such as the ones extracted from the Function Call Graph or the Inter-Component Call Graph of the applications.

The first notable Android malware detectors are based on static analysis of the DPK: DroidAPIMiner and Drebin. DroidAPIMiner [1], a lightweight classifier based on features extracted from the API level information within the bytecode. In particular, they focus on critical API calls, their package level information and their parameters. Afterwards, a K-NN takes as input a subset of the top APIs to classify malware from benign applications. Drebin [3] used static analysis to extract features from the manifest and the disassembled code of an application such as restricted API calls, used permissions, network addresses, etc.,and embeds them in a joint vector space to classify malware. A linear support vector machine was considered in their work to learn a separation between benign and malicious applications.

Two more detectors were build going beyond static analysis: Maladroid and DroidSec. MaMaDROID [8] is an android malware detection system that relies on app behavior to build a behavioral model, in the form of a Markov chain, from the sequence of API calls performed by an app. Then, these API calls are used to extract features and perform classification. Z. Yuan et al. [10] built a detection system based on more than 200 features extracted from both static and dynamic analysis of Android applications. Then, a Deep Belief Network is trained to detect malicious applications.

We also highlight two applications which rely on a graph representation of the APK, store graphs from malicious applications and compare them to detect the new malware. DroidSIFT [11] is a system that constructs features from a weighted contextual API dependency graph. Then, a graph similarity algorithm compares the graph of a given application with those inside the graph database. Finally, Apposcopy [6] is a signature matching algorithm that uses a combination of static taint analysis and a representation called Inter-Component Call Graph to detect Android applications with certain malicious control and data-flow properties.

However, all these methods rely on the extraction of hand-crafted features manually designed by an expert in the domain. In addition, DroidSIFT and Apposcopy suffer from heavy runtime overhead due to the need of having to measure graph and signatures similarities among applications, respectively. Instead, our approach generates a fixed embedding representation of the function call graph using random walks. This embedding learns a mapping from a graph to a matrix space, while preserving relevant graph properties, such as neighbourhood. Afterwards, a convolutional neural network is trained to differentiate between benign and malicious applications by automatically learning which subsequences of API function calls are more discriminant.

3. Methodology

In this section, the details of the graph generation and malware detection system are discussed. Cf. Figure 1. The framework is composed by three main components: (1) Construction of the function call graph; (2) Graph embedding representation; and (3) pattern learning and malware identification.

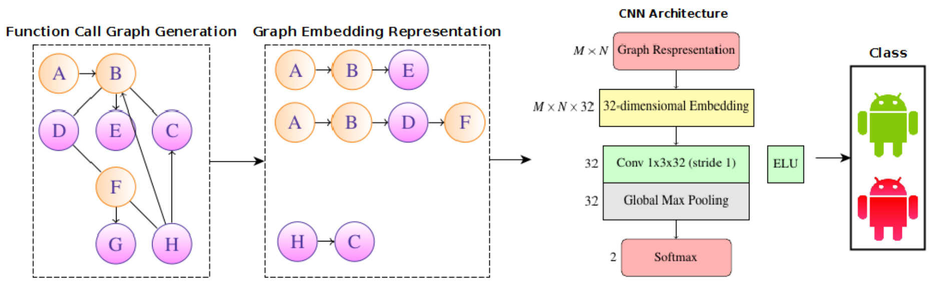

Function Call Graph Generation
Graph Embedding Representation
CNN Architecture
$M \times N$ Graph Respresentation
$M \times N \times 32$ 32-dimensiomal Embedding
32 Conv 1x3x32 (stride 1)
ELU
32 Global Max Pooling
2 Softmax
Class

Figure 1. System overview

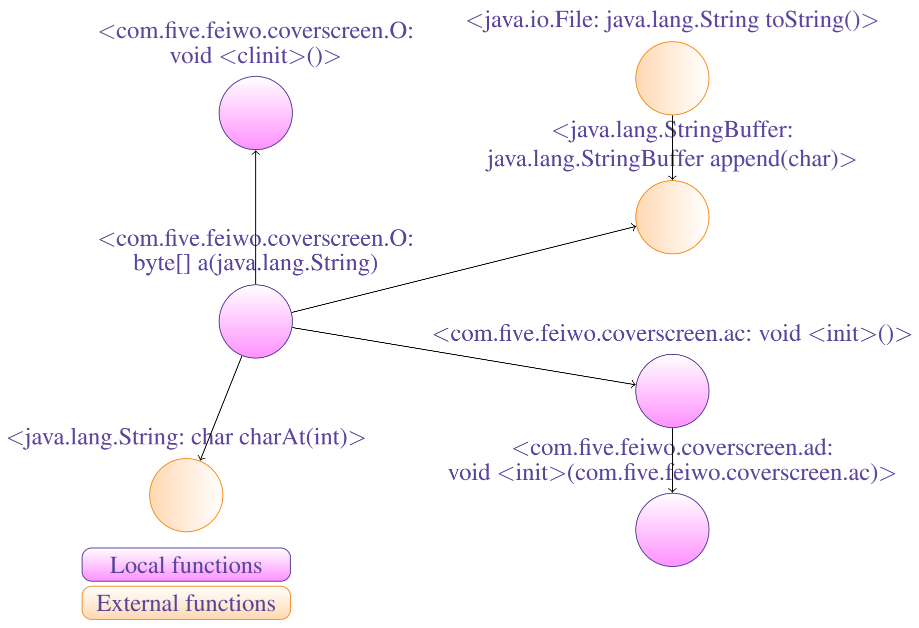

<com.five.feiwo.coverscreen.O: void <clinit>()>
<java.io.File: java.lang.String toString()>
<java.lang.StringBuffer: java.lang.StringBuffer append(char)>
<com.five.feiwo.coverscreen.O: byte[] a(java.lang.String)
<com.five.feiwo.coverscreen.ac: void <init>()>
<java.lang.String: char charAt(int)>
<com.five.feiwo.coverscreen.ad: void <init>(com.five.feiwo.coverscreen.ac)>
Local functions
External functions

Figure 2. Function Call Graph example.

3.1. Function Call Graph Construction

The first component is responsible of extracting the function call graph (FCG) of a given APK. A call graph is a directed graph $G = (V,E)$ whose vertices V, denote the functions a program or application is composed of, and the edges E denote the function calls between them. See Figure 2. Functions are classified in two types: (1) local functions and (2) external functions. On the one hand, local functions are those written by the developers to perform a specific task. Local functions might carry different names in different APKs even though their functionality is the same. On the other hand, external functions are system or library functions and its name is consistent across different applications. The FCGs of every application were generated using FlowDroid [4]. To limit the vocabulary size, a policy was defined to eliminate those nodes that appeared in less than three applications by replacing them for nodes labeled as ”UNK”. This procedure simplified the function call graph and allowed the learning algorithm to focus on the most relevant nodes.

3.2. APK Embedding

The second component's aim is to generate a fixed size representation for every call graph. To do so, we represent every graph G as a sequence of random walks. As a result, a given graph G is represented as a matrix $x^{m,k}$, where m denotes the number of random walks and k the size of each random walk.

A walk is a sequence of vertices $v_1, v_2, v_3, ..., v_{k+1}$ not necessarily distinct, such that (v_i, v_{i+1}) is an edge in the graph. A random walk is a random sequence of points selected as following: Given a graph G and a starting point or node v_j, we select a neighbor of it and move to this neighbor; then we select a neighbor of this point and move to it, etc. The probability that it moves to a neighbor vertex v_i is $\frac{1}{deg(v_j)}$ if (v_j, v_i) is an edge in graph G. $deg(v_j)$ denotes the degree of vertex v_j in a graph, which is the number of $deg(v_j)$ of edges which contain v_j. In a directed graph, we define the *out degree* of v_j to be the number $deg^+(v_j)$ of v_j as to be the number of edges which start at v_j. In consequence, the probability to moving to a vertex v_i in a directed graph is $\frac{1}{deg^+(v_j)}$.

3.3. Learning and Classification

After the embedding of an APK is generated, the detection process is conducted to determine whether a given application is malicious or not. Therefore, our model gets the graph embedding and learn patterns that are indicative of maliciousness. Afterwards, the model produces the estimated label for a given application. Figure 3 shows the architecture of

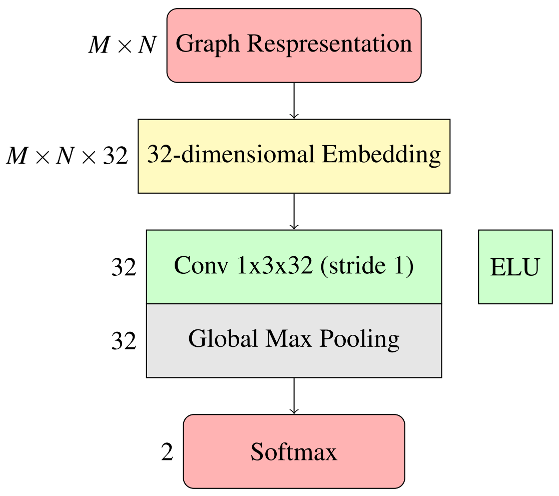


Figure 3. Convolutional neural network for Android malware detection.

the convolutional neural network for malware detection. The architecture comprises the following layers:

Input layer. The network takes as input a set of pairs x^i, y^i, where x^i is an application and y^i is its corresponding label. The label associated to benign applications is 0. Otherwise, $y^i = 1$. Each application is represented as a sequence of random walks, $x^i \in \mathbb{Z}^{M,N}$, where M denotes the number of random walks and N denotes the length of each walk. As the network cannot be fed with text, each function name is converted to a one-hot vector. A one-hot encoded vector is a vector of zeros except for the element at the index representing the corresponding function in the vocabulary. The output of the input layer is a 3-dimensional vector $v^i \in \mathbb{Z}^{M,N,V}$, where V is the size of the vocabulary, i.e. the number of distinct function calls.

Embedding layer. As one-hot vectors cannot encode semantic information about similar functions, each function call is represented as a distributed representation of size E, i.e. a low-dimensional vector of real values, where each value captures a dimension of the function meaning. Thus, the output of the embedding layer is a 3-dimensional vector $e^i \in \mathbb{Z}^{M,N,E}$, where E represents the embedding size.

Convolutional layer. The convolutional layer consists of 32 filters of size $[1,K,E]$, where K determines how many nodes or function calls the filter comprises and E is the embedding size. In our settings, K is set to 3. As a result, filters are applied to subsequences of 3 function calls.

Global max-pooling layer. The global max-pooling is applied to extract the maximum activation of each of the feature map activations from all random walks. In consequence, its output is a feature vector of size $F = 32$.

Softmax layer. The resulting feature vector is passed as input to a fully-connected softmax layer whose output is the probability of the application of being malicious.

4. Evaluation

4.1. Dataset

Our approach has been evaluated on a dataset of 3871 Android APKs, obtained from Androzoo [2]. Androzoo is a collection of Android applications collected from various sources including Google's App Store, AppChina, Anzhi App Market, among others. Each application has been sent to VirusTotal [6], a website that aggregates over 60 antivirus products from renown vendors including McAfee, Symantec, Avast, Invencea or Kaspersky, to verify whether or not the APKs are malicious. APKs used in our experiments have less than 2 Mb in size and have. APKs detected as malicious by more than 3 AV vendors are labeled as malware. On the other hand, benign APKs are those that are not detected by any AV vendor. There are a total of 1873 and 1998 benign and malicious APKs, respectively. Figures 4 and 5 present the distribution of nodes and edges in our dataset, respectively. It can be observed that in average, the number of nodes and edges of malicious applications is almost twice as bigger as those of benign applications. In particular, the average graph size of benign applications is 342.52 while the average size of malicious applications is 646.07.

[6]https://www.virustotal.com

4.2. Experimentation

To determine the best parameters for our network we split the training data into two sets: (1) the training set and (2) the validation set. Models were trained using the training set and evaluated on the validation set. Table 1 shows the performance of the network on the validation set for different values of the embedding size. Values greater than 32 did not increase the accuracy and highly increased the training time. For this reason, we decided to use $E = 32$ in our final setup.

Embedding size (E)	Validation accuracy
4	0.9199
8	0.9354
16	0.9328
32	0.9419

Table 1. Validation accuracy for different embedding size values.

To estimate the generalization performance of our approach we used K-fold cross validation, where K = 10. Table 2 presents the detection results achieved by DREBIN on our dataset, which includes the F1-score and accuracy for each experiment. As it can be observed, our method achieves higher accuracy and F1-score than DREBIN, even that it only relies on the function call graph representation of the applications. On the other hand, DREBIN extracts hand-crafted features related to the permissions, network addresses, restricted API calls, etc. Unfortunately, the process that extracts the call graph is computationally intensive and took 11948 seconds to extract all call graphs of the dataset, in comparison with the 7805 seconds needed by DREBIN to extract the features. This is because DREBIN only extracts features from the manifest and the disassembled dex code of the application. Figure 6 presents the overall running times for our dataset, divided by label. You can observe that the median average extraction time for benign and malicious applications is 2.76 and 4.67 seconds, respectively. This is because in average, the call graphs of the malicious applications in our dataset are almost as twice as big as the call graph size of benign applications. Cf. Figures 4 and 5.

Method	Accuracy	F1-score
DREBIN	0.9285	0.9280
Our method	0.9406	0.9406

Table 2. The average F1-score and accuracy of DREBIN and our method.

Additionally, Table 3 presents the error matrix, also known as confusion matrix. There are only 84 false positives and 146 false negatives. However, the number of malicious applications misclassified is almost twice the number of benign samples incorrectly classified. This might be because malicious applications usually try to hide its malicious behavior and appear as being legitimate applications, making harder its detection.

5. Conclusions

In this paper we present a novel malware detection framework that learns features from the embedded representation of Android's applications function call graph. This feature

		Actual class	
		benign	malicious
Predicted class	benign	1789	84
	malicious	146	1852

Table 3. Confusion matrix.

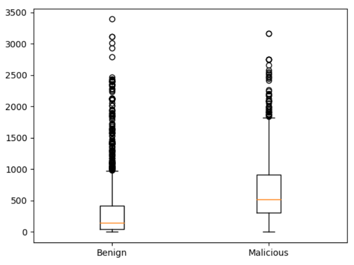


Figure 4. Application size

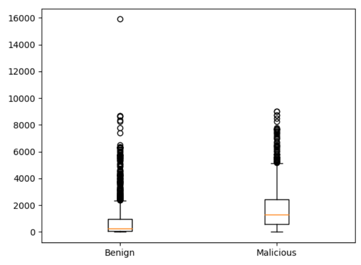


Figure 5. Edges per application

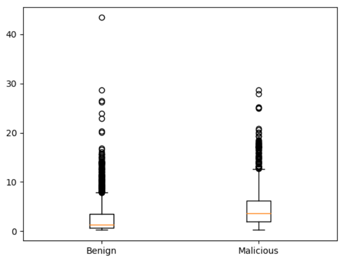


Figure 6. CFG extraction time

learning capability is provided by a convolutional neural network model whose filters are able to detect discriminant subsequences of API function calls. The proposed approach exhibits strong classification performance compared to DREBIN. This is because the CNN model do not relies on whether or not a particular function has been invoked, but instead, is able to focus on discriminative subsequences of function calls.

As far as we know, it is the first approach to apply deep learning to detect malware given the function call graph of Android applications. The main idea behind the embedding is to reduce the dimensionality of the input call graphs to a matrix of a fixed size, and subsequently, be able to automatically extract features with a CNN model. The proposed solution has two major advantages with respect traditional graph-based methods. First, it does not rely on expertise knowledge of the domain as features are extracted by the convolutional layers. Second, feature extraction and classification is performed by the same algorithm, a convolutional neural network and thus, classification is not performed by computing expensive graph similarity or signature-based algorithms.

5.1. Future Work

A future direction of research is to abstract API calls to either the package name of the call (e.g., java.lang) or its source (e.g., java, android, google). This might provide resilience to API changes in the Android framework. A second line of research is to explore new neural network architectures to perform classification. A third line of research is to analyze the inter-component communication graph (ICCG) to describe the internal control flows and inter-component communications of APKs.

References

[1] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in android. In Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao, editors, *Security and Privacy in Communication Networks*, pages 86–103, Cham, 2013. Springer International Publishing.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS'14*. The Internet Society, 2014.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[5] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[6] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[7] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: Installing applications on an android smartphone. In Jim Blyth, Sven Dietrich, and L. Jean Camp, editors, *Financial Cryptography and Data Security*, pages 68–79, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[8] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *CoRR*, abs/1612.04433, 2016.

[9] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play?: A large-scale empirical study. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 231–242, New York, NY, USA, 2018. ACM.

[10] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: Deep learning in android malware detection. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 371–372, New York, NY, USA, 2014. ACM.

[11] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual API dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1105–1116, New York, NY, USA, 2014. ACM.

10.3 Research Stays Abroad

During the elaboration of the thesis, the candidate did two research stays:

- From September 30, 2019 to December 15, 2019 at the Carnegie Mellon University (CMU), United States of America, supervised by Dr. Ruben Martins and Dr. Matt Fredrikson.
- From January 19, 2020 to April 24, 2020 at the Artificial and Natural Intelligence Toulouse Institute, University of Toulouse, France, supervised by Dr. Joao Marques-Silva.

10.4 Participation in Projects

The author of this thesis participated in the following research projects during the doctoral program:

- Razonamiento, satisfacción y optimización: argumentación y problemas. Founded by the Spanish Ministry of Economy, Industry and Competitiveness. Project number: TIN2015-71799-C2-2-P.
- Sistemas de inferencia para información inconsistente: Análisis Argumentativo. Founded by the Spanish Ministry of Sciences, Innovation and Universities. Project Number: PID2019-111544GB-C22 (pending project number confirmation)

Chapter 11

Conclusions

The aim of this chapter is to summarize and highlight the main contributions of the dissertation. The thesis outcome leaves scope for many open lines of research and future developments, further explored in the last section of this chapter.

11.1 Contributions

The objective of this thesis was to investigate the limitations of n-gram methods and develop of an efficient featureless alternative to manual feature engineering for the task of malware detection and classification. To this end, this dissertation presents a set of experiments designed to address the following research questions:

1. In which scenarios is end-to-end learning feasible?

2. Which deep learning architectures are most appropriate?

3. How can we fuse multiple modalities of information in a end-to-end learning architecture?

In the following lines, additional discussion around the aforementioned research questions are provided. To goal of this section is to understand the current trends and recent findings with regard to each of the research questions above and to describe how the experiments presented in this thesis contribute to the state-of-the-art.

11.1.1 In Which Scenarios Is End-to-End Learning Feasible?

Part of the dissertation has centered around the investigation of featureless alternatives to manual feature engineering through deep learning. In this context, deep learning replaces the feature engineering process by an underlying system, which typically consists of a neural network with multiple layers, that performs both feature learning and classification. With deep learning one can start with raw data as features will be automatically created by the network during the training procedure. However, the success and consolidation of end-to-end learning approaches would not have been possible without the confluence of three recent developments:

- The first development is the increase in labeled feeds of data, e.g. malware, meaning that, for the first time, labeled malware started to become available not only to the security community but also to the research community. The size of these feeds ranges from limited high-quality samples, like the ones provided by Microsoft [66] for the Big Data Innovators Gathering Anti-Malware Prediction Challenge, to huge volumes of malware, such as theZoo [76] or VirusShare [72]. For reproducibility purposes, the performance of all the methods presented in this dissertation has been assessed on the dataset provided by Microsoft [66], a labeled high-quality public benchmark for malware research.

- The second development is that computational power has increased rapidly and at the same time has become cheaper and closer to the budget of most researchers. Consequently, it allowed researchers to speed-up the iterative training process and fit larger and more complex models to the ever increasing data. For instance, research GPUs and TPUs allow faster training of complex models, in parallel if desired, given large input sequences consisting of millions of timesteps like the byte sequences representing malware's binary content. A few years ago, the idea of training on such input data was inconceivable. In fact, it was not until 2018 that the first approach to train a model using as input the whole byte sequence was presented [63]. Due to memory limitations, previous research focused on simplifying the byte sequences into some kind of compressed representation before training the model [31, 26].

- The third development is that the machine learning field has evolved at an increased pace during the last decade, achieving breakthrough success in terms of accuracy and stability on a wide range of tasks, such as computer vision, speech recognition and natural language processing. In particular, this success would not have occurred without recent advances in gradient descent optimization [20, 77, 46], function activations [58, 15], network architectures [52, 36, 14, 6, 73], the availability of public frameworks to speed up research such as TensorFlow [1] and PyTorch [62], among other relevant advances.

11.1.2 Which Deep Learning Architectures Are Most Appropriate?

This section discusses the principal neural network types and their appropriateness in the context of malware classification.

Feed-forward neural network. Fully-connected layers are an essential component of feed-forward and convolutional neural networks. In a fully-connected layer, all nodes in one layer connect to all nodes in the next layer. Feed-forward networks are the simplest type of neural networks devised and are composed by one or more fully-connected layers. In a feed-forward network, information moves in only one direction, from the input nodes to the output nodes, passing through one or more fully-connected layers. Treating each byte or opcode as a unit in a sequence, we are dealing with a sequence classification problem in the order of millions or thousands of time steps, respectively. Subsequently, feed-forward networks are inefficient and inappropriate for sequence classification problems as they require a huge number of connections and network parameters.

Convolutional neural network (CNN). Convolutional neural networks have several types of layers:

- Convolutional layer. In a convolutional layer one or more "filters" pass over an N-array such as an image or a sequence of text, scanning a few pixels or words at a time and creating one or more feature maps;
- Pooling layer. The pooling layer reduces the amount of information in each feature map obtained in a convolutional layer while maintaining the most important information. Common pooling operations are max-pooling or average-pooling. A convolutional neural network usually consists of several rounds of convolutional and pooling layers.
- Fully-connected layer. In a convolutional neural network, one or more fully-connected layers are stacked together in a way that takes as input the end result of the convolution and pooling layers (the features extracted by the convolution/pooling process) and reaches a classification decision (the last fully-connected layer, also known as output layer.).

Convolutional neural networks were originally designed for computer vision tasks but have proven highly useful for natural language processing tasks as well [45, 41] The main benefit of convolutional neural networks for malware classification is their ability to process huge sequences of text and detect discriminant patterns independently of their position in the text. This is achieved by adding a global max pooling operation (other pooling operations might produce similar results, e.g. global average pooling) at the end of the convolutional layers to retrieve the maximum value of each feature map [24].

Recurrent neural network (RNN). Vanilla recurrent neural networks use an architecture similar to the traditional feed-forward network but they introduce the concept of memory in the form of a different type of link. Unlike feed-forward networks, the outputs of some layers are fed back into the inputs of the previous layer. This allows for the analysis of sequential data, which feed-forward network are incapable of performing. Recurrent neural networks are able to recognize and take advantage of the time-related context in a time series sequence. A popular recurrent neural network is the so-called Long-Short Term Memory Network (LSTM), which is a type of RNN capable of learning long-term relationships. However, the suitability of RNNs for the task of malware detection and classification is questionable for the following reasons:

- By treating each byte as a unit in a sequence, the size of the resulting byte sequences could consist of several million time steps, making it among the most challenging sequence classification problems. Diversely, by treating each opcode as a unit in a sequence, the size of the resulting opcode sequences would consist of thousands of time steps. In vanilla RNNs, the more timesteps there are, the higher is the change that the backpropagation gradients explode or vanish. That is, the long-term information has to sequentially travel through all cells before reaching the present cell and, thus, the information can be easily corrupted. To overcome this problem, Long-Short Term Memory Networks came to the rescue. However, binary files exhibit various levels of spatial correlation. Adjacent machine instructions tend to be correlated spatially, but, due to jumps and function calls, this correlation might not always hold, as they transfer the control of the program into other addresses in memory and the execution continues from there. Consequently, these discontinuities are maintained on the binary file and in its hexadecimal representation. Therefore, the information provided in a byte or opcode sequence does not move only to one direction but might jump backward and forward, and contain cycles or loops and thus, the learning ability of recurrent neural networks is compromised.

Therefore, during this dissertation, and through extensive experimentation, state-of-the-art results have been achieved mainly with convolutional neural network architectures [24, 31, 26, 32]. The methods presented in this thesis can be grouped in two categories depending on the raw data taken as input:

- Byte-based approaches [31, 26, 32]. Byte-based approaches take as input the raw byte sequences extracted from the hexadecimal representation of malware's binary content.
  - Section 4.3.3 presents a file agnostic deep learning approach [31] to categorize malware based on its structural entropy representation.
  - Section 4.3.4 introduces an approach [26] to learn a hidden representation of the malware's binary content based on denoising autoencoders.
  - Section 6 proposes an approach [32] to efficiently group malicious software into families based on a set of discriminant patterns extracted from malware's representation as gray-scale images.
- Opcode-based approaches [24, 27]. Opcode-based approaches take as input the raw opcode sequences extracted from the assembly language source code of malware.
  - Section 4.3.1 presents an approach [24] to extract N-gram like features from malware's machine instructions.
  - Section 4.3.2 proposes an architecture [27] to deal with the hierarchical information in a Portable Executable file's assembly language source code.

However, the task of malware detection and classification is characterized as multimodal as it includes multiple modalities of information. By only taking as input the raw bytes or opcodes sequence a great deal of useful information for classification is overlooked such as structural information of the Portable Executable (PE) file, the import address table (IAT) which is used as a lookup table when the application is calling a function from a different module, etc. In fact, multimodal methods based on the combination of various hand-crafted features [3, 79] remained unbeaten in terms of classification performance and have been the way to go for detecting and classifying malware. This leads to the following research question: *How can we fuse multiple modalities of information in a end-to-end learning architecture?*

11.1.3 How Can We Fuse Multiple Modalities Of Information In A End-to-End Learning Architecture?

State-of-the-art multimodal approaches [3, 79] create a joint representation of the unimodal features extracted separately from multiple modalities by concatenating the unimodal feature vectors. Afterwards, the resulting feature vector is passed as input to a classification algorithm (gradient boosting classifiers and Extra-Trees) that generates the corresponding classification output. However, the end-to-end learning approaches presented in this dissertation take as input raw data and a direct concatenation of the raw byte and opcode sequences is unreasonable. To this end, we presented Orthrus [29] and HYDRA [28], two approaches that use an intermediate fusion strategy to construct a shared representation by merging the intermediate features extracted from the different modalities in a end-to-end architecture. Orthus[29] combines two modalities of data: (1) the byte sequence representing the malware's binary content, and (2) the assembly language instructions extracted from the assembly language source code of malware. Differently, HYDRA's [28] wide and deep architecture consists of both hand-engineered (API function calls) and end-to-end components (byte and opcode sequences) to combine the benefits of feature engineering and deep learning. Regardless of the architectural details, per-modality pretraining and multimodal dropout have been critical for the success of the aforementioned approaches. On the one hand, per-modality pretraining avoids overfitting one subset of features belonging to one modality and underfitting the features belonging to the others. On the other hand, multimodal dropout prevents the co-adaptation of the sub-networks to a specific feature type or modality of data. Furthermore, the European Patent Application EESR EP19382656, presents a computer-implemented method, system and computer program for identifying a malicious file that performs a fuzzy inference procedure based on fuzzy rules using as input a set of features and the preliminary classification output produced by the aforementioned approaches [31, 32, 24].

11.1.4 Future Work

Despite the increase in the research and deployment of anti-malware engines powered by machine learning, there is little research about the vulnerability of these models to adversarial examples. This dissertation provides an evaluation of machine learning approaches against common metamorphic techniques such as the dead code insertion technique, the subroutine reordering technique, etc (See Section 9.4.1). However, malware authors also employ polymorphic techniques such as encryption and packing to bypass detection. Consequently, a future line of research might be the investigation of the robustness of machine learning models against the aforementioned polymorphic techniques. In addition, malware evolves rapidly to exploit new attack surfaces but models are built through training on older malware samples and thus, it is hard for the machine learning models to generalize to future, previously-unseen behaviors resulting in new malware evading detection more often than desired. This phenomenon is known as concept drift, that is, the underlying relationships in the data change over time. As a result, it is of crucial importance to identify when a model shows signs of degradation whereby it fails to recognize new malware in order to build sustainable models for malware classification.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware dynamic analysis evasion techniques: A survey. *CoRR*, abs/1811.01190, 2018.

[3] Mansour Ahmadi, Giorgio Giacinto, Dmitry Ulyanov, Stanislav Semenov, and Mikhail Trofimov. Novel feature extraction, selection and fusion for effective malware family classification. *CoRR*, abs/1511.04317, 2015.

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

[5] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*. The Internet Society, 2014.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[7] Tadas Baltrusaitis, Chaitanya Ahuja, and Louis-Philippe Morency. Multimodal machine learning: A survey and taxonomy. *IEEE Trans. Pattern Anal. Mach. Intell.*, 41(2):423–443, February 2019.

[8] Richard E Bellman. *Adaptive control processes: a guided tour*. Princeton university press, 2015.

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

[10] Daniel Bilar. Statistical structures: Fingerprinting malware for classification and analysis. *ICGeS'07: Proceedings of the 3rd International Conference on Global E-Security*, 01 2006.

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

[12] Lei Chen. *Curse of Dimensionality*, pages 545–546. Springer US, Boston, MA, 2009.

[13] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[14] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[16] Symantec Corporation. Symantec 2018 internet security threat report. Technical report, Symantec Corporation, 2018.

[17] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries. *CoRR*, abs/1901.03583, 2019.

[18] Hui Ding, Goce Trajcevski, Peter Scheuermann, Xiaoyue Wang, and Eamonn Keogh. Querying and mining of time series data: Experimental comparison of representations and distance measures. *Proc. VLDB Endow.*, 1(2):1542–1552, August 2008.

[19] Jake Drew, Michael Hahsler, and Tyler Moore. Polymorphic malware detection using sequence classification methods and ensembles. *EURASIP Journal on Information Security*, 2017(1):2, 2017.

[20] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159, July 2011.

[21] J. B. Fraley and J. Cannady. The promise of machine learning in cybersecurity. In *SoutheastCon 2017*, pages 1–6, March 2017.

[22] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, March 2014.

[23] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Mach. Learn.*, 63(1):3–42, April 2006.

[24] Daniel Gibert, Javier Béjar, Carles Mateu, Jordi Planes, Daniel Solis, and Ramon Vicens. Convolutional neural networks for classification of malware assembly code. In *Recent Advances in Artificial Intelligence Research and Development - Proceedings of the 20th International Conference of the Catalan Association for Artificial Intelligence, Deltebre, Terres de l'Ebre, Spain, October 25-27, 2017*, pages 221–226, 2017.

[25] Daniel GIBERT, Alba LAMAS, Ruben MARTINS, and Carles MATEU. An android malware detection framework using graph embeddings and convolutional neural networks. In *Artificial Intelligence Research and Development: Proceedings of the 22nd International Conference of the Catalan Association for Artificial Intelligence*, volume 319, page 45. IOS Press, 2019.

[26] Daniel Gibert, Carles Mateu, and Jordi Planes. An end-to-end deep learning architecture for classification of malware's binary content. In Věra Kůrková, Yannis Manolopoulos, Barbara Hammer, Lazaros Iliadis, and Ilias Maglogiannis, editors, *Artificial Neural Networks and Machine Learning – ICANN 2018*, pages 383–391, Cham, 2018. Springer International Publishing.

[27] Daniel Gibert, Carles Mateu, and Jordi Planes. A hierarchical convolutional neural network for malware classification. In *International Joint Conference on Neural Networks (IJCNN) 2019 Budapest, Hungary, July 14-19, 2019*, page 8, 2019.

[28] Daniel Gibert, Carles Mateu, and Jordi Planes. Hydra: A multimodal deep learning framework for malware classification. *Computers & Security*, 95:101873, 2020.

[29] Daniel Gibert, Carles Mateu, and Jordi Planes. Orthrus: A bimodal learning architecture for malware classification. In *IEEE World Congress on Computational Intelligence (WCCI), International Joint Conference on Neural Networks (IJCNN) 2020 Glassgow, UK, July 19-24, 2020*, page 8, 2020.

[30] Daniel Gibert, Carles Mateu, and Jordi Planes. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526, 2020.

[31] Daniel Gibert, Carles Mateu, Jordi Planes, and Ramon Vicens. Classification of malware by using structural entropy on convolutional neural networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 7759–7764, 2018.

[32] Daniel Gibert, Carles Mateu, Jordi Planes, and Ramon Vicens. Using convolutional neural networks for classification of malware represented as images. *Journal of Computer Virology and Hacking Techniques*, Aug 2018.

[33] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An approach to evaluating interpretability of machine learning. *CoRR*, abs/1806.00069, 2018.

[34] Josif Grabocka, Nicolas Schilling, Martin Wistuba, and Lars Schmidt-Thieme. Learning time-series shapelets. In *KDD'14*, pages 392–401. ACM, 2014.

[35] X. Guo, Y. Yin, C. Dong, G. Yang, and G. Zhou. On the class imbalance problem. In *2008 Fourth International Conference on Natural Computation*, volume 4, pages 192–201, Oct 2008.

[36] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[37] Xin Hu, Jiyong Jang, Ting Wang, Z. Ashraf, Marc Ph. Stoecklin, and Dhilung Kirat. Scalable malware classification with multifaceted content features and threat intelligence. *IBM Journal of Research and Development*, 60(4):6, 2016.

[38] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. Mutantx-s: Scalable malware clustering based on static features. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 187–198, San Jose, CA, 2013. USENIX.

[39] Alex Huang, Abdullah Al-Dujaili, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. *CoRR*, abs/1801.02950, 2018.

[40] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I.P. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, AISec '11, pages 43–58, New York, NY, USA, 2011. ACM.

[41] Alon Jacovi, Oren Sar Shalom, and Yoav Goldberg. Understanding convolutional neural networks for text classification. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 56–65, Brussels, Belgium, November 2018. Association for Computational Linguistics.

[42] Nathalie Japkowicz and Shaju Stephen. The class imbalance problem: A systematic study. *Intelligent Data Analysis*, pages 429–449, 2002.

[43] Roberto Jordaney, Kumar Sharad, Santanu K. Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 625–642, Vancouver, BC, August 2017. USENIX Association.

[44] Alex Kantchelian, Sadia Afroz, Ling Huang, Aylin Caliskan Islam, Brad Miller, Michael Carl Tschantz, Rachel Greenstadt, Anthony D. Joseph, and J. D. Tygar. Approaches to adversarial drift. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 99–110, New York, NY, USA, 2013. ACM.

[45] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.

[46] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[47] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[48] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *26th European Signal Processing Conference, EUSIPCO 2018, Roma, Italy, September 3-7, 2018*, pages 533–537. IEEE, 2018.

[49] Marek Krčál, Ondřej Švec, Martin Bálek, and Otakar Jašek. Deep convolutional malware classifiers can learn from raw executables and labels only, 2018.

[50] Nir Kshetri. *The global cybercrime industry: Economic, institutional and strategic perspectives.* Springer, 01 2010.

[51] Quan Le, Oisín Boydell, Brian Mac Namee, and Mark Scanlon. Deep learning at the shallow end: Malware classification for non-domain experts. *Digital Investigation*, 26:S118 – S126, 2018.

[52] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, December 1989.

[53] M. M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, *Software Process Technology*, pages 108–124, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[54] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5(2):40–45, March 2007.

[55] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. Towards adversarial malware detection: Lessons learned from pdf-based attacks. *ACM Comput. Surv.*, 52(4):78:1–78:36, August 2019.

[56] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam

Doupé, and Gail Joon Ahn. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 301–308, New York, NY, USA, 2017. ACM.

[57] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, 2007.

[58] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[59] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: Visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, VizSec '11, pages 4:1–4:7, New York, NY, USA, 2011. ACM.

[60] P. OKane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. *IEEE Security Privacy*, 9(5):41–47, Sept 2011.

[61] CSIS Commission on Cybersecurity for the 44th Presidency, J. Langevin, J.A. Lewis, Center for Strategic, and D.C.) International Studies (Washington. *A Human Capital Crisis in Cybersecurity: Technical Proficiency Matters : a White Paper of the CSIS Commission on Cybersecurity for the 44th Presidency.* Center for Strategic and International Studies, 2010.

[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[63] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. Malware detection by eating a whole EXE. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018.*, pages 268–276, 2018.

[64] Edward Raff and Charles Nicholas. Hash-grams: Faster n-gram features for classification and malware detection. In *Proceedings of the ACM Symposium on Document Engineering 2018*, DocEng '18, New York, NY, USA, 2018. Association for Computing Machinery.

[65] Edward Raff, Richard Zak, Russell Cox, Jared Sylvester, Paul Yacci, Rebecca Ward, Anna Tracy, Mark McLean, and Charles Nicholas. An investigation of

byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, 14(1):1–20, Feb 2018.

[66] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *CoRR*, abs/1802.10135, 2018.

[67] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64 – 82, 2013. Data Mining for Information Security.

[68] S. Shirataki and S. Yamaguchi. A study on interpretability of decision of machine learning. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 4830–4831, Dec 2017.

[69] Ivan Sorokin. Comparing files using structural entropy. *Journal in Computer Virology*, 7(4):259, Jun 2011.

[70] Octavian Suciu, Scott E. Coull, and Jeffrey Johns. Exploring adversarial examples in malware detection. *CoRR*, abs/1810.08280, 2018.

[71] Jorge R. Vergara and Pablo A. Estévez. A review of feature selection methods based on mutual information. *Neural Computing and Applications*, 24(1):175–186, 2014.

[72] VirusShare. Vxshare, 2011.

[73] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander J. Smola, and Eduard H. Hovy. Hierarchical attention networks for document classification. In *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, pages 1480–1489, 2016.

[74] Lexiang Ye and Eamonn Keogh. Time series shapelets: A new primitive for data mining. In *KDD'09*, pages 947–956. ACM, 2009.

[75] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, Nov 2010.

[76] Lahad Ludad Yuval Nativ and 5fingers. The zoo, 2015.

[77] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[78] FuYong Zhang and Tiezhu Zhao. Malware detection and classification based on n-grams attribute similarity. In *2017 IEEE International Conference on Computational Science and Engineering, CSE 2017, and IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2017, Guangzhou, China, July 21-24, 2017, Volume 1*, pages 793–796. IEEE Computer Society, 2017.

[79] Y. Zhang, Q. Huang, X. Ma, Z. Yang, and J. Jiang. Using multi-features and ensemble learning method for imbalanced malware classification. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 965–973, Aug 2016.