University of Salerno

Department of Computer Science

Ph.D. in Computer Science
Curriculum Internet of Things and Smart Technologies
XXXV Cycle

PH.D. THESIS

Intelligent Technologies for Threat Detection in Networks Security Scenarios

Antonio ROBUSTELLI
Antonio Robustelli

SUPERVISOR:	**Prof. Francesco PALMIERI**	Francesco Palmieri
CO-SUPERVISOR:	**Prof. Gianni D'ANGELO**	Gianni D'Angelo
PHD PROGRAM DIRECTOR:	**Prof. Andrea DE LUCIA**	Andrea De Lucia

Academic Year 2022/2023

I propose to consider the question, 'Can machines think?'
I shall replace the question by another, which is closely related to it
and is expressed in relatively unambiguous words.

- Alan Turing -

Stay Hungry, Stay Foolish!

- Steve Jobs -

Code never lies,
comments sometimes do.

- Ron Jeffries -

I solemnly swear,
that I am up to no good!

- Harry Potter -

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my supervisors, Professors Francesco Palmieri and Gianni D'Angelo: they convincingly encouraged me to be professional and do the right thing even when the road got tough. Without their persistent help, the goal of this project would not have been achieved.

I want to pay my special regards to Chiara for her help during my PhD. She kept me going on, and this work would not have been possible without her lovely support.

Finally, I wish to acknowledge the support and great love of my family, my colleagues (Eslam and Leila), my friends (Alfonso, Cinzia, Gaia, Giusy, and Maria Giovanna), and all the people that have always believed in me all the way, and whose assistance was a milestone in the completion of this project.

ABSTRACT

The advent of the Internet of Things (IoT), with the consequent changes in network architectures and communication dynamics, has strongly conditioned the security market by radically shifting traditional perceptions of the current Internet toward an integrated vision of smart interconnected objects. However, due to their provided features and popularity, such devices have become one of the main targets for attackers who, exploiting several systems-related vulnerabilities and well-engineered applications, are able to conduct different hostile activities. For this reason, also thanks to the great success of Machine Learning (ML) and Deep Learning (DL) based techniques in the last decade, many innovative solutions have been proposed in order to counteract the exponential and yearly growth of malware applications. However, since the related detection models should provide an adequate generalization capability, their success strongly depends on the right choice of the employed features. To this purpose, new empowered strategies are needed to spot malware threats in several network security scenarios, with particular attention to those related to IoT and Federated environments, respectively.

Therefore, this thesis focuses on the enhancement of detection solutions that, due to the presence of many vulnerable and hardware-constrained devices, are characterized by several challenges regarding security and privacy. Under this vision, Chapter 1 presents a detailed overview of the state-of-the-art by highlighting the weaknesses of the existing approaches. Next, Chapters 2, 3, and 4 focus on the empowerment and effectiveness of such solutions by employing new dynamic and static-based feature representation techniques. Also, they highlight the capabilities of DL to offer sophisticated models capable of reducing Run-time damages and involving the computation capabilities of federated environments, respectively.

On the other hand, due to the recent explosion of IoT-related malware applications and the necessity of protecting privacy, the thesis extends its focus to malware detection activities in

Federated organizations. Therefore, Chapter 5 proposes a new Markov Chains-based detector capable of improving the most famous Federated Learning (FL) based solutions. To this purpose, a dedicated privacy-preserving architecture is employed, in which the involved clients build the related detection model by indirectly sharing the analyzed applications. Finally, Chapter 6 presents the Conclusions about the reported contributions by highlighting possible and relevant future research directions.

CONTENTS

1 INTRODUCTION

1.1 OVERVIEW OF THE THESIS

The advent of the Internet of Things (IoT), with the consequent changes in network architectures and communication dynamics, has strongly conditioned the security market by radically shifting the traditional perception of the Internet toward an integrated vision of smart interconnected objects. However, due to their worldwide popularity and provided features, such devices have become one of the main targets for attackers who, exploiting several systems-related vulnerabilities and well-engineered applications, are able to conduct different hostile activities and cause many security-related issues. For this reason, also thanks to the great success of Machine Learning (ML) and Deep Learning (DL) based techniques, many innovative solutions have been proposed in order to counteract one of the most famous network security issues, namely the exponential and yearly growth of malware applications. For instance, Aonzo et al. [6] presented BAdDroIds, a mobile application that leverages static features for detecting malware, such as the required permissions and API methods extracted from the DEX file. While Abderrahmane et al. [1] employed a Convolutional Neural Network (CNN) to classify malware applications using a matrix representation of system calls.

However, malware static analysis-based approaches can be strongly affected by obfuscation tools. Also, they become ineffective against most existing malware capable of evading pattern-matching detection mechanisms, namely metamorphic and polymorphic ones [40]. Hence, several dynamic analysis-based approaches have been adopted to overcome these issues and capture the behavior of analyzed applications [40]. However, such solutions cannot perform classification tasks at Run-time because they consider features derived by the post-analysis report, which

describes applications that have already been executed and thus that reached their malicious goals.

On the other hand, due to the recent explosion of IoT-related malware applications, involved users and organizations became increasingly reluctant to share their data. For this reason, one of the most popular and recent options is associated with Federated Learning (FL)-based solutions [71], in which each entity trains an individual model using only its data and, in order to create a global and shared model, sends the derived model parameters to a central server [120]. In this direction, many FL-based solutions have been proposed and adopted in several application domains, such as Healthcare applications [20], Failure prognosis [29], and Network traffic detection and classification [125, 127].

However, as highlighted in many notable literature studies [18, 46, 57, 83], non-Independent and Identically Distributed (non-IID) data often adversely affects FL-based solutions regarding the required training time, convergence, learning processes, and classification results. Also, such strategies are often strongly influenced by the configuration of some additional hyperparameters (e.g. threshold values) that might limit their applicability [49, 78, 125].

Finally, since the related detection models should also have an adequate generalization capability, their success strongly depends on the right choice of the employed features. For this reason, as highlighted below by the state-of-the-art, new empowered detection strategies are needed to effectively spot malware threats in several network security scenarios, with particular attention to those related to IoT and Federated environments, respectively.

1.2 THE STATE-OF-THE-ART ON MALWARE DETECTORS

Nowadays, many detection frameworks and methodologies based on static and dynamic analysis have been proposed [17, 65, 66, 105] to counter the continuous diffusion of malware. To this end, static-based approaches can extract useful features (e.g. signatures) without executing the analyzed applications. Indeed, to accomplish this, such solutions disassemble the applications by performing several reverse engineering operations. For example,

Zhang et al. [122] proposed a semantic-based approach using dependency graphs created from the application code. Instead, Onwuzurike et al. [80] presented MaMaDROID, a malware detection system that finds the sequences of methods potentially invoked and maps them to their corresponding packages and classes.

However, malware static analysis-based approaches can be strongly affected by obfuscation tools. Also, they become ineffective against most existing malware capable of evading pattern-matching detection mechanisms, namely metamorphic and polymorphic ones [40]. Therefore, in order to overcome these issues, several dynamic-based solutions have been proposed to analyze the dynamic behavior of an application and consider several related aspects, such as parameter monitoring and API-Calls tracing [32]. For instance, Kolosnjaji et al. [51] leveraged Deep Neural Networks (DNNs) to analyze the sequence of system calls extracted at Run-time. To accomplish this, they combined convolutional and recurrent layers by obtaining an average accuracy of 89.0% on 10 Android malware categories. Similarly, Abderrahmane et al. [1] employed a CNN to perform malware analysis in the presence of an unbalanced dataset. More precisely, they achieved an accuracy of 93.3% by classifying a matrix representation of system calls. Next, in 2020, D'Angelo et al. [22] improved this representation technique by considering the API-Calls sequences arranged as corresponding matrices. More precisely, they demonstrated the effectiveness of the proposed approach, called API-Images, by obtaining an average accuracy of 95% in performing binary classification tasks through Sparse Autoencoders (SAEs).

Therefore, to also provide detection models with an adequate generalization capability, I explore the effectiveness of dynamic-based approaches in classifying several Android malware families. To this purpose, I use a features representation technique, called API-Images [22], to represent the dynamic behavior of applications as corresponding images. Then, I employ the derived API-Images to propose different kinds of Deep Learning (DL)-based classifiers.

However, dynamic-based approaches cannot perform classification tasks at Run-time because they consider features derived

by the post-analysis report, which describes applications that have already been executed and thus have fully shown their behavior. On the other hand, thanks to the great success of DL in many research fields, several solutions have been proposed to face the related classification tasks as corresponding Video-Classification activities. For instance, Perez et al. [85] proposed a novel method to distinguish porn and no-porn videos based on concatenated static and motion features. More precisely, they validated their approach by considering two famous datasets, namely Pornography-800 and Pornography-2k [9]. The achieved results, carried out by a CNN, have shown an accuracy of 96.4% and 97.9%, respectively. Instead, Xu et al. [119] investigated a new approach to detect violent videos using two P3D-LSTM neural networks. To accomplish this, they first considered static frames and optical flows separately. Then, they classified such videos by combining the extracted features through a late fusion strategy. The achieved results, derived from different public and self-built datasets, have proven the effectiveness of the presented approach with an average accuracy of 97.97%. Instead, Santacroce et al. [93] proposed a Time Distributed CNN based on the executable code of applications. Hence, they first transformed the related bytecodes into corresponding streams through a fixed-length time windows mechanism. Then, they proved the effectiveness of the following neural network by achieving an average accuracy between 98.74% and 99.36%. However, since the following approach works on a stream representation of the bytecode, it can be adversely affected by obfuscation techniques and polymorphic malware.

Therefore, in order to support Run-time malware classification activities, I employ an Autoencoders-based approach to consider streams of API-Images and then face malware classification tasks as corresponding Video-Classification activities. Hence, unlike the most famous state-of-the-art solutions, the presented approach might be involved in Run-time monitoring processes capable of detecting malicious applications in the shortest possible time.

Although dynamic-based solutions can trace the behavior of applications, static-based ones are often preferable because they can acquire relevant signatures and features without executing

the source code. For this reason, most notable DL-based malware detectors leverage static information, such as permissions, Java methods, app components, and intent filters. For instance, Vinayakumar et al. [110] employed several Long-Short Term Memory (LSTM) network configurations to classify Android applications as benign or malicious. To accomplish this, for each APK file, they considered the translated Android permissions as corresponding numerical information. However, the obtained results, characterized by a maximum accuracy of 89.7%, have proven the ineffectiveness of this approach. For this reason, Li et al. [59] compared several DNNs configurations by considering both permissions and Java code. More precisely, they achieved an average accuracy between 95% and 97% in classifying several Android malware families. Similarly, Xie et al. [118] proposed RepassDroid, a tool capable of distinguishing malicious applications from benign ones. More precisely, they first explored the effectiveness of the most famous ML-based approaches. Next, since the Random Forest (RF) algorithm had achieved the best results with an accuracy of 99.7%, they employed the following approach as the core module of RepassDroid. In addition, Li et al. [56] proposed a DNN-based detector by obtaining an average accuracy of 99.25%. To accomplish this, they considered other features extracted from the Manifest file, such as app components, hardware features, intent filters, and suspicious methods.

However, the discussed approaches consider Java methods traced from the source code. Consequently, they become ineffective against most existing malware capable of evading pattern-matching detection mechanisms, namely metamorphic and polymorphic ones. Therefore, to face this problem, I present a Federated CNN-based detector that leverages Android permissions and their corresponding severity levels to classify different malware families.

On the other hand, due to the recent explosion of IoT-related malware applications and the necessity of protecting privacy, involved users and organizations became increasingly reluctant to share their data. For this reason, one of the most popular and recent options is associated with Federated Learning (FL)-based solutions [71], in which each entity trains an individual model using only its data and, in order to create a global and

shared model, sends the derived model parameters to a central server [120]. In this direction, many FL-based solutions have been proposed and adopted in several application domains, such as Healthcare applications [20], Failure prognosis [29], and Network traffic detection and classification [125, 127]. Also, they have been used for Malware classification in IoT environments by considering several extracted features, FL algorithms optimization, learning models, and security schemes. Unfortunately, due to the high amount of related works published, performing a complete comparison is very difficult. For this reason, I report only some contributions that can be helpful to understand the potentiality of the proposed architecture. Also, I remand to well-known literature surveys [49, 78, 125] for more detailed information.

In 2020, Ruei-Hau et al. [41] proposed a new Android detection schema to prevent possible Poisoning attacks on a federated model. They protected the federated learning process through a Secure Multi-Party Computation (SMPC) implementation provided by OpenMined. Moreover, they evaluated the discussed detection model in centralized and decentralized scenarios by obtaining an average accuracy of 94.05% and 93.45%, respectively. In the same direction, Galvez et al. [35] presented Less is More (LiM), a Semi-supervised framework that leverages FL and static features to detect Android malware applications. More precisely, the following framework achieved an average F-Score of 95% over 50 iteration rounds and 50.000 Android applications distributed among 200 clients. In 2021, Shukla et al. [99] proposed a Robust and Active Protection with Intelligent Defense (RAPID) strategy against malicious activities based on CNNs. They have proven the effectiveness of the following malware classifier by obtaining a 94% average accuracy. Also, to mitigate possible global model-related poisoning, they introduced a server-side defence mechanism based on the euclidean distances derived from each federated model. Finally, in 2022, Rey et al. [89] and Popoola et al. [86] presented similar network flows-based approaches to detect the presence of cyberattacks in IoT domains. More precisely, the following methods have been trained by considering several learning scenarios (Supervised, Semi-Supervised, and Unsupervised), DNNs models, hyperparameters (number of clients

and rounds), and network features by achieving a 99% average accuracy.

Tab. 1.1 summarizes the discussed malware detectors by highlighting the adopted approach (Static or Dynamic), the employed model, and the achieved accuracy, respectively.

Authors	Title	Approach	Model	Accuracy
Kolosnjaji et al. [51]	Deep Learning for Classification of Malware System Call Sequence	Dynamic	DNN	89.0%
Abderrahmane et al. [1]	Android Malware Detection Based on System Calls Analysis and CNN Classification	Dynamic	CNN	93.3%
D'Angelo et al. [22]	Malware detection in mobile environments based on Autoencoders and API-images	Dynamic	CNN-SAE	95.0%
Santacroce et al. [93]	Detecting Malware Code as Video With Compressed, Time-Distributed Neural Networks	Dynamic	CNN	99.3%
Popoola et al. [86]	Federated Deep Learning for Zero-Day Botnet Attack Detection in IoT-Edge Devices	Dynamic	FL-DNN	99.0%
Rey et al. [89]	Federated learning for malware detection in IoT devices	Dynamic	FL-DNN	99.0%
Vinayakumar et al. [110]	Deep android malware detection and classification	Static	LSTM	89.7%
Li et al. [59]	Android Malware Clustering through Malicious Payload Mining	Static	DNN	96.0%
Xie et al. [118]	RepassDroid: Automatic Detection of Android Malware Based on Essential Permissions and Semantic Features of Sensitive APIs	Static	RF	99.7%
Li et al. [56]	Android Malware Detection Based on Factorization Machine	Static	DNN	99.2%
Quei-Hau et al. [41]	Privacy-Preserving Federated Learning System for Android Malware Detection Based on Edge Computing	Static	FL	94.0%
Shukla et al. [99]	On-device Malware Detection using Performance-Aware and Robust Collaborative Learning	Static	FL-CNN	94.0%

Table 1.1: An overview of the discussed malware detectors.

However, the reported FL-based approaches consider static features that, as previously remarked, are strongly affected by obfuscation techniques and polymorphic malware. Instead, traffic-based ones, since considering features directly derived from the network packets, become ineffective against traffic anonymization techniques. Also, since they cannot analyze the application-related dynamic behavior, they cannot be employed in any possible runtime-based detection or classification strategies [26].

Furthermore, as highlighted in many notable literature studies [18, 46, 57, 83], FL-based solutions are often adversely affected by non-Independent and Identically Distributed (non-IID) data. Hence, to also face such issues, many learning strategies have been proposed in recent years. In 2020, Karimireddy et al. [46] presented the Stochastic Controlled Averaging algorithm (SCAF-

FOLD) that tries to estimate the update direction for server and client models. More precisely, the proposed algorithm, validated in the presence of 100 clients and unbalanced dataset partitions, has outperformed the most famous state-of-the-art Federated algorithms by slightly reducing the required time effort. Li et al. proposed [57] an extension of the FedAvg algorithm based, at each iteration, on the selection of a subset of clients through their local loss function values. The presented FedProx algorithm, tested on four famous federated datasets, has achieved comparable IID and non-IID loss functions but at the expense of slower convergence. In 2021, Lu et al. presented [61] an improvement of the FedAvg algorithm based on Earth Mover's Distance (EMD) between central and local parameters. More precisely, a new metric, defined as node degree contribution, is derived in order to improve the local models-related aggregation at each iteration. Also, the experimental results, carried out with 100 clients and a different number of iterations, have shown a similar convergence behavior and a slight improvement in accuracy compared with the FedAvg algorithm. Finally, Paragliola et al. [81] proposed a PartialNet Strategy that reduces communication costs by considering partially trained models that, at each iteration, are aggregated by the central server if and only if they satisfy a given threshold. The following strategy, validated on a real-world dataset regarding people affected by hypertension, has proven its effectiveness by reducing communication costs in both IID and non-IID data scenarios.

Tab. 1.2 summarizes such discussed FL-based solutions by highlighting their benefits and issues, respectively.

Authors	Title	Benefit	Issue
Karimireddy et al. [46]	SCAFFOLD: Stochastic Controlled Averaging for Federated Learning	Convergence time reduction	Additional hyperparameters
Li et al. [57]	Federated optimization in heterogeneous networks	Similar loss function values	Convergence time increase
Lu et al. [61]	Parameters Compressed Mechanism in Federated Learning for Edge Computing	Convergence time reduction	Additional hyperparameters
Paragliola et al. [81]	Definition of a novel federated learning approach to reduce communication costs	Communication cost reduction	Additional hyperparameters

Table 1.2: FL-based approaches proposed for non-IID data-related issues.

However, as shown in Tab. 1.2, the proposed FL-based strategies are often characterized by additional hyperparameters whose tuning complexity could limit their applicability [49, 78, 125]. Therefore, in order to face the discussed issues and preserve the intellectual property and privacy of the involved subjects, I present a Markov chains-based dynamic detector. I accomplish this by leveraging the effectiveness of Markov chains employed in a federated logic. Also, I show the effectiveness of the proposed approach in non-IID data scenarios that, instead, often adversely affect the convergence and learning processes of the classic FL-based models.

1.3 CONTRIBUTIONS AND ORGANIZATION

Therefore, on the basis of highlighted weaknesses, I focused my PhD activities on proposing new empowered strategies capable of guaranteeing the success of early alerting facilities for Malware detection activities. The achieved results, supported by my publications, are reported in the remaining of the thesis as follows:

Chapter 2 will focus on Malware classification related to Android-based devices, which represent one of the most famous hostile activity sources continuously monitored to preserve IoT ecosystems. First, it will highlight the main advantages of both dynamic and static-based solutions. Then, it will remark on the effectiveness of dynamic features, arranged as API-Images, to classify several Malware families through a Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN), respectively.

Chapter 3 will present an extension of such API-Images, called API-Streams, that is employed to face an enhanced Malware classification as corresponding Video-Classification tasks. Hence, the effectiveness of the proposed approach is proven by classifying different Malware families through a CNN-LSTM Sparse Autoencoder-based detector.

Chapter 4 will continue the discussion about the Android Malware classification by analyzing the effectiveness of a static-based approach. Therefore, it first presents new special features called Permission Maps (Perm-Maps), which combine information related to Android permissions and their corresponding severity

levels. Then, the related effectiveness is proven through a CNN enhanced by a training process based on federated logic.

Chapter 5 will combine the effectiveness of dynamic-based approaches with the cooperation of federated IoT devices by providing a privacy-preserving Malware detector. In this direction, the capabilities of Markov Chains and associative rules are enhanced in order to improve the most famous Federated Learning (FL) based approaches, which are often adversely affected by non-Independent and Identically Distributed (non-IID) data in terms of both the required training time and classification results, respectively.

Finally, Chapter 6 will present the Conclusions about the reported works by highlighting possible and relevant future research directions. More precisely, it discusses how enhanced DL-based strategies will be needed to face the more complex and future issues related to network security scenarios.

2

ANDROID MALWARE CLASSIFICATION THROUGH DYNAMIC FEATURES AND NEURAL NETWORKS

2.1 INTRODUCTION

Due to their popularity and open nature, Android OS-based devices have attracted several end-users around the World and currently are one of the main targets for malware threats. Indeed, as highlighted by famous cybersecurity companies, the presence of unprotected devices (e.g. smartphones, tablets, smart TVs, and wearable devices) represents a great opportunity for malware developers to perform several malicious activities, such as stealing data and launching cyberattacks. For instance, a recent study by Symantec [104] has reported that an average of about 38,000 new malware applications are daily detected, while according to the Statista report [102], the number of mobile app downloads observed worldwide will reach 352 billion in 2021.

Therefore, based on these studies and the rapid growth of malware technologies explicitly targeted for Android platforms [87], researchers are attempting to propose more effective detectors capable of discovering more complex malware and supporting the related zero-day detection. To this purpose, also thanks to the great success of Machine Learning (ML) and Deep Learning (DL) based techniques in the last decade, several static and dynamic analysis-based solutions have been proposed in order to learn relevant features from the analyzed applications [108]. For instance, Aonzo et al. [6] presented BAdDroIds, a mobile application that leverages deep learning for detecting malware by considering static features, such as the required permissions and the API methods extracted from the DEX file. Abderrahmane et al. [1] employed a Convolutional Neural Network (CNN) to classify malware applications using dynamic features, namely a matrix representation of system calls.

However, malware static analysis-based approaches can be strongly affected by obfuscation tools. Also, they become ineffective against most existing malware capable of evading pattern-matching detection mechanisms, namely metamorphic and polymorphic ones [40]. Hence, in order to overcome these issues, several dynamic-based solutions have been proposed to analyze the behavior of an application by considering several related aspects, such as parameter monitoring and API-Calls tracing [32].

Therefore, to also provide detection models with an adequate generalization capability, the main aim of the following chapter is to prove the effectiveness of dynamic features in classifying several Android malware families through DL-based approaches[1]. More precisely, a features representation technique, called API-Images [22], is used to represent the dynamic behavior of applications as corresponding images. Then, the derived API-Images are employed to train a Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN).

The remainder of the chapter is organized as follows. Sec. 2.2 will present a preliminary overview of the different DNNs adopted typologies. Sec. 2.3 will describe the employed approach based on API-Images. Finally, Sec. 2.4 will discuss the experimental results, while Sec. 2.5 will show the conclusions and future work.

2.2 BACKGROUND

This Section briefly describes the main building blocks of this thesis, namely DNNs and the employed features. First, I report some theoretical information about CNNs and RNNs that represent the core of the discussed detectors. Then, I present an overview of the sandbox tool used to perform the malware analyses. Finally, I describe the main characteristics of the employed dataset and its provided features.

1 Article published in [27]

2.2.1 *Deep Neural Network*

Deep Neural Networks (DNNs) are an extension of the standard neural networks characterized by several hidden layers and neurons, which confer the capability to learn a set of relevant features and thus gain good classification results. For this reason, according to their ability to develop robust prediction models and adjust to non-linear environments, DNNs are increasingly being used in a multitude of application domains [96].

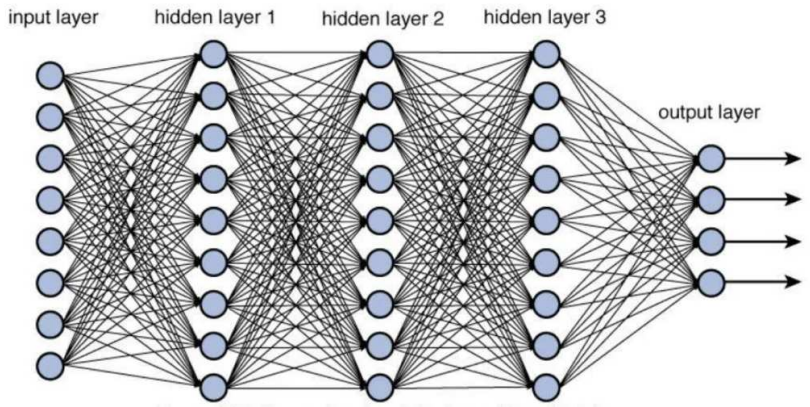

Figure 2.1: DNN-related architecture [82].

DNNs, also named deep fully-connected feed-forward neural networks, are commonly described through a Directed Acyclic Graph (DAG) in which data flows unidirectionally from the input to the output layer [55]. Their related architecture, shown in Fig. 2.1, is derived by setting different hyper-parameters during the training process [36]. Also, to provide a prediction, DNNs present an output layer composed of a fixed number of neurons corresponding to the total number of possible classes. Therefore, as done in this thesis, DNNs provide a probability distribution that describes the possibility that a data sample being classified in its corresponding category.

2.2.2 *Convolutional Neural Network*

Convolutional Neural Networks (CNNs) represent a class of famous DL-based models that have been effectively applied in

various scientific fields, including machine vision [100], Natural Language Processing (NLP) [94], and computational biology [2]. Their success is gained by a particular mathematical process applied in the hidden layers, which confer to these networks the ability to extract relevant and correlated features from input data [30].

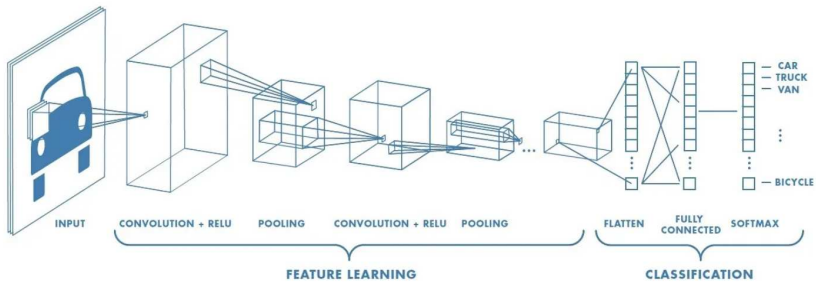

Figure 2.2: CNN-related architecture [103].

As shown in Fig. 2.2, a CNN can use two principal operations sequentially applied in each layer, called Convolution and Pooling. Convolution is the first operation that employs multiple convolutional filters, each responsible for detecting a particular feature from the input data. Then, the Pooling operation (e.g. Max Pooling) can be applied to aggregate the previous features among them in order to derive new relevant information and reduce the output size, respectively. Also, since the derived features are usually arranged into a two-dimensional or three-dimensional vector, an additional third operation of Flattening is applied to convert each matrix into a corresponding one-dimensional vector.

Therefore, in this thesis, I use CNNs to extract flattened relevant features that are directly and indirectly employed, through a DNN, to perform several detection and classification tasks [10].

2.2.3 *Recurrent Neural Network*

Recurrent Neural Networks (RNNs) can work on instances structured as progressive observations (i.e. time series) by considering their mutual dependencies and evolution over time. In this way, a given output depends on the previous ones [10]. To accomplish this, RNNs present many recurrent layers trained using a variant

of the Backpropagation (BP) algorithm, called Backpropagation Through Time (BPTT) algorithm. The application of BPTT implies that, as shown in Fig. 2.3, the unrolling of the trained network as a deep network with multiple hidden layers.

To build a RNN, we can use SimpleRNN and Long Short-Term Memory (LSTM) layers as input or hidden layers, while the output part often consists of a DNN employed to obtain the input data classification. In this thesis, I use SimpleRNN and LSTM layers to analyze recurrence phenomena related to Malware classification and traffic anomaly detection, respectively. Also, notice that the LSTM layer is employed to overcome the vanishing gradient problem [39].

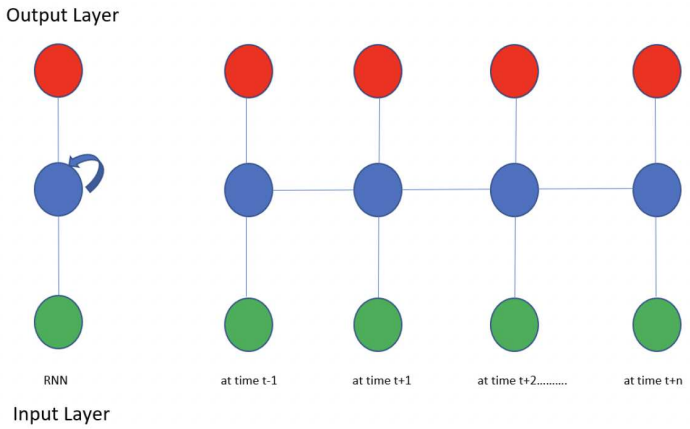

Figure 2.3: RNN-related architecture [10].

2.2.4 *Sandboxing approach*

Sandbox environments can be employed in several configurations, depending on the needs of security experts. Sandboxes can be Virtual Machine (VM)-based, emulating an end user's operating system, or fully simulating a physical device [109]. More precisely, a sandbox is a proactive system for malware detection that runs a suspicious application in an isolated environment (e.g. a new malware typology) and provides several related features [47].

Therefore, to perform Malware analyses and extract the employed static and dynamic features, I used the CuckooDroid

Sandbox [12]. I chose such tool because it is an extension of Cuckoo [11] specifically designed to analyze Android applications and overcome anti-emulation strategies. Indeed, these strategies adversely affect the most famous sandboxing tools, such as DroidMat [117] and CopperDroid [106].

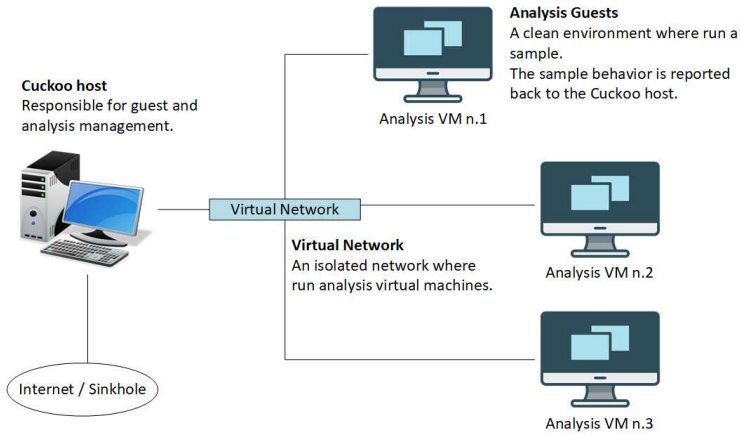

Figure 2.4: High-level architecture of CuckooDroid.

As shown in Fig. 2.4, the high-level architecture of CuckooDroid consists of two main parts, namely the Host and Guest, respectively. The Guest part can consist of several physical or virtual machines in which applications are analyzed, while the Host part is a machine devoted to running the central management software. More precisely, the management software allows the submission of applications and manages the communication between the Host and Guest parts, respectively. Also, the CuckooDroid sandbox can communicate with three guest machines types, namely: with an Android emulator installed on a Linux machine (Android On Linux Machine), with a physical Android-based device (Android device cross-platform), and with an Android emulator installed on the host machine (Android emulator). Fig. 2.5 shows the architecture of the third type of guest machine (i.e. Android Emulator) employed to perform malware analyses.

More precisely, it consists of the following main modules:

- Python Agent: is the script implementing the central management software;

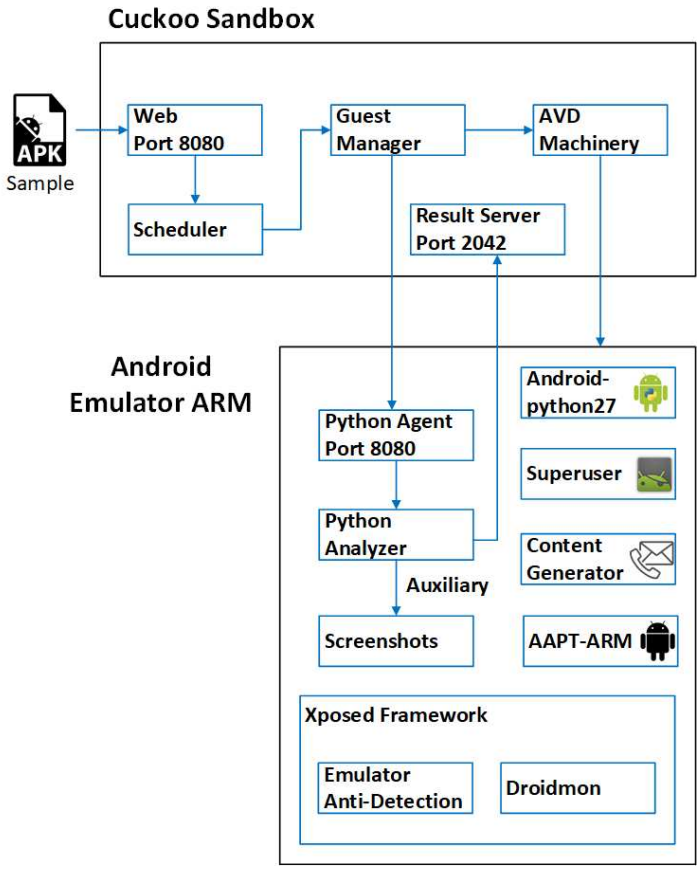

Figure 2.5: Android emulator architecture.

- Android Analyzer: is the app analyzer installed during the configuration of each guest machine;
- Xposed: is a framework that can communicate with other modules or applications, such as the Emulator Anti-Detection module and Droidmon;
- Superuser: is an Android application that can manage root permissions required by applications;
- Content Generator: is an Android application that generates a random list of contacts;
- AAPT-ARM: is a tool that extracts static features and other information related to the analyzed application.

2.2.5 *Dataset and Analyzed features*

Since static and dynamic-based approaches are often combined to improve the probabilities of identifying an application as malware, this thesis discusses the effectiveness of dynamic API-Calls and Android Permissions, directly and indirectly, employed to enhance the DL-based detectors. These basic features have been extracted through the CuckooDroid Sandbox and are available within a recently proposed dataset, namely the Unisa Malware Dataset (UMD) [27].

More precisely, since one of the most critical malware analysis challenges is providing a dataset that represents the malware's characteristics directly employable for models training and knowledge construction, I created the following dataset by considering the 24553 applications of Android Malware Dataset (AMD) [58, 115] and 5560 applications of Drebin [7], respectively. I chose such datasets because they have been widely employed in literature by achieving notable results [13, 67, 77, 124]. Instead, I did not include other recently published datasets (e.g. those available in [75]) because they do not consider some famous malware families or do not provide some post-analysis features, such as permissions, API Calls, and network packets.

Thus, out of 30113 applications considered, only 25275 have been successfully analyzed and thus uniquely stored through their SHA256 value. Therefore, on the basis of this process, I collected 20426 malware of AMD and 4849 of Drebin, respectively. Tab. 2.1 shows the five most representative families of the considered datasets, while Tab. 2.2 summarizes the main characteristics of UMD.

Dataset	Malware Families				
AMD	Airpush	Dowgin	FakeInst	Mecor	Youmi
	5989	2985	2167	1820	1244
Drebin	FakeInst	DKFu	Opfake	Plankton	BaseBridge
	881	635	607	443	314

Table 2.1: Representative post-analysis families of AMD and Drebin.

	Total Apk	Analyzed Apk	Families	Dim. (GB)
AMD	24553	20426	66	100.08
Drebin	5560	4849	143	17.55
Total	30113	25275	209	117.63

Table 2.2: UMD main characteristics.

Next, in order to provide an optimized version of UMD (UMD-V2), I merged the two main folders (AMD and Drebin) by applying the following two steps:

1. Merge the identical families by removing duplicates;
2. Remove all applications characterized by some missing or malformed files.

More precisely, the application of both steps has removed 24 malware and 990 applications. Consequently, the dataset dimension (Dim.) has also been reduced by 5 GB. Tab. 2.3 compares the two versions of UMD datasets, while Tab. 2.4 summarizes the most representative post-cleaning malware families.

	Total Apk	Families	Dim. (GB)
UMD-V1	25275	209	117.63
UMD-V2	24285	185	112.45
Difference	990	24	5.18

Table 2.3: Comparison between the UMD's versions.

Dataset	Malware Families				
UMD-V2	Airpush	Dowgin	FakeInst	Mecor	Youmi
	5983	2985	2917	1820	1239

Table 2.4: Representative post-cleaning families of UMD.

2.3 THE EMPLOYED APPROACH

This Section describes the employed approach for training the CNN and RNN-based classifiers, respectively. For this reason, I first provide an overview of dynamic API-Calls. Then, I describe a famous API-Calls-based representation technique, called API-Images [22], which has been proven effective in several malware classification tasks.

2.3.1 *API-Calls*

API-Calls play a relevant role in Android malware analysis because they summarize the behavior of analyzed applications. More precisely, since API-Calls represent the API methods sequence invoked over time, they are often characterized by some extra information, such as the timestamp, passed arguments, and returned values. Therefore, API-Calls can be strongly helpful in supporting Malware classification tasks.

Notice that the employed CuckooDroid Sandbox can collect API-Calls through a module called Droidmon [42], which traces API-Calls and stores them in a dedicated file called *droidmon.log*. More precisely, as shown in Fig. 2.6, this file contains the list of traced API-Calls and the related information arranged as JSON objects, respectively.


```
{"timestamp":1595723134491,"result":"false","class":"java.io.File","method":
{"timestamp":1595723134521,"result":"false","class":"java.io.File","method":
{"timestamp":1595723134554,"result":"false","class":"java.io.File","method":
{"timestamp":1595723134652,"result":"true","class":"java.io.File","method":"
{"timestamp":1595723134655,"result":"\/data\/data\/com.freegame.kkknfjbgbgra
{"timestamp":1595723134699,"result":"358240051111110","class":"android.telep
{"timestamp":1595723134722,"result":"false","class":"java.io.File","method":
```


Figure 2.6: Droidmon.log file.

2.3.2 *API-Images*

API-Images are a set of sparse matrices representing a complete snapshot of analyzed applications. More precisely, an API-Image can be built through the following two steps:

1) Identification of each API-Call with a unique ID.

2) Matrix creation through the traced API-Calls sequence.

The first step can be accomplished using a dictionary in which each textual API-Call corresponds to a numeric ID. Then, given a traced API-Calls sequence, every two consecutive calls can be used as coordinates to build the related API-Image matrix. Hence, each pair of API-Calls is employed to draw a fixed point in the API-Image. For instance, let a_1, a_2, and a_3 be three APIs where a_2 is called after a_1 and a_3 is called after a_2. On the basis of described process, we can consider only two pairs of coordinates, $P1 = (a_1, a_2)$ and $P2 = (a_2, a_3)$, respectively. Also, since an application may invoke different times the same API-Calls sequence, some pairs can occur more than once. This situation can be managed through different colour scales (e.g. RGB or Gray-scale) in which each colour represents a frequency.

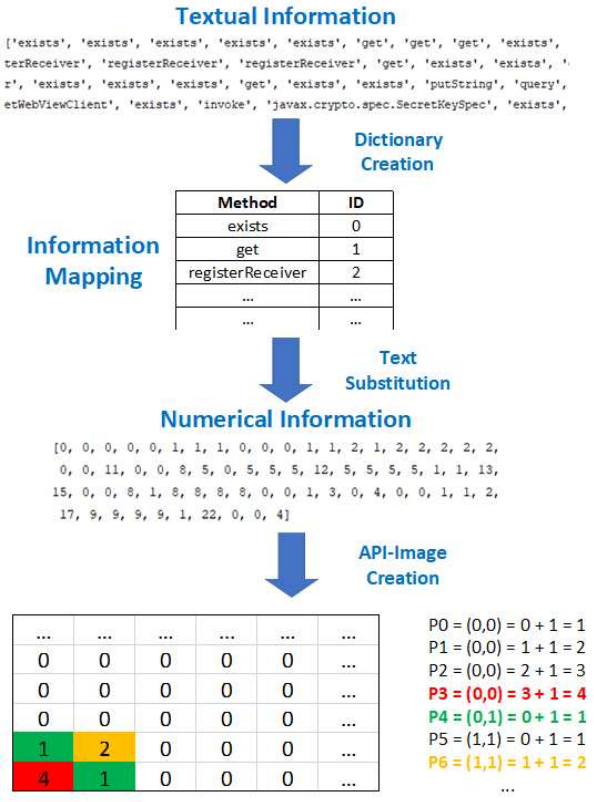

Figure 2.7: The workflow to obtain an API-Image.

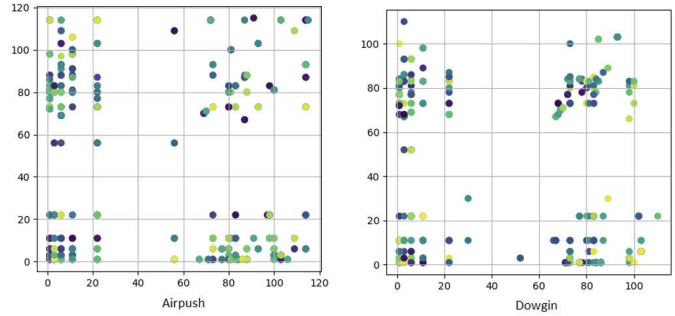

Figure 2.8: An example of 2 API-Images.

Fig. 2.7 summarizes the workflow to generate an API-Image, while Fig. 2.8 shows two RGB API-Images representing the Android malware families Airpush and Dowgin, respectively. More precisely, the reported families show frequencies related to each API-Call pair. Indeed, for each fixed point, a light colour represents those pairs repeated a few times, while a dark colour represents those pairs repeated many times. Besides, it is possible to observe how both images are characterized by two well-distinct behavior that uniquely identifies each malware family.

2.4 EXPERIMENTAL RESULTS

The goal of the presented experiments is devoted to demonstrating the effectiveness of API-Images in enhancing the proposed DL-based classifiers. To accomplish this, I show the abilities of CNN and RNN in classifying several malware applications and overcoming the most famous ML-based approaches, respectively.

2.4.1 *Dataset and Experimental setting*

The dataset considered in the following experiments has been derived by Unisa Malware Dataset (UMD), composed of about 2500 applications grouped into 5 Android families: Airpush, Dowgin, DroidKungFu (DKFu), FakeInstaller (FakeInst), and Opfake. More precisely, in order to obtain the related API-Call sequences, I analyzed each application through the Cuckoo Sandbox tool [12]. Thus, by applying the previously described process (see

Fig. 2.7), the obtained sequences have been used to generate one-channel API-Images (116×116) in accordance with the maximum number of different API-Calls observed. Next, to evaluate the performances of the employed classifiers, I divided the resulting dataset into training and testing sets according to the 70/30 criteria, as reported in Tab. 2.5.

Family	Training	Testing	Total
Airpush	350	150	500
Dowgin	350	150	500
DKFu	350	150	500
FakeInst	350	150	500
Opfake	350	150	500
Total	1750	750	2500

Table 2.5: Dataset division according to the 70/30 criteria.

Finally, notice that any experiments (including training and testing phases) have been done with an iMac equipped with an Intel 6-Core i7 CPU @ 3.20GHz and 16 GB RAM.

2.4.2 *Evaluation metrics*

To appreciate the classification quality of the employed DNNs, I used the following evaluation metrics derived from the multi-class confusion matrix: Accuracy (Acc.), Sensitivity (Sens.), Specificity (Spec.), Precision (Prec.), F-Score (F-Mea.), and Area Under the ROC Curve (AUC).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

$$Sensitivity = \frac{TP}{TP + FN} \tag{2.2}$$

$$Specificity = \frac{TN}{TN + FP} \tag{2.3}$$

$$Precision = \frac{TP}{TP + FP} \tag{2.4}$$

$$F - Score = \frac{2 * Sens * Prec}{Sens + Prec} \quad (2.5)$$

$$AUC = \frac{Sens + Spec}{2} \quad (2.6)$$

For each category, TPs (True Positives) refer to the applications correctly classified, while TNs (True Negatives) refer to the applications correctly identified in another category. Conversely, FPs (False Positives) are the applications mistakenly identified as the considered category, while FNs (False Negatives) are the applications mistakenly identified in another category. Also, to achieve a global perspective of the detector effectiveness, the average performance values (Avg.) among all the observed malware classes have been derived.

2.4.3 *DNNs description and Results*

As mentioned earlier, I used a CNN and a RNN to classify several Android malware families. Accordingly, in this Section, the DNNs-related implementation details are first reported. Then, the experimental results are shown. Notice that I derived the described architecture according to the results derived by using the 70/30 criteria. Also, I did not employ simple DNNs (composed only of fully connected layers) because they have already been adopted in literature to face similar tasks [51].

First, the employed CNN includes a sequence of two Conv2D layers with kernel_size=(4,4), max pool_size=(2,2), and activation=relu, characterized by having 64 and 32 filters, respectively. After that, a Flatten layer is employed to map the extracted features as one-dimensional latent vectors and fed a fully connected softmax network. Hence, I used three Dense layers having activation=relu, dropout=0.4, and 128 neurons, respectively. In addition, to consider the classification results as probability distributions, I used a fourth dense layer with five neurons and activation=softmax as the output layer.

Instead, regarding the RNN, it includes a sequence of five SimpleRNN layers with 100 neurons and return_sequences set to True for the first four layers. Then, to again consider the classification results as probability distributions, I used a sixth dense layer with five neurons and activation=softmax as the

output layer. Therefore, I trained both networks with Adam optimizer and CategoricalCrossentropy loss function for 150 epochs and batch_size=256.

In addition, the following architectures have been derived by varying the following hyper-parameters:

- numConvLayers: the number of Conv2D layers (1, 2, 3);
- numFilters: the number of filters for each Conv2D layer (8, 16, 32, 64);
- kernel_size: kernel_size values ((2,2), (3,3), (4,4));
- numPoolLayers: the number of MaxPool2D layers (1, 2, 3);
- pool_size: pool_size values ((2,2), (3,3), (4,4));
- numDenseLayers: the number of Dense layers (2, 3, 4);
- numNeurons: the number of neurons for each Dense layer (5, 16, 32, 64, 128, 256, 512);
- dropout: dropout values for each Dense layer (0.2, 0.3, 0.4, 0.5);
- activation: activation functions used (relu, softmax, sigmoid);
- batch_size: batch_size values (64, 128, 256, 512);
- loss: employed loss functions (Mean Squared Error - MSE, CategoricalCrossentropy).

Tabs. 2.6 and 2.7 report the excellent performance metrics derived by the multi-class confusion matrix for the employed CNN and RNN, respectively.

2.4.4 *Comparison and Discussion*

Finally, to highlight the potentialities of the presented DL-based classifiers, I compared the achieved results with those derived by the most famous ML-based methods provided by WEKA [116], namely: the Multi-Layer Perceptron (MLP) classifier, J48 Trees

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
DKFu	0.9960	0.9948	0.9814	1.0000	0.9906	0.9974
FakeInst	0.9987	1.0000	1.0000	0.9928	0.9964	0.9964
Opfake	0.9972	1.0000	1.0000	0.9869	0.9935	0.9968
Airpush	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Dowgin	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Avg.	0.9984	0.9990	0.9963	0.9960	0.9961	0.9981

Table 2.6: CNN performance metrics.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
DKFu	0.9987	1.0000	1.0000	0.9927	0.9964	0.9964
FakeInst	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Opfake	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Airpush	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Dowgin	0.9987	0.9982	0.9935	1.0000	0.9966	0.9974
Avg.	0.9995	0.9997	0.9987	0.9986	0.9986	0.9988

Table 2.7: RNN performance metrics.

(J48) algorithm, and Naive Bayes (NB) classifier, respectively. I accomplished this by arranging the API-Images as corresponding flattened sequences. Tab. 2.8 compares the achieved classification metrics with those derived by each ML-based method used.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
RNN	0.9995	0.9997	0.9987	0.9986	0.9986	0.9988
CNN	0.9984	0.9990	0.9963	0.9960	0.9961	0.9981
MLP	0.9990	0.9950	0.9950	0.9950	0.9950	0.9950
J48	0.9970	0.9870	0.9870	0.9870	0.9870	0.9960
NB	0.9710	0.8970	0.8890	0.8870	0.8860	0.9900

Table 2.8: Comparison with ML-based methods provided by WEKA (only Avg. values are reported).

As shown in Tab. 2.8, the proposed DNNs outperformed each traditional ML-based method taken into consideration. More precisely, it achieved an average F-Score improvement of 11% on the NB classifier. In addition, the J48 Trees and the MLP classifier, which have also achieved excellent classification results, have been outperformed by the employed DNNs. Indeed, CNN and RNN have obtained the best classification metrics (see Precision, Sensibility, and F-Score) in recognising the different malware families. Therefore, these results prove the effectiveness of dynamic features in reducing classification error and enhancing the DL-based classifiers, respectively.

2.5 CONCLUSIONS AND FUTURE WORKS

In this Chapter, I investigated the effectiveness of dynamic features in enhancing DL-based malware classifiers. More precisely, I analyzed the five most representative Android malware families of the Unisa Malware Dataset (UMD). Hence, I first arranged the traced API-Calls sequences as corresponding one-channel API-Images. Then, I proved their effectiveness by employing CNN and RNN-based classifiers, respectively. The obtained results have shown an average accuracy of 99% for both employed DNNs by also outperforming the most famous ML-based approaches.

However, due to the continuous release of sophisticated and dangerous malware, this study highlights two possible future works. First, enhanced DL-based approaches should be investigated in order to propose new malware detectors capable of performing Run-time classification tasks. For instance, Chap. 3 discusses the effectiveness of Sparse Autoencoders (SAEs) in considering streams of API-Images sampled over time. Second, the static-based approaches should be explored to propose malware detectors not affected by the most famous obfuscation techniques. For instance, Chap. 4 presents a Federated classifier based only on Android Permissions and their corresponding severity levels.

3

ANDROID MALWARE DETECTION THROUGH API-STREAMS AND CNN-LSTM AUTOENCODERS

3.1 INTRODUCTION

The sudden and rapid displacement of the global workforce towards the houses, caused by the outbreak of the COVID-19 pandemic, has forced companies worldwide to make significant changes to their infrastructures. In this new scenario, mobile devices have been used more than ever to access corporate systems making them more susceptible to new cyber threats. For instance, as reported by the Mobile Security Report of Check Point [53], 97% of organizations have faced several threats from mobile devices, while 46% of organizations have had at least one employee that downloaded a malicious app. Consequently, due to their popularity, mobile devices remain one of the main targets for cyber-criminals, which constantly analyze systems-related vulnerabilities and the worldwide trending topics in order to conduct several hostile activities [50, 54].

For this reason, also thank the great success of Deep Learning (DL) in many research fields [30, 43, 126], several DL-based solutions have been investigated to face malware classification by considering both static and dynamic-based approaches [1, 6, 51, 56]. However, as remarked in Chap. 2, static-based solutions can be strongly affected by obfuscation tools. Also, they become ineffective against most existing malware capable of evading pattern-matching detection mechanisms, namely metamorphic and polymorphic ones [40]. On the other hand, dynamic-based approaches cannot perform classification tasks at Run-time because they consider features derived by the post-analysis report, which describes applications that have already been executed and thus shown their behavior.

Therefore, the main goal of this chapter aims to propose a novel feature representation technique, called API-Streams[1], to perform Run-time malware classification tasks. To accomplish this, the related API-Streams workflow, based on multiple API-Images sampled during the applications' execution, is presented. Then, the effectiveness of the proposed approach is investigated by facing several Video-Classification tasks. I accomplished this by combining the capability of CNN-LSTM Sparse Autoencoders (CNN-LSTM-SAEs) in finding relevant features, the goal of Video Classification in distinguishing objects from a stream of frames, and the classification abilities of Deep Neural Networks (DNNs), respectively. Therefore, unlike the most famous state-of-the-art solutions [1, 22, 51], the presented API-Streams might be involved in Run-time monitoring processes capable of detecting malicious applications in the shortest possible time. Consequently, the proposed approach might also be helpful in reducing the damages caused when applications are still running.

Hence, the main contributions of this chapter can be summarized as follows:

1. A novel approach called API-Streams, based on several API-Images, is proposed in order to represent malware behavior at Run-time;

2. Several Video-Classification tasks, based on CNN-LSTM Sparse Autoencoders, are faced to demonstrate the effectiveness of the proposed approach.

The remainder of the chapter is organized as follows. Sec. 3.2 will present a preliminary overview of the different DNNs adopted typologies. Sec. 3.3 will describe the proposed approach based on API-Streams. Finally, Sec. 3.4 will discuss the experimental results, while Sec. 3.5 will show the conclusions and future work.

1 Article published in [26]

3.2 BACKGROUND

This Section briefly describes the main building blocks employed by the presented Android malware classifier, namely Autoencoders and Stacked Neural Networks.

3.2.1 *Autoencoders*

As depicted in Fig. 3.1, Autoencoders (AEs) represent one of the most famous unsupervised neural network configurations, characterized by having a symmetrical structure composed of two main components known as encoder and decoder, respectively. From a mathematical perspective, the encoder converts the input data sample, x, into a low-dimensional latent vector $h = f(x)$, while the decoder reconstructs the input from the latent vector as $\tilde{x} = g(h)$, such that $\tilde{x} = x$ [8]. Therefore, since the principal objective of an AE is learning to reconstruct its input as the desired output, it is forced to extract only the relevant and representative features from the input data.

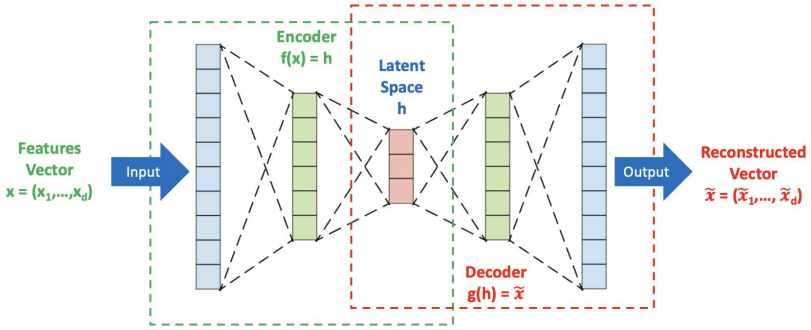

Figure 3.1: Schematic representation of an Autoencoder.

However, differently from the classical features selection and extraction techniques, the effectiveness of AEs descends on the capability of representing the input using non-linear combinations of the derived features. For this reason, AEs outperform the most famous dimensionality reduction-based techniques, such as Principal Component Analysis (PCA), Linear discriminant analysis (LDA), and Discriminant Function Analysis (DFA) [79].

Finally, since the encoding-decoding process can be implemented through different neural network topologies, seven main AEs categories have been proposed in the literature, namely: the Sparse AE [73], the Undercomplete AE [107], the Denoising AE [111], the Deep AE [121], the Convolutional AE [63], the Variational AE [44], and the Contractive AE [90]. In this proposal, we explore the capabilities of CNN-LSTM Sparse Autoencoders (CNN-LSTM-SAEs) in extracting meaningful features derived from the input streams. Also, I remand to [19] and [30] for more detailed information about the theoretical and practical aspects of AEs.

3.2.2 *Stacked Neural Network*

Building reliable and robust neural network models for a specific classification task is often challenging and time-consuming. However, their effectiveness can be improved by combining multiple related models as one entire Stacked Neural Network (SNN). The following theory is founded on the assumption that the composition of several feature typologies, respectively learned by a specific network, decreases the uncertainty and enhances the trade-off between training speed and classification accuracy [74]. Therefore, the training procedure of an SSN starts with the training of each involved model, performed independently, and then proceeds by fine-tuning the whole network using a supervised scheme, such as the Backpropagation algorithms.

Similarly, in this chapter, the capabilities of CNN-LSTM-SAEs and DNNs are combined to propose a SNN-based malware classifier capable of detecting malware applications during their execution.

3.3 THE PROPOSED APPROACH

This Section presents the proposed approach for implementing the Android malware classifier. I accomplished this through the following two main steps, namely: finding out the more representative features from the input streams and then performing classification using a Stacked Neural Network.

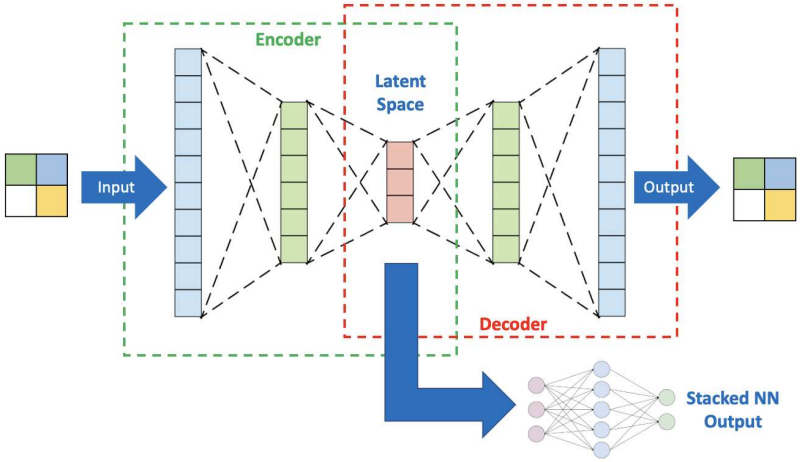

Figure 3.2: SNN high-level architecture.

More precisely, as shown in Fig. 3.2, the former step is performed using a Sparse AE (SAE) implemented by a CNN-LSTM network that, as known in the literature, is capable of mining both spatial and temporal features [30]. Whereas, for implementing the classification step, the latent layer of the SAE is employed as input to a fully connected neural network. In this way, I combine the capability of SAEs in finding compact and relevant features, the goal of Video-Classification in distinguishing objects from a stream of frames, and the classification abilities of DNNs, respectively.

Therefore, I first present a brief mathematical formulation of AEs. Then, I combine the given formulation of CNN-SAEs and LSTM-SAEs by defining the employed CNN-LSTM-SAE. Finally, I describe the generation process of API-Steams by providing some definitions.

3.3.1 *AE Definition*

In this configuration, I suppose that the AE used for processing aggregated API-Images, with reference to Fig. 3.1, includes only three layers typologies, namely: input, hidden, and output layers, respectively.

Let $x \in \mathbb{R}^d$ be the input vector of Fig. 3.1, and let $x^{AE} \in \mathbb{R}^{d'}$ be the input of the AE under consideration, then $x \equiv x^{AE}$ and $d = d'$.

As explained in Subsection 3.2.1, an AE tries to reconstruct its input by encoding it into a latent space h, that is then decoded into an output $\tilde{x}^{AE}$ as reported in the following:

$$\tilde{x}^{AE} = y_{(W',b')}(h_{(W,b)}(x^{AE})) \equiv x^{AE} \tag{3.1}$$

where (W, b) and (W', b') represent the matrix of the weights and the bias vector of the encoder and decoder respectively, whereas y is the activation function of the decoder.

Moreover, let n be the number of hidden neurons of AE, then $W \in \mathbb{R}^{n \times d'}$, $b \in \mathbb{R}^n$, and $h_{(W,b)}$ is given by:

$$h_{(W,b)}(x^{AE}) = \sigma(Wx^{AE} + b) \tag{3.2}$$

where σ is the activation function of the encoder.

Since an Autoencoder is trained by minimizing a loss function $\mathcal{F}$, it is possible to consider additional constraints, also referred to as regularization terms, to give the AE some specific capabilities. For instance, Sparse AEs are often employed to extract meaningful and relevant features from input data and, consequently, to improve the classification performance. Specifically, Sparsity can be achieved with different strategies (e.g. L1 regularization and KL regularization) by letting the AE of interest to have only a few nodes that are simultaneously active (1 in theory) in order to positively affect the learning process [64]. Concerning SAEs, their regularization is accomplished by adding a penalty term to the loss function, that is:

$$\mathcal{F}(x^{AE}, \tilde{x}^{AE})_{Sparse} = \mathcal{F}(x^{AE}, \tilde{x}^{AE}) + \lambda \mathcal{S}(W, b) \tag{3.3}$$

where λ expresses the degree of regularization, and $\mathcal{S}(W, b)$ represents the sparsity-related term.

Once the training process is completed, the output of the l^{th} hidden neuron h_l can be derived by:

$$h_l = \sigma(\sum_{k=1}^{d'} w_{lk} x_k^{AE} + b_l) \tag{3.4}$$

Hence, since the input data of a Sparse AE is constrained by $\left\| x^{AE} \right\|^2 \leq 1$, each input data component x_k^{AE} activating the l^{th} neuron is given by:

$$x_k^{AE} = \frac{w_{lk}}{\sum_{m=1}^{d'} (w_{lm})^2}, \ \forall \ k, m \in \{1, ..., d'\} \tag{3.5}$$

which extracts a feature that exactly corresponds to the l^{th} output node. Accordingly, a Sparse AE can learn different sets of characteristics from its input data that is at least equal to the number of considered hidden neurons n.

3.3.2 *CNN-SAE Definition*

As detailed in Subsection 3.2.1, AEs are frequently coupled with different DNNs flavors in order to add new functionalities and learn more complex features from input data. Thus, to fully exploit the effectiveness of API-Images, I use several Convolutional layers to derive relevant relations, which are also referred to as spatial-features [30].

Let $X \in \mathbb{R}^{N_x \times N_y}$ be the CNN-related input, $AI \in \mathbb{R}^{N \times N}$ be an API-Image, and $C^f \in \mathbb{R}^{a \times b}$ be the f^{th} filter, respectively. Then, $X \equiv AI$ and $N_x \times N_y = d = N \times N$.

On these assumptions, the convolution operation, applied on the CNN-input AI with N_f filters, is defined by:

$$F_{i,j} = \sum_{f=1}^{N_f} \sum_{p=1}^{a} \sum_{q=1}^{b} C_{p,q}^f AI_{i+p-1, j+q-1} \tag{3.6}$$

with $F_{i,j}$ the components of the filtered input F.

The size of F is defined through its row F_x and column F_y dimensions, by:

$$F_x = \frac{N_x - a + 2P}{S_x} + 1$$
$$F_y = \frac{N_y - b + 2P}{S_y} + 1 \tag{3.7}$$

where P is the Padding referring to the number of zeros surrounding the border of X, while S_x and S_y represent the Strides related to the row and column. Hence, S_x and S_y manage the shift of the filter on the input matrix.

Since $d = N_x \times N_y$, any x_k can be mapped to a corresponding point $AI_{i,j}$ into a two-dimensional array. Hence, with a limited abuse of notation, follows that:

$$x_k \equiv AI_{i,j} \tag{3.8}$$

with $k \in \{1, ..., d\}$, $i \in \{1, ..., N_x\}$, and $j \in \{1, ..., N_y\}$.

By substituting $F_{i,j}$ of Eq. (3.6) into x_k^{AE} of Eq. (3.4), it yields:

$$h_l = \sigma(\sum_{k=1}^{d'} w'_{lk} x_{\phi(k)} + b_l) \tag{3.9}$$

such that $d' = F_x \times F_y$, while

$$w'_{lk} = \sum_{f=1}^{N_f} \sum_{p=1}^{a} \sum_{q=1}^{b} C_{p,q}^{f} w_{lk} \tag{3.10}$$

and

$$x_{\phi(k)} \equiv AI_{i+p-1,j+q-1} \tag{3.11}$$

Analogously to Eqs. 3.4 and 3.5, Eqs. 3.9 and 3.10 define that w'_{lk} (for each l, k) are the new mined features expressing a more complex knowledge since they are expressed as the linear combination of the original w_{lk}. Therefore, the CNN-SAE configuration can deeply analyze the applications-related dynamic behavior, represented as API-Images, by providing meaningful features to employ for the proposed approach.

3.3.3 *LSTM-SAE Definition*

In this configuration, LSTM layers are used, together with SAE, as a Sequence-To-Sequence (STS) architecture [101] in which the sequence of input features (e.g. time-series of a multivariate input) is encoded into a corresponding fixed-length latent vector and, vice-versa, decoded again as the input sequence. Hence, this confers to LSTM-SAE configuration the capability of mining temporal short and long-distance dependencies within the input features.

Let $X^T = (x^{(1)}, x^{(2)}, ..., x^{(T)})$ be a T-length sequence of input sequence at different timestamps $t \in \{1, ..., T\}$, the STS-encoder, after T recursive updates, produces a synthesized output-vector (y^T) of a predetermined $r \times 1$ dimension, which can be expressed by:

$$y^T = \Psi^T(x^{(1)}, x^{(2)}, ..., x^{(T)}) \tag{3.12}$$

with Ψ^T a non-linear multi-variable vector-valuated function gathering the work of the LSTM cell for T timestamps.

Let y_k^T be the k^{th} component of y^T, then substituting y_k^T into x_k^{AE} of Eq. 3.4, it yields:

$$h_l = \sigma(\sum_{k=1}^{r} w_{lk} y_k^T + b_l) \tag{3.13}$$

Analogously to Eq. 3.4, Eq. 3.9 indicates that the extracted features are a non-linear combination of T vectors of input features, which are also referred to as temporal-features [30]. Hence, since these new features express a compact representation of the input features over time, the LSTM-SAE configuration can be employed to analyze the dynamic behavior when applications are still running.

3.3.4 *CNN-LSTM-SAE Definition*

In this configuration, spatial-features are first extracted, and then their behavior over time is evaluated through temporal-features

mined by the LSTM cells. Thus, the d-dimensional input vector is arranged as a two-dimensional array ($N_x \times N_y$). Next, it is fed to CNN that gives a $F_x \times F_y$ matrix as output (see Eq. 3.7). Finally, T CNN-outputs are combined by the LSTM cells to produce a r-dimensional vector (y^T).

With reference to Eq. 3.6, and according to Eqs. 3.8 and 3.11, the components ($Y_{i,j}^{(t)}$) of the CNN-output (Y^t) at timestamp t are given by:

$$F_u^{(t)} \equiv F_{i,j}^{(t)} = \sum_{f=1}^{N_f} \sum_{p=1}^{a} \sum_{q=1}^{b} C_{p,q}^{f} x_{\phi(u)}^{(t)} \tag{3.14}$$

with $t \in \{1, ..., T\}$, $u \in \{1, ..., d'\}$, and $d' = N_x \times N_y$.

According to Eq. 3.12, the k^{th} component (y_k^T) of y^T is given by:

$$y_k^T = \Psi_k^T(Y^{(1)}, Y^{(2)}, ..., Y^{(T)}), \quad k \in \{1, ..., r\} \tag{3.15}$$

which expresses a relation among T spatial-features derived by the CNN.

Thus, with reference to Eq. 3.4, follows that:

$$h_l = \sigma(\sum_{k=1}^{r} w_{lk} y_k^T + b_l) \tag{3.16}$$

Analogously to Eq. 3.13, Eq. 3.16 indicates that the extracted features are a non-linear combination of T spatial-features obtained by the CNN stage. Hence, the CNN-LSTM-SAE configuration can analyze the applications-related behavior in terms of spatial and temporal-features, respectively. For this reason, I employ a CNN-LSTM-SAE to derive relevant features within a stream of T API-Images generated at Run-time.

3.3.5 *API-Streams Definition*

Informally, an API-Stream can be defined as a set of several API-Images generated at Run-time. More precisely, the API-Streams generation process begins by sampling API-Calls in several Δt

time windows, and for each considered sub-sequence, by generating a corresponding API-Image. Next, the obtained sequence of API-Images can be similarly sampled over time and then involved to generate several API-Streams. Therefore, the following approach assumes that each time window identifies a fixed-length sub-sequence of API-Calls or API-Images considered at Run-time. I respectively refer to them with Δt_{Calls} and Δt_{Images}, such that:

$$\Delta t_{Calls} = K_M * D_{Calls} \tag{3.17}$$
$$\Delta t_{Images} = J_M * D_{Images} \tag{3.18}$$

where K_M is the number of API-Calls considered, D_{Calls} is the distance between 2 API-Calls, J_M is the number of API-Images considered, and D_{Images} is the distance between 2 API-Images. Also, $K_M \geq 2$ where 2 is the minimum number of required API-Calls to achieve an API-Image with at least one fixed point, and $J_M = t$ where t is the number of considered time steps.

Therefore, the total number A_M of API-Images generated during the app's execution is derived as follows:

$$A_M = (L_M - \Delta t_{Calls}) + 1 \tag{3.19}$$

where L_M is the total number of API-Calls of an application M.

Finally, to include additional combinations of API-Streams, I consider any sliding of the temporal window Δt_{Images}. To accomplish this, usually, a specific offset (*Stride*) is used [24]. As a consequence, the total number S_M of API-Streams generated during the app's execution is given by:

$$S_M = \left\lceil \frac{(A_M - \Delta t_{Images})}{Stride} + 1 \right\rceil \tag{3.20}$$

Fig. 3.3 provides an instance of Δt_{Calls} and Δt_{Images} windows, while Fig. 3.4 summarizes the API-Streams workflow and its main steps.

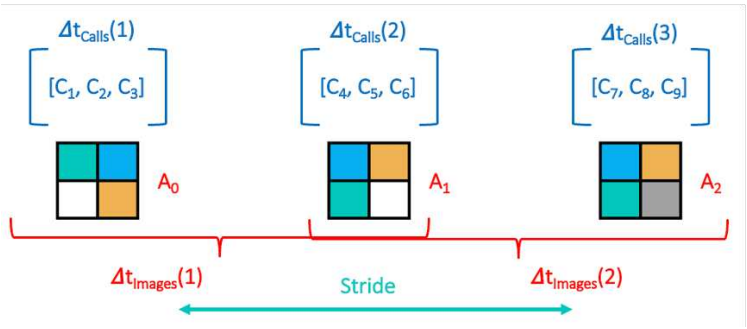

Figure 3.3: An instance of Δt_{Calls} and Δt_{Images} windows.

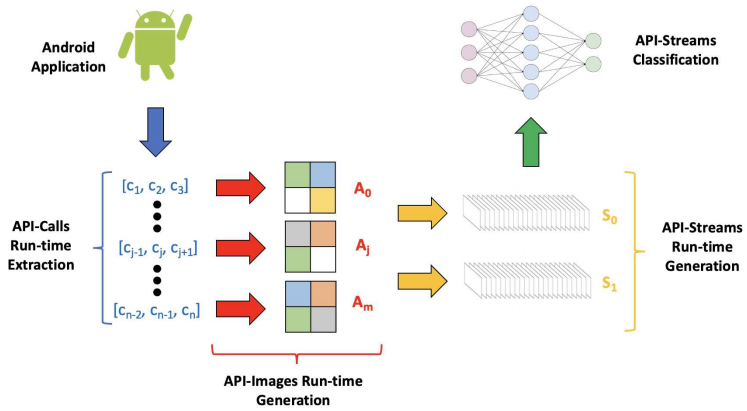

Figure 3.4: API-Streams workflow.

3.4 EXPERIMENTAL RESULTS

The goal of the presented experiments is devoted to demonstrating the effectiveness of API-Streams in classifying Android malware families at Run-time. To accomplish this, I show the abilities of a CNN-LSTM-SAE-SNN in performing Video-Classification tasks in the presence of several unbalanced datasets.

3.4.1 *Dataset and Experimental setting*

The API-Streams datasets considered in the following experiments have been composed of 5 representative Android families of the Unisa Malware Dataset (UMD), namely Dowgin, DroidKungFu (DKFu), FakeInstaller (FakeInst), GinMaster (GinM), and

Plankton. First, by following the workflow shown in Figs. 3.3 and 3.4, I generated A_M API-Images (80×80) in accordance with the maximum number of distinct API-Calls observed. Then, I extracted S_M API-Streams by considering the following parameters:

- K_M: the number of employed API-Calls: (5, 10, 15, 20, 25, 30, 35, 40, 45, 50);
- J_M: the number of employed API-Images: (5, 10);
- D_{Calls}: the distance between 2 API-Calls: (1);
- D_{Images}: the distance between 2 API-Images: (1, 2, 3);
- *Stride*: the distance between 2 time windows: (1, 2).

More precisely, I generated several dataset instances by including, each time, at least 1000, 2000, and 4000 API-Streams for each malware family (namely the bound of S_M). More precisely, the value of S_M, and those related to the other parameters, have been iteratively derived according to the achieved results. Notice that, on 360 different dataset combinations considered, I reported only the instances that gave better results.

Therefore, I split each derived dataset using the 70/30 criteria. Accordingly, I used 70% of each dataset for the learning phase and the remaining 30% for testing. Also, I did an additional validation step by using the 60/40 criteria. The following tables summarize the information about the selected four datasets, which have been generated by considering $K_M = 45$, $J_M = 5$, and $Stride = 1$.

In detail, Tabs. 3.1 and 3.2 report the two datasets generated with $D_{Images} = 1$ by including at least 2000 API-Streams in the former and 4000 API-Streams in the latter. Instead, Tabs. 3.3 and3.4 summarize the two datasets generated with $D_{Images} = 2$ by including at least 2000 API-Streams in the first one and 4000 API-Streams in the second one. For the sake of clarity, I named the following datasets Dataset1, Dataset2, Dataset3, and Dataset4, respectively.

Finally, notice that any experiments (including training and testing phases) have been done with an iMac equipped with an Intel 6-Core i7 CPU @ 3.20GHz and 16 GB RAM.

	Training	Testing	Total
DKFu	1451	662	2113
Dowgin	2445	1020	3465
FakeInst	2802	1236	4038
GinM	1515	624	2139
Plankton	1484	615	2099
Total	9697	4157	13854

Table 3.1: Dataset1 - $D_{Images} = 1$ and 2000 API-Streams.

	Training	Testing	Total
DKFu	2820	1206	4026
Dowgin	5488	2410	7898
FakeInst	2843	1195	4038
GinM	2788	1226	4014
Plankton	2919	1189	4108
Total	16858	7226	24084

Table 3.2: Dataset2 - $D_{Images} = 1$ and 4000 API-Streams.

	Training	Testing	Total
DKFu	1398	615	2013
Dowgin	2794	1151	3945
FakeInst	1398	619	2017
GinM	1411	592	2003
Plankton	1419	632	2051
Total	8420	3609	12029

Table 3.3: Dataset3 - $D_{Images} = 2$ and 2000 API-Streams.

3.4.2 *Evaluation Metrics*

To appreciate the classification quality of the employed model, I used the following evaluation metrics derived from the multi-class confusion matrix: Accuracy (Acc.), Sensitivity (Sens.), Speci-

	Training	Testing	Total
DKFu	2963	1275	4238
Dowgin	2951	1279	4230
FakeInst	2787	1215	4002
GinM	2772	1230	4002
Plankton	2918	1169	4087
Total	14391	6168	20559

Table 3.4: Dataset4 - $D_{Images} = 2$ and 4000 API-Streams.

ficity (Spec.), Precision (Prec.), F-Score (F-Mea.), and Area Under the ROC Curve (AUC).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.21}$$

$$Sensitivity = \frac{TP}{TP + FN} \tag{3.22}$$

$$Specificity = \frac{TN}{TN + FP} \tag{3.23}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.24}$$

$$F - Score = \frac{2 * Sens * Prec}{Sens + Prec} \tag{3.25}$$

$$AUC = \frac{Sens + Spec}{2} \tag{3.26}$$

For each category, TPs (True Positives) refer to the API-Streams correctly classified, while TNs (True Negatives) refer to the API-Streams correctly identified in another category. Conversely, FPs (False Positives) are the API-Streams mistakenly identified as the considered category, while FNs (False Negatives) are the API-Streams mistakenly identified in another category. Also, to achieve a global perspective of the detector effectiveness, the average performance values (Avg.) among all the observed malware classes have been derived.

3.4.3 *Network description and Results*

As mentioned earlier, I analyzed the effectiveness of CNN-LSTM Autoencoders (CNN-LSTM-SAEs) and DNNs to classify several Android malware families at Run-time. Accordingly, in this Section, the network-related implementation details are first reported. Then, the experimental results are shown. Notice that I derived the described architecture according to the results derived by using the 70/30 criteria.

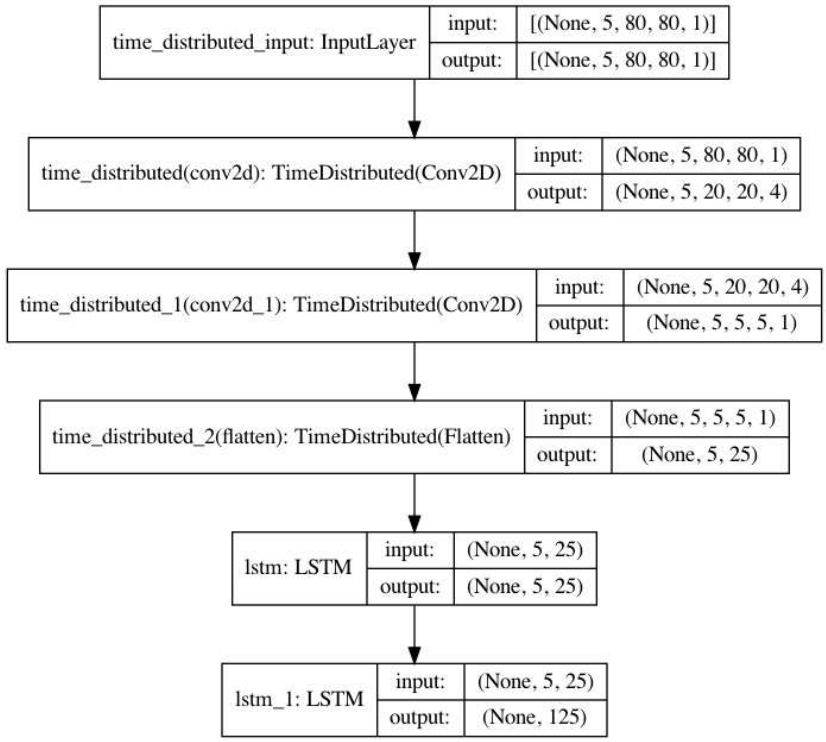

Figure 3.5: Architecture of the employed encoder.

Fig. 3.5 summarizes the high-level organization of the CNN and LSTM layers employed for the encoder side. More precisely, it includes a sequence of two Conv2D layers with kernel_size=(4,4), stride=(4,4), activation=relu, padding=same, and no pooling, characterized by having 4 and 1 filters, respectively. After that, I employed a Flatten layer to map the extracted features as corresponding one-dimensional vectors. Since I used a timeSteps

of 5, three TimeDistributed layers have been employed on the two convolutional layers and the flattening layer, respectively. Next, I used the flattened CNN vectors to fed the first LSTM layer characterized by 25 cells and return_sequence=True. Next, we employed a second LSTM layer including 125 cells and return_sequence=False to achieve a latent vector of 125 features. Hence, after having built the decoder side using the inverse sequence of the encoding layers, I trained the CNN-LSTM-SAE configuration using Adam optimizer and the Mean Squared Error (MSE) loss function for 5 epochs and batch_size=64.

Next, I employed the encoder part to fed a fully connected softmax network comprising two Dense layers having activation=relu, dropout=0.5, and 256 neurons, respectively. In addition, to consider the classification results as probability distributions, I used a third dense layer with 5 neurons and activation=softmax as the output layer. Therefore, I trained the following network with Adam optimizer and the SparseCategoricalCrossentropy loss function [60] for 250 epochs and batch_size=64. Finally, I combined the employed encoder and softmax neural network as a CNN-LSTM-SAE-SNN by performing a fine-tuning step for 20 epochs.

In addition, the following architecture has been derived by varying the following hyper-parameters:

- numConvLayers: the number of Conv2D layers (1, 2, 3);
- numFilters: the number of filters for each Conv2D layer (1, 2, 4, 8, 16);
- kernel_size: kernel_size values ((2,2), (4,4));
- stride: the stride length for each Conv2D layer (1, 2, 4);
- numLSTMLayers: the number of LSTM layers (1, 2, 3);
- LSTM cells: the number of LSTM cells (25, 125, 250);
- timeSteps: the number of observations considered as input time steps (5, 10);
- numDenseLayers: the number of Dense layers (1, 2, 3, 4);

- numNeurons: the number of neurons for each Dense layer (5, 64, 128, 256);
- dropout: dropout values for each Dense layer (0.2, 0.3, 0.4, 0.5);
- activation: activation functions used (relu, softmax);
- batch_size: batch_size values (16, 32, 64, 128);
- loss: employed loss functions (Mean Squared Error - MSE, CategoricalCrossentropy, SparseCategoricalFocalLoss).

The following tables show the excellent statistics metrics derived by applying 70/30 criteria on the involved datasets. Tabs 3.6, 3.8, 3.10, and 3.12 summarize the achieved results, while Tabs. 3.5, 3.7, 3.9, and 3.11 report the corresponding confusion matrices. Finally, Tab. 3.13 summarizes the average results, while Tab. 3.14 those derived by using the 60/40 criteria.

	DKFu	Dowgin	FakeInst	GinM	Plankton
DKFu	611	40	0	1	10
Dowgin	17	1003	0	0	0
FakeInst	0	0	1236	0	0
GinM	0	0	0	614	10
Plankton	24	0	0	22	569

Table 3.5: Multi-Class Confusion Matrix related to Dataset1.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
DKFu	0.9779	0.9852	0.9371	0.9230	0.9300	0.9541
Dowgin	0.9861	0.9943	0.9616	0.9833	0.9724	0.9888
FakeInst	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
GinM	0.9919	0.9971	0.9639	0.9840	0.9738	0.9906
Plankton	0.9844	0.9875	0.9660	0.9252	0.9452	0.9564
Avg.	0.9881	0.9927	0.9656	0.9631	0.9643	0.9780

Table 3.6: Performance metrics related to Dataset1.

	DKFu	Dowgin	FakeInst	GinM	Plankton
DKFu	1151	28	0	1	26
Dowgin	23	2368	0	0	19
FakeInst	0	0	1195	0	0
GinM	60	2	0	1164	0
Plankton	1	29	0	0	1159

Table 3.7: Multi-Class Confusion Matrix related to Dataset2.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
DKFu	0.9805	0.9906	0.9320	0.9544	0.9431	0.9725
Dowgin	0.9859	0.9917	0.9757	0.9826	0.9791	0.9872
FakeInst	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
GinM	0.9910	0.9896	0.9991	0.9494	0.9737	0.9695
Plankton	0.9895	0.9948	0.9626	0.9748	0.9687	0.9848
Avg.	0.9894	0.9932	0.9739	0.9722	0.9729	0.9828

Table 3.8: Performance metrics related to Dataset2.

	DKFu	Dowgin	FakeInst	GinM	Plankton
DKFu	598	11	0	0	6
Dowgin	16	1135	0	0	0
FakeInst	0	0	619	0	0
GinM	0	1	0	578	13
Plankton	81	0	0	0	551

Table 3.9: Multi-Class Confusion Matrix related to Dataset3.

3.4.4 *Comparison and Discussion*

To highlight the potentialities of the presented approach, I compared the achieved results with those derived by the most famous ML-based methods provided by WEKA [116], namely: the Multi-Layer Perceptron (MLP) classifier, J48 Trees (J48) algorithm, and Naive Bayes (NB) classifier, respectively. I accomplished this by arranging the API-Streams as corresponding flattened sequences.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
DKFu	0.9683	0.9940	0.8604	0.9724	0.9130	0.9832
Dowgin	0.9919	0.9931	0.9895	0.9861	0.9878	0.9896
FakeInst	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
GinM	0.9960	0.9951	1.0000	0.9764	0.9880	0.9858
Plankton	0.9721	0.9731	0.9667	0.8718	0.9168	0.9225
Avg.	0.9857	0.9911	0.9633	0.9613	0.9611	0.9761

Table 3.10: Performance metrics related to Dataset3.

	DKFu	Dowgin	FakeInst	GinM	Plankton
DKFu	1185	3	0	5	82
Dowgin	0	1279	0	0	0
FakeInst	0	0	1204	11	0
GinM	0	7	0	1223	0
Plankton	17	61	0	0	1091

Table 3.11: Multi-Class Confusion Matrix related to Dataset4.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
DKFu	0.9827	0.9964	0.9859	0.9294	0.9568	0.9630
Dowgin	0.9885	0.9855	0.9474	1.0000	0.9730	0.9926
FakeInst	0.9981	1.0000	1.0000	0.9909	0.9955	0.9955
GinM	0.9963	0.9968	0.9871	0.9943	0.9907	0.9954
Plankton	0.9741	0.9836	0.9301	0.9333	0.9317	0.9583
Avg.	0.9880	0.9925	0.9701	0.9696	0.9695	0.9808

Table 3.12: Performance metrics related to Dataset4.

However, due to the high number of considered features, any relevant results have not been obtained.

Next, I compared the proposed model with the most famous state-of-the-art static-based approaches. In particular, I considered results achieved by Xie et al. [118] with the Random Forest algorithm (Xie-RF), results obtained by Li et al. [56] with a DNN (Li-DNN), and Linear Learning Algorithm-related results

	Acc.	Spec.	Prec.	Sens.	F-Score	AUC
Dataset1	0.9881	0.9927	0.9656	0.9631	0.9643	0.9780
Dataset2	0.9894	0.9932	0.9739	0.9722	0.9729	0.9828
Dataset3	0.9857	0.9911	0.9633	0.9613	0.9611	0.9761
Dataset4	0.9880	0.9925	0.9701	0.9696	0.9695	0.9808
Avg.	0.9878	0.9924	0.9681	0.9666	0.9670	0.9793

Table 3.13: Average metrics values related to 70/30 criteria.

	Acc.	Spec.	Prec.	Sens.	F-Score	AUC
Dataset1	0.9874	0.9923	0.9634	0.9599	0.9610	0.9761
Dataset2	0.9891	0.9934	0.9694	0.9760	0.9723	0.9847
Dataset3	0.9890	0.9933	0.9695	0.9766	0.9727	0.9850
Dataset4	0.9803	0.9876	0.9525	0.9514	0.9518	0.9695
Avg.	0.9865	0.9917	0.9637	0.9660	0.9645	0.9789

Table 3.14: Average metrics values related to 60/40 criteria.

achieved by Aonzo et al. [6] (Ao-LLA). Tab. 3.15 summarizes the comparison between the proposed Stacked Neural Network (Pr-SNN) and these static-based approaches.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
Li-DNN	0.9925	0.9945	0.9961	0.9904	0.9933	0.9925
Ao-LLA	0.9890	0.9900	0.9900	0.9880	0.9890	0.9890
Pr-SSN	0.9878	0.9924	0.9681	0.9666	0.9670	0.9793
Xie-RF	0.9770	0.9992	0.9775	0.9775	0.9775	0.9884

Table 3.15: Comparison with static-based solutions (only Avg. values are reported).

As reported in Tab. 3.15, the following comparison shows that the Pr-SSN achieved an average accuracy equivalent to those derived by the considered approaches. Also, the Pr-SSN obtained only a 2% less concerning the remaining statistic metrics related to Li-DDN and Ao-LLA, respectively. However, these solutions are based on source code, and consequently, they become inef-

fective against obfuscation techniques. Therefore, this proves the effectiveness of the proposed approach against static-based ones.

Also, I compared the Pr-SNN with the most famous dynamic DL-based solutions. More precisely, I considered results obtained by Abderrahmane et al. [1] with a CNN (Ab-CNN) and results reported in Chap. 2, which have been derived using a RNN (C1-RNN) and a CNN (C1-CNN), respectively. Tab. 3.16 summarizes the comparison between the Pr-SNN and these dynamic-based approaches.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
C1-RNN	0.9995	0.9997	0.9987	0.9986	0.9986	0.9993
C1-CNN	0.9984	0.9990	0.9963	0.9960	0.9961	1.0000
Pr-SSN	0.9878	0.9924	0.9681	0.9666	0.9670	0.9793
Ab-CNN	0.9330	0.9414	0.9410	0.9780	0.9600	0.9560

Table 3.16: Comparison with dynamic DL-based solutions (only Avg. values are reported).

As reported in Tab. 3.16, the following comparison shows that the Pr-SNN outperformed the Ab-CNN with an improvement of 5% in average accuracy. Notice that, the evaluation metrics of Ab-CNN have been derived by only considering a matrix representation of system calls. Consequently, this approach cannot achieve equivalent results as those derived by the Pr-SNN. Instead, C1-RNN and C1-CNN achieved the best results with an improvement of 1% in average accuracy. However, the Pr-SNN is able to classify malware applications by considering their behavior as API-Streams. Also, since the achieved results are slightly lower than those based on the API-Images, API-Streams might be a valid approach to minimize the damages caused at Run-time.

Finally, to provide a graphical interpretation of how the usage of CNN LSTM-SAE affects the features transformation, I visualized the high dimensional latent space, of each involved testing set, on a two-dimensional plane. I accomplished this by using the t-Distributed Stochastic Neighbor Embedding (t-SNE) [62, 114]. More precisely, t-SNE is a famous approach for non-linear data dimensionality reduction that is often employed to allow data

visualization on a two or three-dimensional plane. Figs. 3.6, 3.7, 3.8, and 3.9 show the obtained t-SNE representations.

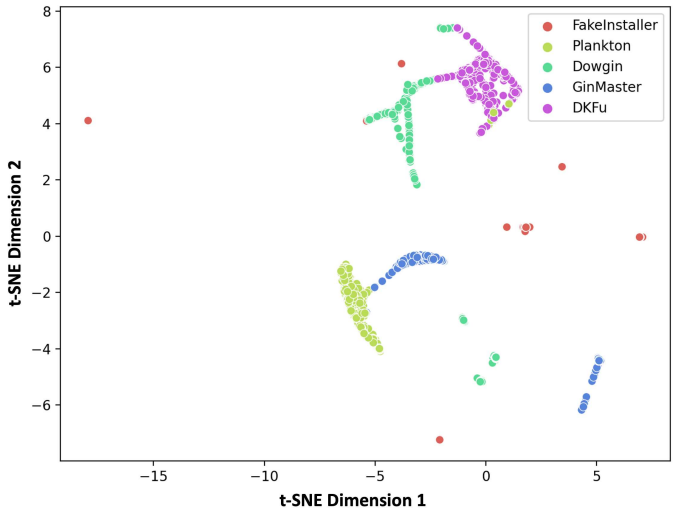

Figure 3.6: t-SNE representation related to Dataset1.

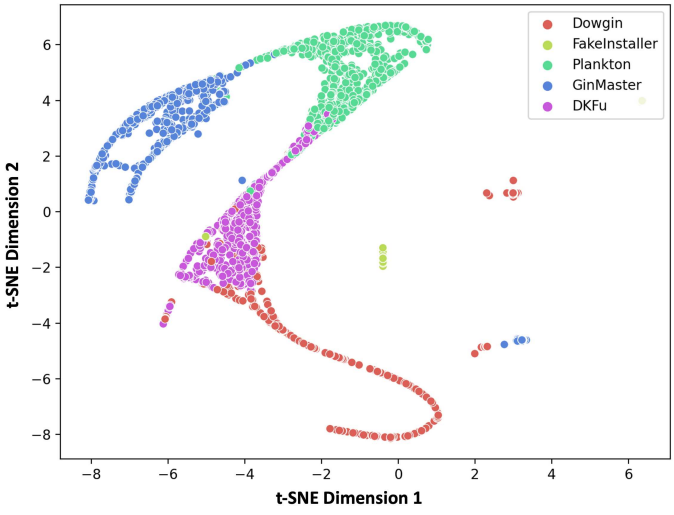

Figure 3.7: t-SNE representation related to Dataset2.

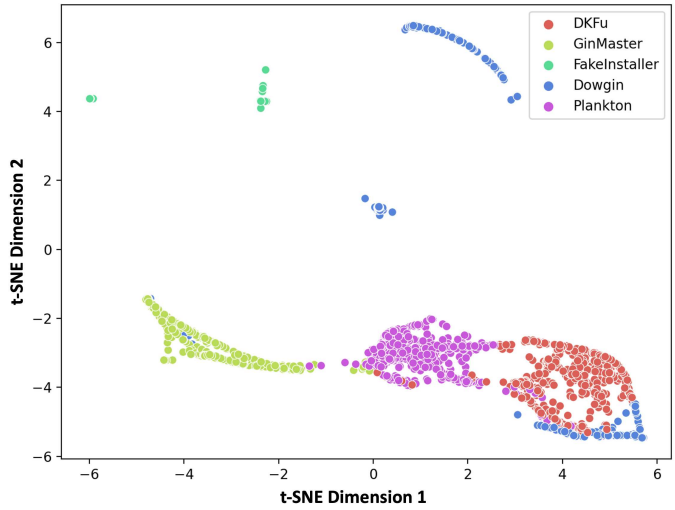

Figure 3.8: t-SNE representation related to Dataset3.

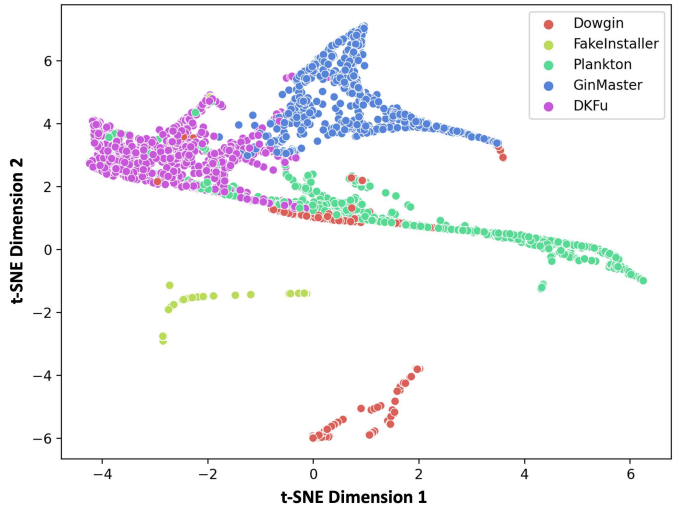

Figure 3.9: t-SNE representation related to Dataset4.

As depicted in the following figures, the analyzed latent vectors have been clustered into five well-visible groups. Therefore, the employed CNN-LSTM-SAE is able to derive relevant features

from the input API-Streams by limiting the overlapping among considered classes. More precisely, FakeInstaller is represented by a unique group, while some overlaps are present among the other groups, as confirmed by the results reported in Tabs 3.5, 3.7, 3.9, and 3.11. Finally, some families, like DKFu and Dowgin, are represented by at least two sub-clusters. Consequently, this might prove the presence of several sub-categories, also called variants, characterized by a different behavior over time.

3.5 CONCLUSIONS AND FUTURE WORKS

In this Chapter, I proposed a novel approach called API-Streams to detect Android malware applications during their execution. More precisely, I combined the capability of Autoencoders in finding relevant features, the goal of Video-Classification in distinguishing objects from a stream of frames, and the classification abilities of DNNs. To accomplish this, I first considered the dynamic behavior, represented as streams of API-Images, related to five representative Android families of the Unisa Malware Dataset (UMD). Then, I investigated the capability of CNN-LSTM Sparse Autoencoders (CNN-LSTM-SAEs) in learning relevant features from the considered behavior. The achieved results, also compared with those derived by state-of-the-art static and dynamic-based approaches, have proven the effectiveness of the proposed model by obtaining an average accuracy of 98% in the presence of several unbalanced training datasets. Finally, I deeply analyzed the abilities of the employed Autoencoder to correctly classify applications in the related family by providing the t-Distributed Stochastic Neighbor Embedding (t-SNE) graphical representation of the considered API-Streams. For each of them, the corresponding latent vectors have clustered in five well-visible groups by confirming the capability of the employed configuration in deriving meaningful features for the Run-time malware classification goal.

However, since some malware families have been grouped into at least two sub-clusters, several malware sub-categories might be characterized by a distinct behavior over time. For this reason, this study proposes two possible future works. First, I will investigate the effectiveness of API-Streams and CNN-

LSTM-SAEs by considering other malware categories. I focused only on five Android malware families because the API-Streams generation process is time-consuming. Second, I will investigate the abilities of Autoencoders in detecting malware sub-categories by considering their behaviour over time. More precisely, the following studies might improve the effectiveness of Run-time malware detection by proposing enhanced and unsupervised DL-based approaches, respectively.

4

A FEDERATED APPROACH TO ANDROID MALWARE CLASSIFICATION THROUGH PERM-MAPS

4.1 INTRODUCTION

Since Android-based devices are used by thousands of end-users every year, more and more malicious applications are continuously developed by cyber-criminals in order to steal sensitive information and conduct hostile activities. According to the last McAfee Mobile Threat Reports [68–70], cyber-criminals increased the effectiveness of their activities with the support of a wide variety of methods, such as back-doors and fake cryptocurrencies. Consequently, as also shown in Fig. 4.1, the number of malicious applications is drastically increased by overcoming 40 million and growing by at least 4 million per year.

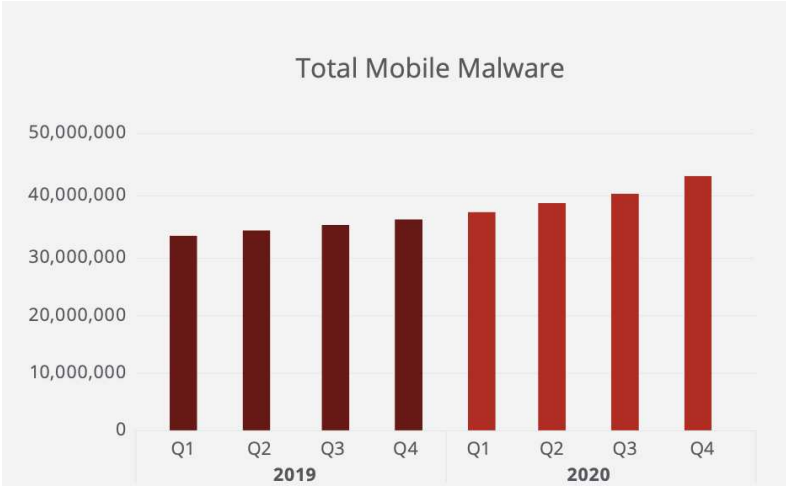

Figure 4.1: Malware applications detected by quarter in 2019 and 2020 [69].

Therefore, to face the following trend, several Machine Learning (ML) and Deep Learning (DL) approaches have proved to be

effective in tackling many aspects related to Android threats, especially when they have been combined with static and dynamic features directly extracted from mobile applications [33, 45, 66]. However, due to the continuous release of Android malware, the related classification tasks are still challenging. Consequently, many state-of-the-art approaches suffer from problems related to dynamic re-training, as well as the updating of their training datasets.

For this reason, the main goal of this chapter aims to propose a novel feature representation technique, called Permission Maps (Perm-Maps)[1], to classify the different malware families by combining Android permissions and their severity levels. Therefore, the Perm-Maps workflow is first discussed and then embedded in a Federated architecture in which end-users build the detection model by sharing their analyzed applications. Next, the effectiveness of the proposed approach is proven using a Convolutional Neural Network (CNN) and compared with the most popular ML and DL-based solutions. Finally, a feature selection technique based on the most frequent permissions is investigated in order to reduce the computational effort required to build the related federated malware classifier.

Hence, the main contributions of this chapter can be summarized as follows:

1. A novel feature representation technique, called Perm-Maps, is proposed in order to exploit the effectiveness of the Android permissions and their security levels arranged as two-dimensional matrices;

2. A Federated architecture is presented to support the detection model-related building process through the Perm-Maps and end-users cooperation, respectively;

3. A Convolutional Neural Network is employed to classify several Android malware families and then compared with the most famous ML and DL-based solutions;

4. A feature selection technique based on the most frequent Android permissions is investigated to reduce the required computational effort.

1 Article published in [25]

The remainder of the chapter is organized as follows. Sec. 4.2 will present a preliminary overview of Android Permissions and their severity levels. Next, Sec. 4.3 will show the Perm-Maps generation workflow, while Sec. 4.4 will describe the employed Federated architecture. Finally, Sec. 4.5 will discuss the experimental results, while Sec. 4.6 will show the conclusions and future work.

4.2 BACKGROUND

In this Section, some key concepts related to Android permissions are reported to understand and better appreciate the novelties of the proposed approach.

4.2.1 *Android Permissions*

At high level, Android permissions can be categorized as *Default* and *Custom*. Also, depending on their purpose, they are characterized by a protection level flag (or severity level flag) set up with several values known as normal, signature, and dangerous [4]. For this reason, there are 3 Android permissions typologies: *Install-time*, *Runtime*, and *Special* [3]. Install-time permissions grant an application limited access to data and perform restricted actions that minimally affect the system or other apps. Hence, when Install-time permission is declared, the Android OS automatically grants the required permission without notifying the end user. There are two types of Install-time permissions called *Normal permissions* and *Signature permissions*:

- *Normal permissions*: allow access to data and actions that present a minimal risk for the system or end-users privacy. They can be used or identified through a protection level set to normal;
- *Signature permissions*: since they are defined in another application, the signature permissions are granted if, and only if, the requesting application and declarant application are signed through the same certificate. They can be used or identified through a protection level set to signed.

Runtime permissions, also known as *Dangerous permissions*, grant an application additional access to restricted data by allowing it to perform actions that substantially affect the system or other apps. When an application requests Runtime permission, the system shows a prompt and waits whether it is granted or not by the end-user. Runtime permissions can be identified through a protection level set to dangerous.

Finally, Special permissions can be only defined by the Original Equipment Manufacturers (OEMs). They provide access control concerning several energy-intensive actions, such as access to other applications. More precisely, they are closely associated with an app operation (app op) related to access control, and they can be used or identified through a protection level set to *appop*.

4.3 PERMISSION MAPS

Although most of the techniques used in literature include both static and dynamic approaches, the static one is the most desired because it can analyze the applications without running them. Accordingly, this Section proposes a new static-based feature representation technique called Permission Map (Perm-Map). More precisely, A Perm-Map is a sparse matrix where Android permissions, and their corresponding severity levels, are related among them as fixed points and arranged in a two-dimensional plane. As depicted in the following, the proposed Perm-Maps can address 2 main issues:

1) Since Default and Custom permissions have different severity levels, a malicious developer could define some low severity level permissions (e.g. Normal permissions) to perform several hostile activities without notifying the end user, such as theft of sensitive data or launch of cyber-attacks;

2) Since Perm-Maps represent static features only extracted from the manifest file, they cannot be influenced by the most famous obfuscator tools, like DexGuard [37], ProGuard [38], and Obfuscapk [5].

4.3.1 *Perm-Map creation workflow*

The creation of a Perm-Map consists mainly of the following 4 steps:

1. Extraction of the Android permissions and their corresponding protection level;
2. Assignment of an identifier (ID^p) to any Android permission;
3. Assignment of an identifier (ID^s) to any severity level;
4. Creation of the Perm-Maps by using pairs of IDs (ID^p; ID^s) as coordinates of fixed points in a two-dimensional plane.

The first three steps can be accomplished using several tools or libraries devoted to the malware static analysis. A typical approach, supported by the Android documentation, could envisage the usage of two dictionaries to collect the well-known Android permissions and their severity levels with corresponding identifiers ID^p and ID^s, respectively. Also, the <*permission*> tag can be employed to know the protection level of Custom permissions. This approach is adopted by several most famous reverse engineering tools, like Androguard [28]. More precisely, for each permission declared in the Manifest file, the tool finds the corresponding severity level if the considered permission is known. Otherwise, it assigns a dangerous protection level. Finally, for each analyzed application, the fourth step is accomplished by using each pair (ID^p; ID^s) to report a fixed point in a two-dimensional plane. For instance, let p_1 and p_2 be two Android permissions and let s_3 and s_2 be their security levels, respectively. On the basis of described process, we can consider two pairs of coordinates $C1 = (p_1, s_3)$ and $C2 = (p_2, s_2)$ and draw the corresponding points in a two-dimensional plane. Also, since severity levels could be different among them, we can highlight these differences using different colour scales (e.g. RGB or Gray-scale). Fig. 4.2 shows the complete workflow to obtain a Perm-Map.

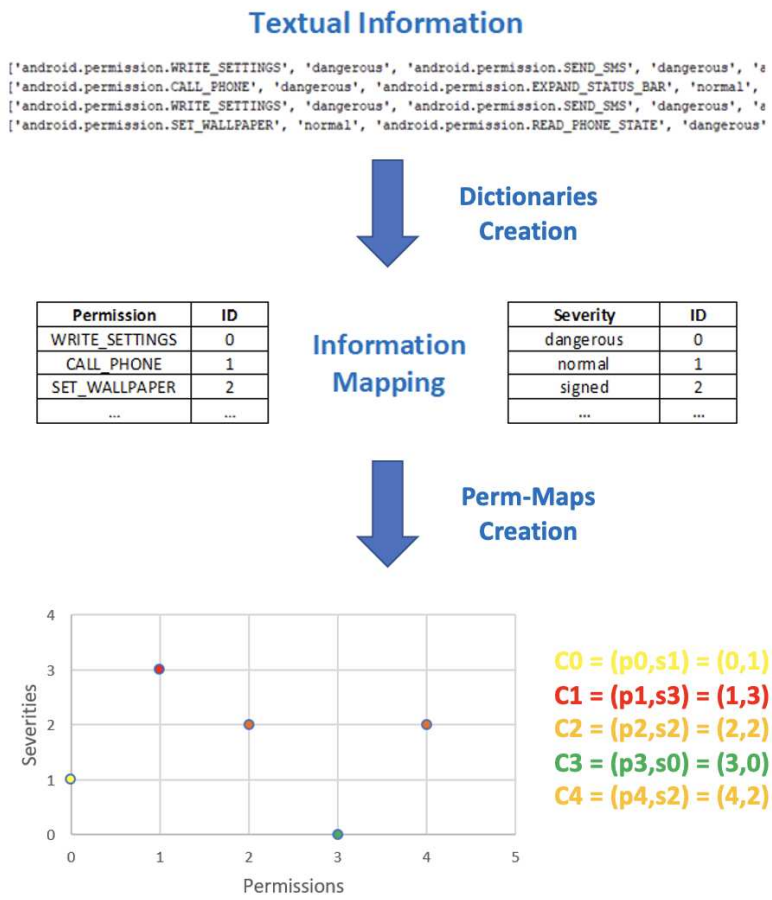

Figure 4.2: Perm-Maps workflow.

4.4 THE PROPOSED ARCHITECTURE

Since millions of Android-based applications are released every year, managing related datasets is a process that requires much effort, such as accessing, searching, and updating them. To overcome these issues, this Section presents a Federated Architecture to support Android malware classification through the proposed Perm-Maps. More precisely, since such architectures employ a Federated Logic, the end-user devices are forced to share their data with a centralized infrastructure typically devoted to providing services [48]. For instance, thanks to its great success in the last decade, such logic has been applied to face several issues

related to the convergence process among Edge and Cloud infrastructures, such as data aggregation [16], data mobility [14], and services migration [98]. Also, it has been involved in many other famous application domains, such as Cryptography solutions to preserve data security [92], optimization frameworks for the Medical of Things Devices [95], and Vehicular Networks optimization [112].

For this reason, the proposed architecture aims to provide a data aggregation workflow in which federated devices share their decentralized data with a server devoted to managing the related classification model. Therefore, the main contributions of the presented architecture, described through the *Model Creation* and *Model Update* processes, can be summarized as follows:

1. An aggregation workflow is presented to collect decentralized data from federated devices;
2. The resultant centralized dataset is employed to create a shared CNN model based on proposed Perm-Maps;
3. A data update workflow is discussed to manage centralized data in order to re-adapt and share the new model.

4.4.1 *Model Creation process*

At beginning of the Model Creation process, each device decompresses the apk file and sends the Manifest to the centralized server. Thus, once all the data are received, the server generates the Perm-Maps following the workflow shown in Fig. 4.2. Next, the server trains the CNN and sends the related model to each device. Finally, end-user devices use the model to receive a notification about the classification activity. Fig. 4.3 shows the discussed process, while its main steps can be summarized as follows:

1. End-user devices decompress the apk file;
2. Devices send the Manifest files to the central server;
3. The server generates the Perm-Maps once all data are received;

4. The server trains and tests the CNN;
5. The server sends the related model to each device;
6. Devices receive a notification about the classification activity.

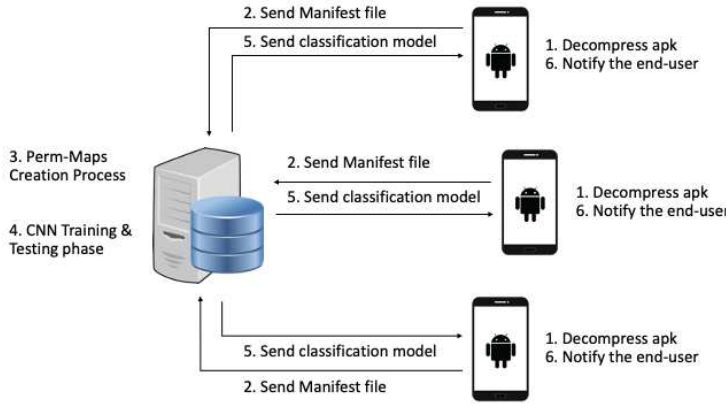

Figure 4.3: Model Creation Process.

4.4.2 *Model Update process*

The following process is responsible for collecting new data when an end-user installs a new application. At a high level, it differs from the Model Creation process in 3 main aspects:

1. If an application is unknown, it automatically stores the related manifest file on the centralized server;
2. If an application is unknown, it considers the end-users feedback to generate a classification label;
3. If a threshold value is reached, it trains and shares the updated CNN by considering new data.

Therefore, when an end-user installs an application, the device decompresses the apk file, extracts the Perm-Map by reading the Manifest, and employs the classification model. If the application

is known, the model provides the achieved prediction. Otherwise, it asks if the application is unknown or trusted and sends the Manifest file and the user's answer to the centralized server. Next, the server stores new data and, once the dataset size reaches a threshold value, generates the Perm-Maps. Finally, the server re-trains the CNN and sends the updated model to each device. Fig. 4.4 shows the discussed process, while its main steps can be summarized as follows:

1. End-user devices decompress the apk file;
2. Devices extract the Perm-Map from the Manifest file;
3. Devices ask if the application is unknown or trusted;
4. Devices send the Manifest and the user's answer to the server;
5. The server stores new data;
6. The server generates the Perm-Maps once the dataset size reaches a threshold value;
7. The server trains and tests the CNN-related model;
8. Finally, the server sends the updated model to each device.

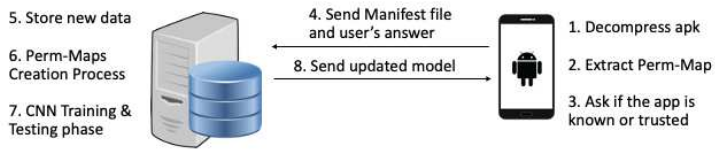

Figure 4.4: Model Update Process.

4.5 EXPERIMENTAL RESULTS

The goal of the presented experiments is devoted to demonstrating the effectiveness of Perm-Maps in classifying Android applications. To accomplish this, I show the abilities of a CNN in performing such classification tasks using an unbalanced dataset. Also, I analyze the effectiveness of a features selection technique to reduce the computational effort required by Perm-Map generation and CNN training processes, respectively.

4.5.1 *Dataset and Experimental setting*

The Perm-Maps dataset considered in the following experiments has been composed of Goodware (GW) applications and 9 famous Android families of the Unisa Malware Dataset (UMD), namely Adrd, Boqx, FakeDoc (FakeD), Fusob, GinMaster (GinM), Iconosys (Isys), Kmin, Lotoor and Mseg. Hence, to simulate the discussed Model Creation Process, each application has been analyzed through the Android Device Cross-Platform mode of CuckooDroid [11, 12]. More precisely, I employed 2 Android Guest virtual machines to decompress each apk file and send the related Manifest file to the Server machine. Thus, by following the workflow shown in Fig. 4.2, I generated the Perm-Maps matrices (4×298) in accordance with the maximum number of severity levels and Android permissions observed, respectively. Next, I split the whole dataset into two mutually exclusive subsets called learning and testing datasets. I used 70% of the entire dataset for learning and the remaining 30% for testing. Also, the K-Fold cross-validation algorithm, with k=10 as recommended value [10], has been employed to tune the hyper-parameters and provide an unbiased evaluation of the employed CNN. Finally, in order to analyze the applications' distribution for each category, I performed an Exploratory Data Analysis (EDA) on the involved dataset [88, 128]. The obtained results, shown in Fig. 4.5 and Tab. 4.1, highlight the unbalanced nature of such dataset.

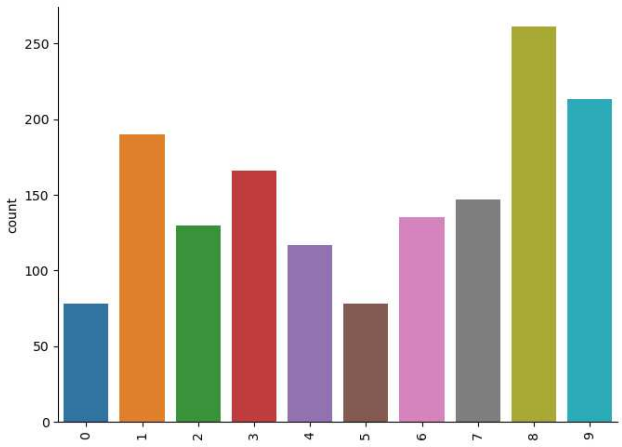

Figure 4.5: Exploratory Data Analysis (EDA) on the involved dataset.

	Training	Testing	Total
Adrd	56	22	78
Boqx	135	55	190
FakeD	99	31	130
Fusob	106	60	166
GinM	82	35	117
GW	55	23	78
Isys	101	34	135
Kmin	104	43	147
Lotoor	183	78	261
Mseg	145	68	213
Total	1066	449	1515

Table 4.1: Summary of the involved dataset.

4.5.2 *Evaluation Metrics*

To appreciate the classification quality of the employed DNNs, I used the following evaluation metrics derived from the multi-class confusion matrix: Accuracy (Acc.), Sensitivity (Sens.), Specificity (Spec.), Precision (Prec.), F-Score (F-Mea.), and Area Under the ROC Curve (AUC).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

$$Sensitivity = \frac{TP}{TP + FN} \tag{4.2}$$

$$Specificity = \frac{TN}{TN + FP} \tag{4.3}$$

$$Precision = \frac{TP}{TP + FP} \tag{4.4}$$

$$F - Score = \frac{2 * Sens * Prec}{Sens + Prec} \tag{4.5}$$

$$AUC = \frac{Sens + Spec}{2} \tag{4.6}$$

For each category, TPs (True Positives) refer to the applications correctly classified, while TNs (True Negatives) refer to the applications correctly identified in another category. Conversely, FPs (False Positives) are the applications mistakenly identified as the considered category, while FNs (False Negatives) are the applications mistakenly identified in another category. Also, to achieve a global perspective of the detector effectiveness, the average performance values (Avg.) among all the observed malware classes have been derived.

4.5.3 *Network description and Results*

As mentioned earlier, I used a CNN to classify several Android applications into the related belonging categories. Accordingly, in this Section, the related implementation details are first reported. Then, the experimental results are shown. Notice that I derived the described architecture according to the results derived by using the 70/30 criteria and the K-Fold cross-validation algorithm, respectively.

Fig. 4.6 summarizes the high-level organization of the employed CNN, which is first composed of two Conv2D layers with activation=relu, no pooling functions, and kernel_size=(2,2). More precisely, I used eight filters and strides=(2,2) for the first layer and two filters and strides=(1,1) for the second one, respectively. After that, a Flatten layer is employed to map the extracted features as one-dimensional latent vectors and fed a fully con-

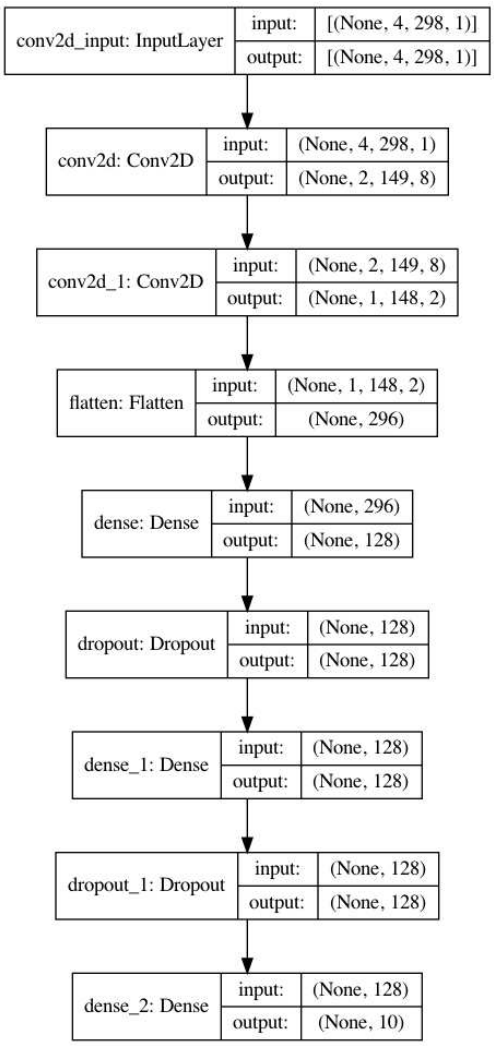

Figure 4.6: Architecture of the employed CNN.

nected softmax network. Hence, I used two Dense layers having activation=relu, dropout=0.5, and 128 neurons, respectively. In addition, to consider the classification results as probability distributions, I employed a third dense layer with ten neurons and activation=softmax as the output layer. Therefore, I trained the following network with Adam optimizer and the SparseCategoricalCrossentropy loss function for 150 epochs and batch_size=64.

Additionally, the following architecture has been derived by varying the following hyper-parameters:

- numConvLayers: the number of Conv2D layers (1, 2, 3);
- numFilters: the number of filters for each Conv2D layer (2, 4, 8, 16);
- kernel_size: kernel_size values ((2,2), (3,3), (4,4));
- stride: the stride length for each Conv2D layer (1, 2, 4);
- numDenseLayers: the number of Dense layers (1, 2, 3, 4);
- numNeurons: the number of neurons for each Dense layer (10, 32, 64, 128, 256);
- dropout: dropout values for each Dense layer (0.2, 0.3, 0.4, 0.5);
- activation: activation functions used (relu, softmax);
- batch_size: batch_size values (16, 32, 64, 128);
- loss: employed loss functions (CategoricalCrossentropy, SparseCategoricalFocalLoss).

More precisely, Tabs. 4.2 and 4.3 report the Multi-Class Confusion Matrix and the statistics metrics derived by applying the 70/30 criteria, while Tab. 4.4 summarizes the metrics related to the K-Fold cross-validation algorithm with k=10.

Furthermore, to analyze the Update Process and the effectiveness of the proposed approach against new malicious applications, I estimated the data growth range within which to readjust the proposed CNN. More precisely, I reduced the whole dataset through an iterative process. At each step, 5% of data has been removed by employing the new sub-dataset to train and test the proposed CNN. Tab. 4.5 summarizes the classification metrics derived by the testing phase for each considered sub-dataset. More precisely, it highlights that CNN should be re-trained when the data dimensions grow between 15% and 20%. For instance, the comparison between the whole dataset (Size 100%) and the dataset reduced by 20% (Size 80%) shows an average worsening of 3% in Precision, 7% in Sensibility, and 6% in F-Score, respectively.

	Adrd	Boqx	FakeD	Fusob	GinM	GW	Isys	Kmin	Lotoor	Mseg
Adrd	22	0	0	0	0	0	0	0	0	0
Boqx	0	55	0	0	0	0	0	0	0	0
FakeD	0	0	31	0	0	0	0	0	0	0
Fusob	0	0	0	60	0	0	0	0	0	0
GinM	0	0	0	0	35	0	0	0	1	0
GW	0	0	0	0	0	23	0	0	0	0
Isys	0	0	0	0	0	0	34	0	1	0
Kmin	0	0	0	0	0	0	0	43	0	0
Lotoor	0	4	0	0	0	0	0	0	78	0
Mseg	0	0	0	0	0	0	0	0	0	68

Table 4.2: Multi-Class Confusion Matrix related to 70/30 criteria.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
Adrd	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Boqx	0.9912	0.9898	0.9322	1.0000	0.9649	0.9949
FakeD	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Fusob	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
GinM	0.9978	0.9976	1.0000	0.9722	0.9859	0.9849
GW	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Isys	0.9978	1.0000	1.0000	0.9714	0.9855	0.9857
Kmin	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Lotoor	0.9867	0.9945	0.9750	0.9512	0.9630	0.9728
Mseg	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Avg.	0.9974	0.9982	0.9906	0.9895	0.9898	0.9937

Table 4.3: Performance metrics related to 70/30 criteria.

4.5.4 *Comparison and Discussion*

To highlight the potentialities of the presented approach, I compared the achieved results with those derived by the most famous ML-based methods provided by WEKA [116], namely: the J48 Trees (J48) algorithm, Naive Bayes (NB) classifier, and Multi-Layer Perceptron (MLP) classifier, respectively. I accomplished this by arranging the Perm-Maps as corresponding flattened sequences.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
Fold 1	0.9960	0.9977	0.9830	0.9757	0.9789	0.9867
Fold 2	0.9956	0.9974	0.9839	0.9698	0.9762	0.9836
Fold 3	0.9965	0.9979	0.9871	0.9754	0.9809	0.9867
Fold 4	0.9965	0.9980	0.9859	0.9768	0.9810	0.9874
Fold 5	0.9965	0.9979	0.9869	0.9736	0.9798	0.9858
Fold 6	0.9960	0.9977	0.9838	0.9737	0.9783	0.9857
Fold 7	0.9952	0.9973	0.9739	0.9700	0.9717	0.9836
Fold 8	0.9947	0.9969	0.9797	0.9600	0.9688	0.9785
Fold 9	0.9956	0.9974	0.9849	0.9720	0.9780	0.9847
Fold 10	0.9960	0.9977	0.9835	0.9731	0.9775	0.9854
Avg.	0.9959	0.9976	0.9833	0.9719	0.9770	0.9847

Table 4.4: Performance metrics related to K-Fold k=10.

Size	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
100%	0.9974	0.9982	0.9906	0.9895	0.9898	0.9937
95%	0.9977	0.9987	0.9853	0.9894	0.9872	0.9940
90%	0.9949	0.9970	0.9786	0.9667	0.9711	0.9819
85%	0.9914	0.9949	0.9666	0.9419	0.9529	0.9684
80%	0.9889	0.9936	0.9623	0.9116	0.9246	0.9526
75%	0.9667	0.9790	0.9188	0.7907	0.8132	0.8849

Table 4.5: Performance metrics related to Update Process.

Tab. 4.6 compares the proposed CNN (Pr-CNN) with ML-based methods.

As shown in Tab. 4.6, the MLP classifier, with an 83% average accuracy, is not able to better distinguish application categories by considering Android permissions and their severity levels. Instead, J48 trees and the NB classifier have achieved good results by obtaining an average accuracy of 96%. However, the proposed CNN outperformed such approaches by achieving up to a 3% improvement over the NB classifier and J48 trees, and up to 16% over the Multi-Layer Perceptron classifier. Consequently, the

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
Pr-CNN	0.9974	0.9982	0.9906	0.9895	0.9898	0.9937
J48	0.9670	0.9670	0.9670	0.9680	0.9670	0.9670
NB	0.9647	0.9650	0.9650	0.9670	0.9640	0.9660
MLP	0.8348	0.8350	0.8330	0.8350	0.8340	0.8350

Table 4.6: Comparison between the proposed CNN and ML-based methods (only Avg. values are reported).

involved CNN can reduce the number of FPs and FNs and thus better minimize the classification error.

Also, I compared the proposed CNN with the state-of-art ML and DL-based solutions, namely: the Random Forest results respectively achieved by Kumar et al. (Kum-RF) [52] and Xie et al. (Xie-RF) [118], LSTM neural network results achieved by Vinayakumar et al. (Vi-LSTM) [110], and DNN results obtained by Li et al. (Li-DNN) [56]. Tab. 4.7 summarizes the comparison between the Pr-CNN and such approaches.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
Pr-CNN	0.9974	0.9982	0.9906	0.9895	0.9898	0.9937
Li-DNN	0.9925	0.9945	0.9961	0.9904	0.9933	0.9925
Xie-RF	0.9770	0.9992	0.9775	0.9775	0.9775	0.9884
Kum-RF	0.9100	0.9200	0.9000	0.9300	0.9147	0.9250
Vi-LSTM	0.8970	0.6280	0.9100	0.9600	0.9147	0.7690

Table 4.7: Comparison between the proposed CNN and state-of-art solutions (only Avg. values are reported).

As shown in Tab. 4.7, the Vi-LSTM and Kum-RF solutions achieved discrete results and have been outperformed by the proposed CNN, which obtained up to 10% and 8% improvement in average accuracy, respectively. More precisely, the Vi-LSTM evaluation metrics were derived considering only Android permissions translated as numerical information. Instead, the Kum-RF evaluation metrics were derived considering Grayscale images directly generated without performing any code extraction and decompiling operations. Consequently, these static features are

not relevant in order to achieving excellent classification results. On the other hand, Xie-RF and Li-DNN have achieved equivalent results to those obtained by the proposed CNN. However, notice that the proposed representation technique is based only on Android permissions and their severity levels, while Xie-RF and Li-DNN employ Android permissions and Java methods. For this reason, Xie-RF and Li-DNN become ineffective against obfuscation techniques. Finally, Tab. 4.8 reports a final comparison among proposed CNN, ML-based methods of WEKA, and state-of-art solutions.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
Pr-CNN	0.9974	0.9982	0.9906	0.9895	0.9898	0.9937
Li-DNN	0.9925	0.9945	0.9961	0.9904	0.9933	0.9925
Xie-RF	0.9770	0.9992	0.9775	0.9775	0.9775	0.9884
J48	0.9670	0.9670	0.9670	0.9680	0.9670	0.9670
NB	0.9647	0.9650	0.9650	0.9670	0.9640	0.9660
Kum-RF	0.9100	0.9200	0.9000	0.9300	0.9147	0.9250
Vi-LSTM	0.8970	0.6280	0.9100	0.9600	0.9147	0.7690
MLP	0.8348	0.8350	0.8330	0.8350	0.8340	0.8350

Table 4.8: Overview among proposed CNN, ML-based methods of WEKA, and state-of-art solutions (only Avg. values are reported).

4.5.5 *Features Optimization*

Since the number of employed permissions is 298, the final goal of the experimental phase is devoted to reducing the computational effort required by the Perm-Maps generation and CNN training processes, respectively. To this purpose, I explored the effectiveness of a features selection technique based on the most frequent Android permissions. More precisely, I analyzed the related frequency distribution to find the minimum frequency that would be able to reduce the employed permissions and preserve the number of involved applications. Therefore, as a result of this analysis, I considered Android permissions collected at least 50

times from the whole dataset. Consequently, only 57 common permissions on 298 have been chosen for the training and testing phase, respectively. Fig. 4.7 shows the five most required Android permissions.

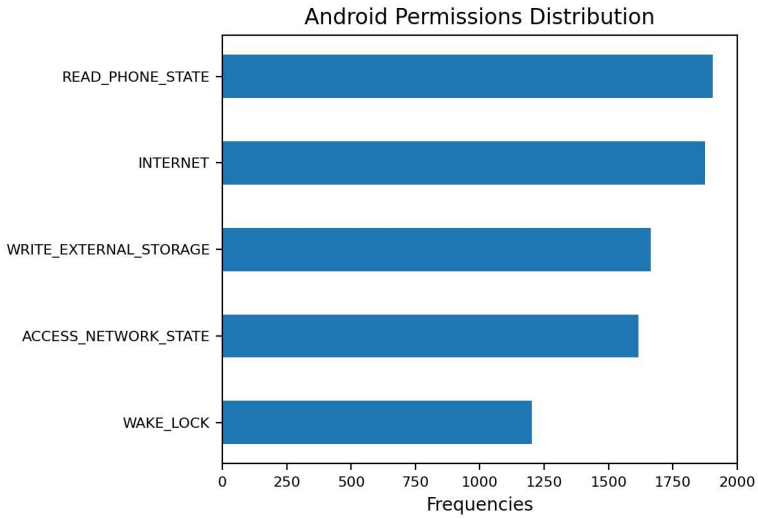

Figure 4.7: Most required Android permissions.

Hence, following the workflow shown in Fig. 4.2, I employed the 57 Android permissions to generate each Perm-Map as a corresponding matrix (4×64) in accordance with the maximum number of severity levels (4) and an over-bound for Android permissions (64). I considered such over-bound to simplify the operations performed by Convolutional layers. Next, in order to run the experiments, I split the new dataset into two mutually exclusive subsets called learning and testing datasets, respectively. I used 70% of the entire dataset for learning and the remaining 30% for testing. The employed neural network, trained with Adam optimizer for 150 epochs and batch_size=64, presents the same architecture shown in Fig. 4.6 except for the input shape=(4,64,1) and Dense layers with dropout=0.45. Finally, the computational effort required for the text substitution, Perm-Maps generation, and training processes has been derived with and without considering the employed selection technique. Tab. 4.9 reports the

computational effort required for each analyzed phase, Tab. 4.10 shows the achieved results, while Tab. 4.11 summarizes the comparison between the employed CNNs. I refer to them with CNN-NoExtraction (CNN-NE) and CNN-WithExtraction (CNN-WE), respectively.

	No Sel. (s)	With Sel. (s)	Diff. (s)
Text Sub.	0.255570	0.152351	0.103219
Perm-Maps	0.058101	0.046116	0.011985
Training	11.589643	8.073685	3.515958
Total	11.903314	8.272152	3.631162

Table 4.9: Required computational effort.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
Adrd	0.9978	1.0000	1.0000	0.9500	0.9744	0.9750
Boqx	0.9956	1.0000	1.0000	0.9688	0.9841	0.9844
FakeD	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Fusob	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
GinM	0.9933	0.9976	0.9773	0.9556	0.9663	0.9766
GW	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Isys	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Kmin	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Lotoor	0.9911	0.9894	0.9524	1.0000	0.9756	0.9947
Mseg	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Avg.	0.9978	0.9987	0.9930	0.9874	0.9900	0.9931

Table 4.10: Performance metrics related to the features section.

	Acc.	Spec.	Prec.	Sens.	F-Mea.	AUC
CNN-NE	0.9974	0.9982	0.9906	0.9895	0.9898	0.9937
CNN-WE	0.9978	0.9987	0.9930	0.9874	0.9900	0.9931

Table 4.11: Comparison between the proposed CNNs.

The obtained results show that the employed selection technique might reduce the computational effort required by each analyzed process. More precisely, as reported in Tab. 4.9, the total effort has been improved by 3.6 seconds. Finally, the comparison shown in Tab. 4.11 demonstrates that proposed CNNs obtained equivalent evaluation metrics. Therefore, the employed selection technique might also optimize the proposed representation approach by drastically reducing the number of employed Android permissions.

4.6 CONCLUSIONS AND FUTURE WORKS

In this Chapter, I proposed a novel approach called Permission Maps (Perm-Maps) to classify Android applications using static features, namely permissions and their corresponding severity levels. Therefore, I first proved the effectiveness of Perm-Maps employing a CNN enhanced by a federated learning process, in which end-users extract static features locally and send them to a centralized server devoted to training the employed network. The achieved results, also compared with those derived using ML-based approaches, proved the effectiveness of the proposed Perm-Maps. More precisely, the employed CNN archived up to a 3% improvement in average accuracy over the Naive Bayes classifier and J48 trees, and up to 16% over the Multi-Layer Perceptron classifier, respectively. Finally, I explored a feature selection technique in order to reduce required to build the related federated malware classifier. The reported results have shown that using the most frequent permissions can reduce such effort by achieving equivalent results.

However, due to the high number of existing Android malware categories, this study proposes two possible future works. First, the effectiveness of the proposed Perm-Maps could be deeply investigated using a considerable quantity of decentralized data and applying a pure Federated Learning-based training process, respectively. Second, the presented Federated logic might support dynamic-based approaches in detecting more sophisticated and malicious applications, such as zero-day malware. For instance, Chap. 5 presents a malware detector based on associative rules and Markov Chains using the same Federated logic.

5

PRIVACY-PRESERVING MALWARE DETECTION THROUGH FEDERATED MARKOV CHAINS

5.1 INTRODUCTION

The exponential growth of the Internet of Things (IoT) technology, together with the success of the Android OS, caused the explosion of the number of mobile apps developed for many kinds of devices, such as smart TVs, smart watches, refrigerators and other interconnected gadgets that can be easily controlled by using common smartphones. Also, since such smartphones frequently assume the role of gateways for many mission-critical applications, they prove to be one of the main driving forces within IoT ecosystems. However, despite the importance of work done, such devices introduce new cybersecurity issues and risks. In particular, due to the lack of appropriate protection mechanisms on the most famous IoT embedded platforms, the large volume of yearly released malware applications poses challenges that need the designing of detection and classification strategies reliable and effective against the different malware families [76].

According to earlier classification studies [21–23], the intensive usage of concealment and obfuscation strategies is one of the primary factors related to the significant growth of malware targeting IoT devices. However, malware applications usually belong to a family with similar behaviors, implying that most new malware is derived as new versions of already existing malware. Hence, the possibility of developing strategies that can effectively categorize malware depending on its family, regardless of being a variant, appears particularly promising to prevent and control its evolution over time. For this reason, many dynamic analysis-based techniques have been proposed by considering the dynamic behavior described through sequences of system-level API calls. More precisely, these strategies assume that malicious applications might contain a set of well-distinct APIs invoked

more often or in a different order than those called by goodware applications [123].

For instance, Markov chains are one of the most effective cutting-edge strategies that describe the API calls invoked by applications and construct the representative behavioral patterns of particular malware families [33]. They consider the sequence of API calls to model the application-related behavior as a graph in which each node represents a unique API, while each edge represents the transition probability between two APIs. Also, Markov chains-based detectors have been proven resistant to evasion efforts against irrelevant API calls injected throughout malicious code and thus be effective against polymorphic malware [66].

However, building a sufficiently complete knowledge base on emerging malware attacks is currently a slow and challenging process also for state-of-the-art ML and DL-based solutions. In addition, the involved organizations (companies and end-users) are often unwilling to share their data because they are focused on preserving their intellectual property related to their IoT applications and systems.

Therefore, to face these issues, Federated Learning (FL)-based approaches represent a recent privacy-preserving solution, which leverages ML and DL models' capabilities to enhance several classification and detection tasks without sharing data [35, 41, 89]. However, as highlighted in many literature studies [18, 46, 57, 83], non-Independent and Identically Distributed (non-IID) data often adversely affects FL-based models regarding the required training time, convergence, learning processes, and classification results. Also, such strategies are often strongly influenced by the configuration of some additional hyperparameters (e.g. threshold values) that might limit their applicability.

For this reason, the main goal of this chapter aims to propose a federated Markov chains-based detector for IoT malware classification[1]. More precisely, Markov chains and associative rules are employed within a similar logic to that described in Chap. 4, in which users analyze each application and then send the extracted information to a central entity devoted to building the proposed detector. Therefore, such paradigm makes data owners proactive contributors to the related building process,

1 Article submitted to Future Generation Computer Systems

also providing them with a mechanism to timely update the global model without sharing their private raw data. Next, the effectiveness of the proposed strategy, compared with the state-of-the-art ML-based approaches, is analyzed within a realistic IoT scenario. Finally, the required time effort is evaluated by considering several dataset partitions and involved clients. More precisely, I show that the proposed detector obtains comparable time performances in the presence of non-IID data.

Hence, the main contributions of this chapter can be summarized as follows:

1. A federated architecture is presented to support the rules mining process as corresponding Markov chains;
2. The resulting associative rules-based detector is employed in recognizing the different malware families by considering both centralized and decentralized data;
3. A performance study is done to show the effectiveness of the proposed approach in the presence of non-IID data.

The remainder of the chapter is organized as follows. Sec. 5.2 will report a background overview of the employed detector. Next, Sec. 5.3 will describe the proposed Federated architecture. Finally, Sec. 5.4 will discuss the experimental results, while Sec. 5.5 will show the conclusions and future work.

5.2 BACKGROUND

Since a sequence of API calls can be effectively used to model the most representative behavioral features associated with a specific malware application, the background concepts related to the association rules-based detector, presented in [23], are recalled in this Section. More precisely, I first describe the detector-related workflow through a step-by-step example. Then, I provide some mathematical definitions related to the rules pruning phase, which is essential to obtain relevant classification results.

5.2.1 *Association rules-based detector*

The execution flow of a specific application t_m can be represented as a sequence of API calls. As a consequence, rules representing the given application can be extracted in the form $\{API_i \rightarrow API_j\}$, with $API_i \prec API_j$. Note that such rules could include API calls not necessarily contiguous. The number of skipped API calls is considered the "spacing" of the rule. After that, such rules can be associated with nodes of a Markov chain to represent the application as a sequence of independent state transitions [34, 66]. Ultimately, we can describe such an API calls flow (i.e. a specific application) by using a graph, with nodes representing two (not necessarily contiguous) API calls and edges identifying their transition timeline (the sequence of invocation) [33]. In order to mine any possible insight from the API calls sequence, different pairs of API calls (rules) are extracted by varying the spacing.

Hence, with a little abuse of notation needed for simplification purposes, the related training process consists, for each application t_m, of several progressive steps n, with $k \in [1, \ldots, n]$ representing the spacing in terms of positions to be skipped within the API calls sequence.

For each intermediate step k (with $k < n$), all the $k-1$ spaced transitions between two API calls are arranged in a Markov-like chain represented by a graph $G_k^{t_m}$, in which the x-th node, N_x, is defined as follows:

$$N_x = [(API_i \rightarrow API_j), \sigma_x] \tag{5.1}$$

where $(API_i \rightarrow API_j)$ represents the mined rule and σ_x is the related number of occurrences, while the edges represent the transition states.

Next, in the last step ($k = n$), all the graphs are merged into one, namely G^{t_m}, representing the entire execution profile of the application t_m. Also, both the edges and the total number of occurrences of each node x of G^{t_m}, i.e. $\sigma^{t_m}(x)$, are updated. In particular, the occurrences are computed by summing the occurrences σ_x of the same nodes of the previous $(n-1)$ graphs.

For instance, let $t_1 = ADDDBCDD$ and $t_2 = ADDBCCCC$ be the API call sequences of two applications, by assuming only

three steps (n = 3), Figs. 5.1 and 5.2 show the extracted graphs $G_k^{t_m}$ of each analyzed application at time-step $k = 1$ and $k = 2$, respectively. As depicted, at time-step $k = 1$, the transitions considered are contiguous, and no elements are present between two API calls used for mining a rule. Contrary, at the time step $k = 2$, every rule is extracted by skipping an API call.

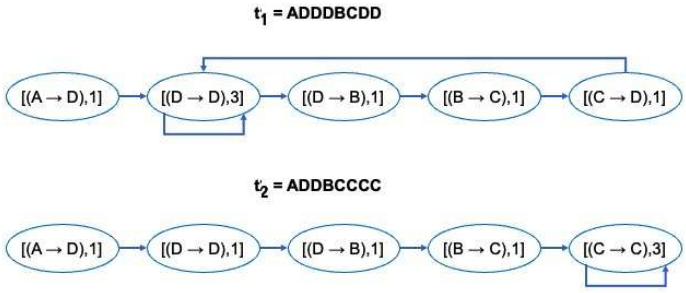

Figure 5.1: Extracted graphs $G_k^{t_1}$ and $G_k^{t_2}$ at time-step $k = 1$.

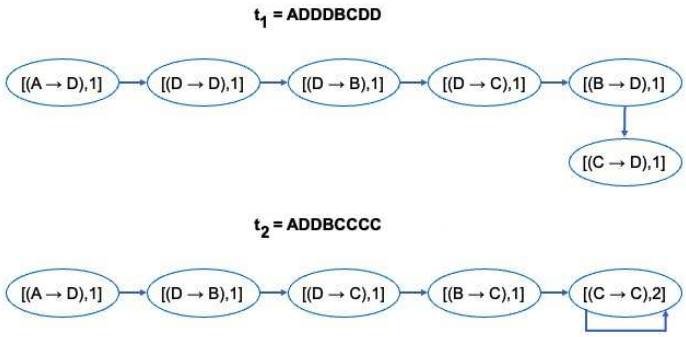

Figure 5.2: Extracted graphs $G_k^{t_1}$ and $G_k^{t_2}$ at time-step $k = 2$.

Next, at time-step $k = 3$, the previous graphs are merged into a new graph G^{t_m} representing the Run-time behavior of a considered application, as shown in Fig. 5.3.

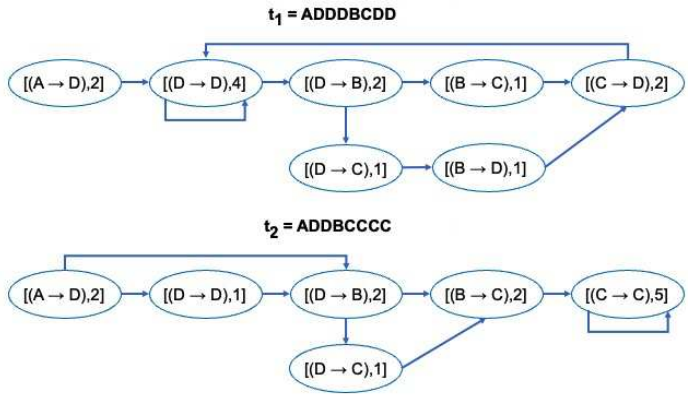

Figure 5.3: Merged graphs G^{t_1} and G^{t_2} related to t_1 and t_2 when $k = 3$.

Finally, since the set of association rules mined from different applications can be used as a signature characterizing a specific malware category (or class) $c \in C$ (where $C = \{c_1, \ldots, c_{|C|}\}$ is the set of malware categories), all the graphs G^{t_m} associated with $\mathcal{N}(c)$ applications belonging to the class c of the training dataset T, are further merged as a final graph G_c, in which the total number of occurrences of each node, σ_c, is again updated as well as the related edges, as shown in Fig. 5.4. Note that, as better explained below (see Equ. (5.3)), σ_c is estimated by taking into account the different lengths (in terms of number of API calls) of the applications. Therefore, the proposed detector is characterized by a time complexity of $O(|T| \times n \times l)$, where $|T|$ is the training dataset dimension, n is the number of rules extraction-related steps, and l is the number of API calls.

5.2.2 *Pruning phase definition*

Since many rules could be extracted from the training process, a pruning step is of paramount importance in order to remove unnecessary information and thus achieve relevant classification results. It is performed by comparing the occurrence of each rule with a threshold value. For instance, Fig. 5.5 shows all the pruned rules (see dashed nodes) if a threshold less than 2 is considered.

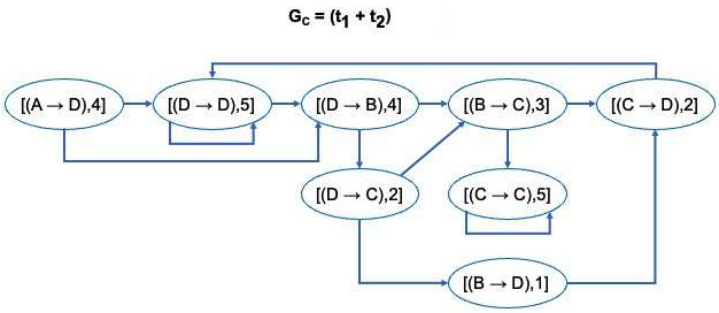

Figure 5.4: Final graph G_c derived by merging t_1 and t_2.

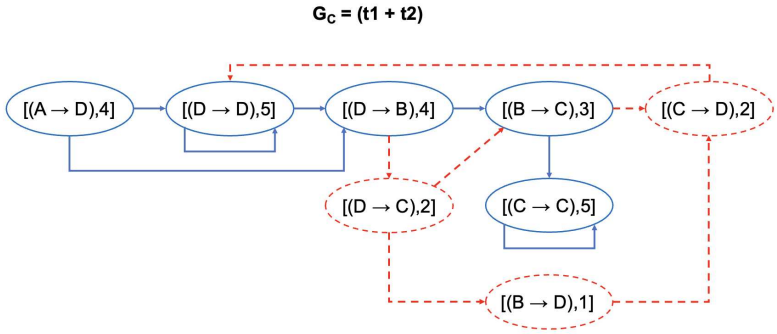

Figure 5.5: Pruning phase of rules with $\sigma_x < 2$.

However, the existence of similar rules among different malware families could lead to incorrect classification results. To address this issue, a more complex pruning phase is employed, in which only rules whose values of support and confidence satisfy a given property (e.g., associated with a specific threshold value) are considered valid.

More precisely, let $A = \{a_1, ..., a_{|A|}\}$ the set of admissible API calls, a generic rule is defined as follows:

$$R_{pq} = \{a_p \rightarrow a_q\} \tag{5.2}$$

with $a_p, a_q \in A$ and $a_p \prec a_q$, which denotes that the API a_p is called before than a_q.

By recalling the aforementioned association between rules and nodes in the merged graph G^{t_i} associated with the application $t_i \in T$, and defining $\sigma^{t_i}(R_{pq})$ as the number of occurrence of the rule R_{pq} in a given application $t_i \in T$, then the support of the rule R_{pq} with respect to the class c is defined as follows:

$$\Phi^c(R_{pq}) = \frac{|R_{pq}| \sum_{i=1}^{\mathcal{N}(c)} \frac{\sigma^{t_i}(R_{pq})}{l_i}}{\mathcal{N}(c)} \tag{5.3}$$

where l_i is the number of API calls of t_i, while $|R_{pq}|$ is the cardinality of the rule (i.e. 2).

Notice that, to take into account the different APIs flow lengths that could occur among the applications of a given class c as well as the unbalancing among the applications falling within classes of T, the terms l_i, $|R_{pq}|$, and $\mathcal{N}(c)$ of Equ. 5.3 are used to normalize the support within the range $[0, 1]$.

However, as it is known, the support is not sufficient to estimate the quality of the rules in representing the applications for multi-class contexts. Thus, the confidence of a given rule R_{pq} on a class c is also defined, as follows:

$$\Gamma^c(R_{pq}) = \frac{\Phi^c(R_{pq})}{\sum_{v \in C} \Phi^v(R_{pq})} \tag{5.4}$$

Equ. 5.4 expresses the ability of a rule to be unique for a specific class. Indeed, high values denote high uniqueness, while low values indicate that the rule is also present in other malware classes.

As depicted, Equ. 5.3 and Equ. 5.4 express a metric characterizing a given rule concerning a class c, and thus, they can be involved in the pruning phase. More specifically, rules having support and confidence less than given thresholds are pruned.

5.2.3 *Classification*

Once the training phase is performed and, thus, several rules are mined, new incoming applications need to be classified. To accomplish this, the following metrics are defined [23].

Firstly, the following confidence is provided:

$$\Gamma^{c}_{R_{pq}}(t_m) = \begin{cases} 0 & R_{pq} \not\subseteq t_m, \\ \sigma^{t_m}(R_{pq}) * \Gamma^{c}(R_{pq}) & R_{pq} \subseteq t_m \ \wedge \ \gamma(t_m) = c, \\ 1/\Gamma^{c}(R_{pq}) & R_{pq} \subseteq t_m \ \wedge \ \gamma(t_m) \neq c. \end{cases} \quad (5.5)$$

where $\gamma(t_m)$ is the hypothesis of membership classes associated with t_m.

It represents the confidence of a rule R_{pq} with respect to a class c associated with an application t_m. As shown, its value depends on the presence of the rule R_{pq} within the application t_m, as well as on the value assumed by the hypothesis of membership classes.

After that, the degree of belonging to class c of an application t_m is estimated by evaluating a rank ρ, which is given by:

$$\rho^{c}(t_m) = \sum_{c \in C} \sum_{\forall p,q} \Gamma^{c}_{R_{pq}}(t_m), \quad (5.6)$$

where the summation on p and q takes into account all the rules derived from the training phase.

Finally, the softmax function is used to classify the application t_m, as follows:

$$\gamma(t_m) = \arg\max_{h \in C} \frac{e^{\rho^{h}(t_m)}}{\sum_{v \in C} e^{\rho^{v}(t_m)}}. \quad (5.7)$$

5.3 THE PROPOSED APPROACH

This section presents the proposed federated architecture. More precisely, I extend the support and confidence indexes within a privacy-preserving federated environment. Then, I provide some implementation details related to core modules. Finally, I describe the proposed schema by highlighting its main advantages.

5.3.1 *Federated indexes definition*

The first step necessary for implementing the previously-discussed detector within a federated logic is to extend both support and confidence by considering the entire dataset T split among several federated clients. Note that different clients could deal with

similar malware. As a consequence, T could include multiple copies of the same applications.

Let M be the number of considered clients, and T_j be the j-th dataset gathered by the client j, then the entire dataset T is subject to the following:

$$T = \bigcup_{j=1}^{M} T_j \tag{5.8}$$

Notice that, during the learning process, no T_j dataset is sent to the central server, but only the applications-related graphs G_c are sent, guaranteeing privacy. Furthermore, Equ. 5.8 expresses the ability of our approach to face the problem of learning from non-IID data. In fact, according to Equ. 5.8, all partial graphs are merged into a single one, and then the support and confidence are evaluated only by the central server.

Accordingly, let $\mathcal{N}(c, T_j)$ be the number of application $t_i \in T_j$ whose class is c, then the federated support of the rule R_{pq} can be defined as follows:

$$\Phi_f^c(R_{pq}) = \frac{|R_{pq}| \sum_{j=1}^{M} \sum_{i=1}^{\mathcal{N}(c,T_j)} \frac{\sigma^{t_i}(R_{pq})}{l_i}}{\mathcal{N}(c)} \tag{5.9}$$

Similarly, the federated confidence of a given rule R_{pq} on a class c is defined as follows:

$$\Gamma_f^c(R_{pq}) = \frac{\Phi_f^c(R_{pq})}{\sum_{v \in C} \Phi_f^v(R_{pq})} \tag{5.10}$$

Finally, Equations 5.5 and 5.6 can be easily generalized to the federated case by replacing Φ with Φ_f and Γ with Γ_f, respectively and, also in this case, the classification of an application is performed by Equ. 5.7.

5.3.2 *The federated rules-based detector*

To face the confidentiality and privacy issues related to sharing data, I present an architecture that aims to support malware classification tasks by embedding the proposed associative rules-based detector within a federated logic, in which the involved IoT

devices need to send their data to a central aggregation point devoted to sharing information. More precisely, the proposed architecture aims to provide a privacy-preserving workflow in which federated entities asynchronously share only their applications-related graphs (and not the raw data) and receive the malware detector. Thus, the proposed architecture can guarantee the interchange of sensitive information, and their protection against the most common data leakage threats, as done in traditional Federated Learning-based approaches. Furthermore, it also avoids the issues related to integration operations. Indeed, the centralized aggregation of graphs is extremely simple and presents a low computational effort. In addition, since Markov chains are defined as a memoryless stochastic process, each set of mined k-spaced associative rules, derived from the dynamic analysis of API calls, represents a locally trained model, which is equivalent to the local model of the classic FL-based solutions.

Moreover, the proposed workflow needs (in theory) only one centralized aggregation to build the presented classifier, and that does not need any usage of sophisticated algorithms, such as the Federated Averaging (FedAvg) [72], Federated Matched Averaging (FedMA) [113], and Federated Distance (FedDist) [31]. Therefore, the obtained results no longer depend on applications processed by the federated entities (e.g., influenced by particular data distribution) but only from the derived global model. Indeed, since the defined support and confidence indexes (see Equ. 5.9 and Equ. 5.10) are computed on the centralized and aggregated associative rules, I highlight the ability of the proposed logic to face the non-IID data-related issues that, instead, adversely affect the traditional federated learning-based solutions.

For this reason, the presented workflow differs from pure FL-based ones and is also extremely suitable to improve the convergence process among Edge and Cloud infrastructures, with specific reference to data aggregation, data security, and services migration [15, 91, 98]. Hence, to report as much detailed information as possible, I describe the resulting architecture, structured according to a Publish-Subscribe model/policy, by defining three processes named *Client-Side Extraction*, *Server-Side Aggregation*, and *Detector Update*, respectively.

Therefore, the main goals of the proposed architecture can be summarized as follows:

1. A data extraction workflow is needed to collect associative rules from each federated entity (Client-Side Extraction);
2. A data aggregation workflow is useful to manage the received rules as category graphs and share a malware detector with each entity (Server-Side Aggregation);
3. A data update workflow is necessary for periodically re-adapting and re-sharing the malware detector (Detector Update).

5.3.2.1 *Client-Side Extraction process*

At the beginning of the Client-Side Extraction process, each federated entity asks to subscribe to the central server and receives the number of steps n to run. Next, for each analyzed application, and at each iteration $k < n$, the client extracts the k-spaced associative rules. Finally, the client sends the related graph to the central server. Fig. 5.6 reports the discussed Client-Side Extraction process, while its steps can be summarized as Alg. 1.

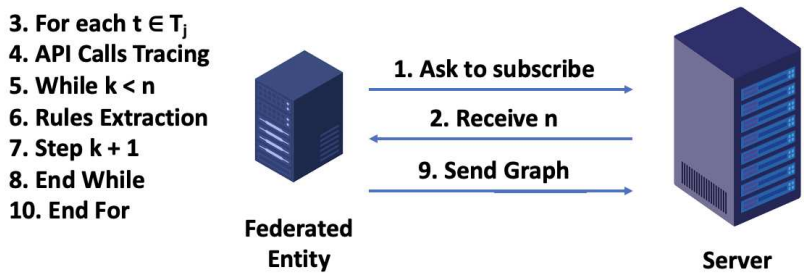

Figure 5.6: The high-level steps of the Client-Side Extraction process that summarize the application pre-processing and sending of the related graph.

More precisely, for each application t, Alg. 1 derives the corresponding class c and the list of API calls (*apisList*). Next, at each iteration $k < n$, the algorithm extracts the set of k-spaced associative rules by storing them in G. Note that, after the while

Algorithm 1 Client-Side Extraction

Require: T_j (j-th dataset of applications)
1: $n \leftarrow$ askSubscription()
2: **for each** t $\in T_j$ **do**
3: k $\leftarrow 1$
4: G $\leftarrow \emptyset$
5: c $\leftarrow$ Class(t)
6: apisList $\leftarrow$ traceAPIs(t)
7: **while** k $< n$ **do**
8: G $\leftarrow$ extractRules(apisList,k)
9: $k \leftarrow k + 1$
10: **end while**
11: sendGraph(G,c)
12: **end for**

loop, G will represent the graph containing any mined rules. Finally, the algorithm sends G and c to the central server and processes another application.

5.3.2.2 *Server-Side Aggregation process*

The Server-Side Aggregation process has the fundamental task of collecting the application-related graphs to share the proposed model with each federated entity. To accomplish this, the server first merges each graph with those previously received. Then, it uses the obtained information (i.e. the learned rules) to perform the pruning phase and share the malware detector. Note that the described process confers to the server the capability of continuously sharing an updated classifier each time a new application is received. Therefore, the server can immediately support the federated entities for their detection activities, also in presence of zero-day malware. Fig. 5.7 shows the Server-Side Aggregation process, while its steps can be summarized as Alg. 2.

More precisely, every time the i-th client sends the graph G and the corresponding class c, Alg. 2 merges G with the graphs previously stored in $graphList[c]$. Note that $graphList[c]$ contains the set of k-spaced rules related to class c. Next, the algorithm performs the pruning phase and sends the *detector* to each subscribed client.

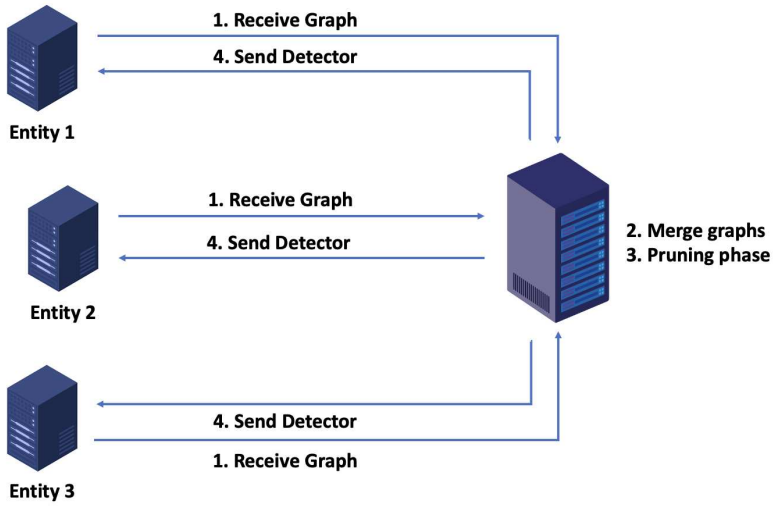

Figure 5.7: The high-level steps of the Server-Side Aggregation process that summarize the pruning phase and sharing of the detector.

Algorithm 2 Server-Side Aggregation

Require: M (number of subscribed clients)

```
for each i ≤ M do
    (G,c) ← receiveGraph(i)
    graphList[c] ← mergeGraph(G)
    detector ← pruningPhase(graphList)
    sendToClients(detector)
end for
```

5.3.2.3 *Detector Update process*

This process is responsible for taking into account new malware applications when a detector has already been shared. To accomplish this, it combines the previous phases by considering new applications. Therefore, another ability of the presented architecture is to continuously share an updated classifier regardless of the presence of new or non-IID data.

More precisely, when an unknown malware application is detected from a federated entity, the application-related graphs are built and sent to the server along with the related malware

class. Next, the server will merge the received graphs with those already stored, and after the pruning phase, it will share the updated malware detector with each subscribed entity.

5.4 EXPERIMENTAL RESULTS

The first goal of the experiments is devoted to demonstrating the contribution of the proposed architecture concerning the classification of several malware applications, while the second is to study the required computational effort through performance analysis. To accomplish this, I show the effectiveness of the proposed detector, compared with other state-of-the-art approaches, by considering both centralized and decentralized data. Next, I analyze the required computational effort in the presence of several federated entities and non-IID data partitions, respectively.

5.4.1 *Dataset and Experimental setting*

The dataset considered in the following experiments has been derived by Unisa Malware Dataset (UMD), composed of about 3500 applications grouped into 8 Android families: Airpush (Air), DroidKungFu (DKF), Fusob (Fus), Genpua (Gen), GinMaster (Gin), Jisut (Jis), Opfake (Opf), and SmsPay (Sms). More precisely, to evaluate the effectiveness of the proposed detector, I first divided the related API-Calls dataset into training and testing sets according to the 70/30 criteria, as reported in Tab. 5.1.

Thus, I employed the obtained sets for the centralized and decentralized learning scenarios by respectively considering the following splitting criteria and number of involved clients:

- Splitting: Horizontal (Samples of each category equally distributed), Vertical (Samples of a category assigned only for some clients), and Mixed;
- Clients: 4, 8, 12, and 16.

Finally, I simulated the proposed architecture within a socket-based Client-Server scenario, using a MacBook Pro equipped with an Apple M1 CPU and 16 GB of unified memory for the

Family	Training	Testing	Total
Air	265	114	379
DKF	700	301	1001
Fus	117	49	166
Gen	220	94	314
Gin	372	160	532
Jis	376	161	537
Opf	431	184	615
Sms	122	52	174
Total	2603	1115	3718

Table 5.1: Dataset division according to the 70/30 criteria.

sever-side and several virtualized clients Linux-based with 2 Processors and 2 GB RAM, respectively.

5.4.2 *Evaluation metrics*

To appreciate the classification quality of the proposed detector, I derived the following evaluation metrics from the multi-class confusion matrix: Accuracy (Acc.), Sensitivity (Sens.), Specificity (Spec.), Precision (Prec.), F-Score (F-Mea.), and Area Under the ROC Curve (AUC).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.11}$$

$$Sensitivity = \frac{TP}{TP + FN} \tag{5.12}$$

$$Specificity = \frac{TN}{TN + FP} \tag{5.13}$$

$$Precision = \frac{TP}{TP + FP} \tag{5.14}$$

$$F - Score = \frac{2 * Sens * Prec}{Sens + Prec} \tag{5.15}$$

$$AUC = \frac{Sens + Spec}{2} \tag{5.16}$$

For each category, TPs (True Positives) refer to the applications correctly classified, while TNs (True Negatives) refer to the applications correctly identified in another category. Conversely, FPs (False Positives) are the applications mistakenly identified as the considered category, while FNs (False Negatives) present the applications mistakenly identified in another category. Notice that all reported values have been defined in $[0, 1]$. Also, in order to achieve a global evaluation of the proposed detector, the average performance values (Avg.) and the standard deviation (Dev.) for each considered metric.

5.4.3 *Achieved results*

To demonstrate the effectiveness and evaluate the performances of the proposed federated architecture, I first considered the employed dataset as centralized data processed by one client only. More precisely, for each application related to the training set, all the possible associative rules have been mined by using a progressive spacing $k \in [1, 100]$ and then sent to the server as related graphs. Then, for each malware family, I merged the corresponding graphs as one representative graph. Next, since the number of extracted rules was too high and could adversely affect the classification results, I performed the pruning phase by considering any possible value of support (supp.) and confidence (conf.) between 0.1 and 1.0. Finally, I employed the testing set to evaluate the quality of the selected training rules by achieving the best classification performances using the combination of supp./conf. = 0.7/1.0. The obtained results, respectively shown in Tabs. 5.2 and 5.3, demonstrate the effectiveness of the detector in classifying several malware families with an average accuracy of 99% and a limited number of FPs and FNs.

Subsequently, to show the effectiveness in a federated environment, we repeated the training process by considering several training set sub-partitions distributed, each time, among different clients. As shown in Tab. 5.4, since the application-related graphs extraction process is performed only by clients, the proposed federated detector achieved excellent classification performances independently of the splitting criteria used and the number of subscribed clients, respectively.

	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
Air	112	0	0	1	1	0	0	0
DKF	4	289	0	2	1	0	0	5
Fus	0	0	49	0	0	0	0	0
Gen	1	0	0	89	1	0	0	3
Gin	1	3	1	1	153	0	0	1
Jis	0	0	0	0	0	161	0	0
Opf	0	0	0	0	0	0	184	0
Sms	1	1	0	6	1	0	0	43

Table 5.2: Confusion matrix related to centralized data.

	Acc.	Sens.	Spec.	Prec.	F-Mea.	AUC
Air	0.9917	0.9412	0.9979	0.9825	0.9614	0.9696
DKF	0.9854	0.9863	0.9851	0.9601	0.9731	0.9857
Fus	0.9991	0.9800	1.0000	1.0000	0.9899	0.9900
Gen	0.9863	0.8990	0.9950	0.9468	0.9223	0.9470
Gin	0.9899	0.9745	0.9925	0.9563	0.9653	0.9835
Jis	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Opf	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Sms	0.9836	0.8269	0.9914	0.8269	0.8269	0.9092
Avg.	0.9920	0.9510	0.9952	0.9591	0.9549	0.9731
Dev. (+/-)	0.0064	0.0566	0.0050	0.0539	0.0539	0.0291

Table 5.3: Performance results related to centralized data.

5.4.4 *Comparison and discussion*

Next, to highlight the potentialities of the presented classifier, I compared the achieved results with those derived by the most famous ML-based methods provided by the Scikit-learn [84] Python library, namely: the Random Forest (RF) algorithm, Linear Support Vector Machine (SVM) classifier, Decision Trees (DTs), and Gaussian Naive Bayes (GNB) classifier, respectively. I accomplished this by arranging the k-spaced APIs sequences as

Clients	Acc.	Sens.	Spec.	Prec.	F-Mea.	AUC
Centr.	0.9920	0.9510	0.9952	0.9591	0.9549	0.9731
4	0.9920	0.9510	0.9952	0.9591	0.9549	0.9731
8	0.9920	0.9510	0.9952	0.9591	0.9549	0.9731
12	0.9920	0.9510	0.9952	0.9591	0.9549	0.9731
16	0.9920	0.9510	0.9952	0.9591	0.9549	0.9731

Table 5.4: Comparison with different splitting criteria and subscribed clients.

corresponding adjacent matrices, in which each APIs pair has been represented with the related frequency. Tabs. 5.5, 5.6, 5.7, and 5.8 report the obtained classification metrics for each ML-based method used, while Tab. 5.9 compares these methods with the proposed detector.

	Acc.	Sens.	Spec.	Prec.	F-Mea.	AUC
Air	0.9748	0.9023	0.9844	0.8844	0.8933	0.9434
DKF	0.9474	0.9494	0.9468	0.8513	0.8977	0.9481
Fus	0.9998	1.0000	0.9998	0.9969	0.9985	0.9999
Gen	0.9677	0.8063	0.9871	0.8830	0.8429	0.8967
Gin	0.9681	0.8757	0.9869	0.9318	0.9029	0.9313
Jis	0.9988	0.9979	0.9990	0.9934	0.9956	0.9984
Opf	0.9970	0.9900	0.9978	0.9822	0.9861	0.9939
Sms	0.9690	0.6348	0.9902	0.8034	0.7093	0.8125
Avg.	0.9778	0.8946	0.9865	0.9158	0.9033	0.9405
Dev. (+/-)	0.0177	0.1170	0.0160	0.0672	0.0907	0.0594

Table 5.5: Performance results related to Random Forest.

As reported in Tab. 5.9, the proposed approach outperformed each traditional ML-based method taken into consideration. More precisely, it achieved an average F-Score improvement of 19% on the GNB classifier and 11% on the Decision Trees. In addition, the RF and SVM classifiers, which have obtained good classification results, have been outperformed by the federated detector with

	Acc.	Sens.	Spec.	Prec.	F-Mea.	AUC
Air	0.9682	0.8820	0.9797	0.8517	0.8666	0.9308
DKF	0.9422	0.9042	0.9544	0.8642	0.8837	0.9293
Fus	0.9973	1.0000	0.9971	0.9585	0.9788	0.9986
Gen	0.9557	0.7907	0.9756	0.7960	0.7933	0.8832
Gin	0.9556	0.8578	0.9755	0.8770	0.8673	0.9167
Jis	0.9978	0.9901	0.9989	0.9929	0.9915	0.9945
Opf	0.9931	0.9736	0.9955	0.9639	0.9688	0.9846
Sms	0.9525	0.4810	0.9824	0.6346	0.5472	0.7317
Avg.	0.9703	0.8599	0.9824	0.8674	0.8622	0.9212
Dev. (+/-)	0.0211	0.1582	0.0139	0.1078	0.1352	0.0813

Table 5.6: Performance results related to Linear SVM.

	Acc.	Sens.	Spec.	Prec.	F-Mea.	AUC
Air	0.9516	0.8116	0.9702	0.7828	0.7969	0.8909
DKF	0.9336	0.8715	0.9535	0.8574	0.8644	0.9125
Fus	0.9986	1.0000	0.9985	0.9786	0.9892	0.9993
Gen	0.9461	0.7297	0.9721	0.7591	0.7441	0.8509
Gin	0.9489	0.8473	0.9696	0.8501	0.8487	0.9084
Jis	0.9982	0.9979	0.9983	0.9888	0.9933	0.9981
Opf	0.9936	0.9843	0.9948	0.9586	0.9713	0.9895
Sms	0.9519	0.5201	0.9792	0.6135	0.5629	0.7496
Avg.	0.9653	0.8453	0.9795	0.8486	0.8464	0.9124
Dev. (+/-)	0.0250	0.1530	0.0153	0.1206	0.1375	0.0803

Table 5.7: Performance results related to Decision Trees.

an average improvement of 5% and 9%, respectively. However, differently from the proposed approach, such methods have been trained on the centralized raw data arranged as adjacent matrices. Therefore, they need an extra pre-processing step as well as not guarantee the privacy of the involved clients and the confidentiality of related data.

	Acc.	Sens.	Spec.	Prec.	F-Mea.	AUC
Air	0.9432	0.7546	0.9682	0.7587	0.7567	0.8614
DKF	0.8760	0.7551	0.9149	0.7401	0.7475	0.8350
Fus	0.9925	1.0000	0.9920	0.8935	0.9438	0.9960
Gen	0.9449	0.6280	0.9830	0.8165	0.7100	0.8055
Gin	0.9033	0.5411	0.9771	0.8280	0.6545	0.7591
Jis	0.9850	0.9994	0.9828	0.8984	0.9462	0.9911
Opf	0.9359	0.9811	0.9304	0.6334	0.7698	0.9558
Sms	0.9541	0.5894	0.9772	0.6215	0.6050	0.7833
Avg.	0.9419	0.7811	0.9657	0.7738	0.7667	0.8734
Dev. (+/-)	0.0361	0.1784	0.0259	0.0993	0.1151	0.0888

Table 5.8: Performance results related to Gaussian NB.

	Acc.	Sens.	Spec.	Prec.	F-Mea.	AUC
Proposed	0.9920	0.9510	0.9952	0.9591	0.9549	0.9731
RF	0.9778	0.8946	0.9865	0.9158	0.9033	0.9405
SVM	0.9703	0.8599	0.9824	0.8674	0.8622	0.9212
DTs	0.9653	0.8453	0.9795	0.8486	0.8464	0.9124
GNB	0.9419	0.7811	0.9657	0.7738	0.7667	0.8734

Table 5.9: Comparison with ML-based methods.

In addition, I used the adjacent matrices (73×73) to test a DL-based approach. I accomplished this through a CNN, shown in Fig. 5.8, trained in the centralized and federated-learning scenarios, respectively. Tab. 5.10 summarizes the results derived by considering each splitting criteria and the number of subscribed clients, Tab. 5.11 reports the client-related Accuracy values derived on the testing dataset, while Tab. 5.12 compares our approach with the employed CNN.

As shown in Tabs. 5.10 and 5.11, the employed CNN achieved comparable results only with the Mixed splitting criteria (Mix). Vice-versa, due to the lack of convergence of the Federated CNN, no relevant results have been achieved by the other splitting criteria (Vertical and Horizontal). More precisely, the Vertical one

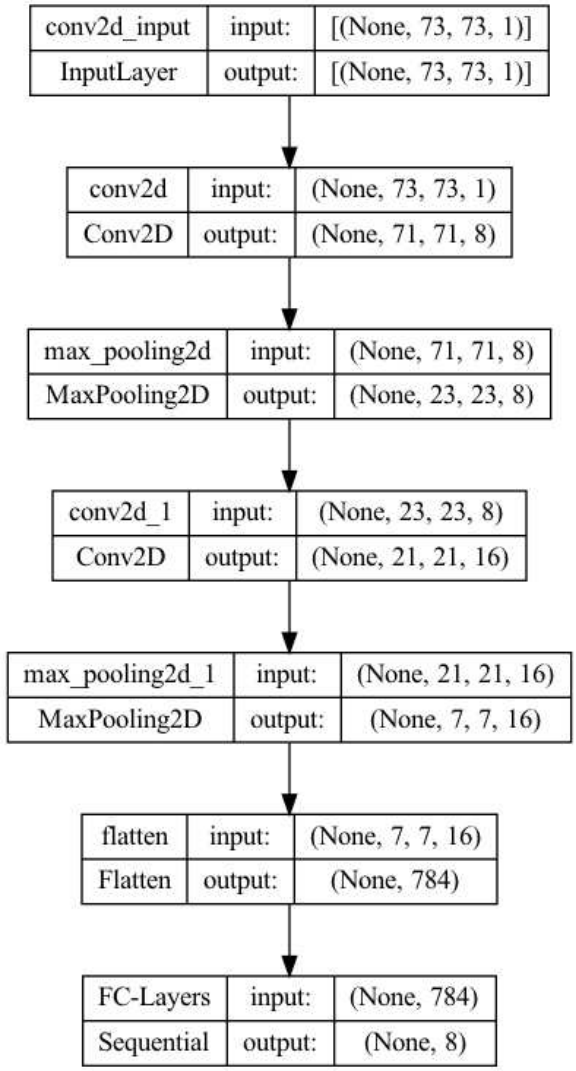

Figure 5.8: Architecture of the CNN used in comparisons.

has highlighted how DL-based models cannot learn if the data-related categories are distributed only to some clients, adversely affecting the federated learning process. Instead, the Horizontal one has highlighted how IID data can also adversely influence the FL-based models. This phenomenon occurs when data are characterized from many sub-categories, producing a sort of nested Vertical splitting. In our case, this means that each k-spaced adjacent matrix can be related to a specific k-th sub-

Split	Acc.	Sens.	Spec.	Prec.	F-Mea.	AUC
Centr.	0.9697	0.8692	0.9820	0.8741	0.8711	0.9256
Mix4	0.9695	0.8705	0.9819	0.8754	0.8714	0.9262
Mix8	0.9659	0.8585	0.9799	0.8576	0.8563	0.9192
Mix12	0.9635	0.8467	0.9784	0.8486	0.8462	0.9126
Mix16	0.9603	0.8318	0.9766	0.8325	0.8297	0.9042

Table 5.10: Performance Results related to CNN.

Clients	Split	min_Acc.	max_Acc.	avg_Acc.
4	Mix	0.9678	0.9687	0.9683
8	Mix	0.9638	0.9655	0.9647
12	Mix	0.9595	0.9622	0.9613
16	Mix	0.9561	0.9595	0.9583

Table 5.11: Clients-related Accuracy values.

Method	Acc.	Sens.	Spec.	Prec.	F-Mea.	AUC
Proposed	0.9920	0.9510	0.9952	0.9591	0.9549	0.9731
Centr.	0.9697	0.8692	0.9820	0.8741	0.8711	0.9256
Mix4	0.9695	0.8705	0.9819	0.8754	0.8714	0.9262

Table 5.12: Comparison with the CNN.

category, with k varying from 1 to 100. Therefore, these matrices adversely affect the federated learning process also when they seem to be Independent and Identically Distributed (IID). Finally, the comparison provided in Tab. 5.12 shows that our approach outperformed the FL-based CNN by achieving an average F-Score improvement of 8%. Therefore, this confirms the effectiveness of the proposed detector in achieving excellent classification metrics for any considered splitting criteria (IID and no-IID) that, instead, adversely affect the classical FL-based solutions [18, 46, 57, 83].

5.4.5 *Performance evaluation*

To evaluate the performance of the proposed architecture during the Client-Side Extraction and Server-Side Aggregation processes, I first derived the required time effort by using the entire training set on a single machine. Next, I partitioned it by applying the previously mentioned splitting criteria and considering the number of involved clients, as reported in Tabs. 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.19, 5.20 and 5.21, respectively.

Clients	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
4	66	175	29	55	93	94	107	30
8	33	87	14	27	46	47	53	15
12	22	58	9	18	31	31	35	10
16	16	43	7	13	23	23	26	7

Table 5.13: Horizontal training set division for each client.

Client	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
C1	265	700	0	0	0	0	0	0
C2	0	0	117	220	0	0	0	0
C3	0	0	0	0	372	376	0	0
C4	0	0	0	0	0	0	431	122

Table 5.14: Vertical training set division for 4 clients.

Client	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
C1	265	350	0	0	0	0	107	0
C2	0	350	117	0	0	0	107	0
C3	0	0	0	110	186	188	107	0
C4	0	0	0	110	186	188	110	122

Table 5.15: Mixed training set division for 4 clients.

Hence, to carefully analyze the model scalability for each considered combination (splitting criteria/number of clients), I derived the parallel Speedup [97], which is given by:

Client	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
C1	132	350	0	0	0	0	0	0
C2	0	0	58	110	0	0	0	0
C3	0	0	0	0	186	188	0	0
C4	0	0	0	0	0	0	215	61
C5	133	350	0	0	0	0	0	0
C6	0	0	59	110	0	0	0	0
C7	0	0	0	0	186	188	0	0
C8	0	0	0	0	0	0	216	61

Table 5.16: Vertical training set division for 8 clients.

Client	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
C1	132	175	0	0	0	0	107	0
C2	0	175	117	0	0	0	107	0
C3	0	0	0	55	93	94	107	0
C4	0	0	0	55	93	94	110	122
C5	133	175	0	0	0	0	0	0
C6	0	175	0	0	0	0	0	0
C7	0	0	0	55	93	94	0	0
C8	0	0	0	55	93	94	0	0

Table 5.17: Mixed training set division for 8 clients.

$$Speedup = \frac{T_s}{T_p}, \tag{5.17}$$

where T_s is the time effort without parallelism and T_p is the required time effort with parallelism. Fig. 5.9 shows the speedup-related behavior for each considered splitting criteria and clients, respectively.

As shown in Fig. 5.9, since the derived Speedup values are slightly different, it is possible to appreciate how the proposed model is characterized by semi-linear scalability. More precisely, when the number of clients is low (i.e. 4 and 8), the system rapidly scales thanks to low communication costs. Instead, when the

Client	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
C1	66	175	0	0	0	0	0	0
C2	0	0	29	55	0	0	0	0
C3	0	0	0	0	93	94	0	0
C4	0	0	0	0	0	0	215	61
C5	133	350	0	0	0	0	0	0
C6	0	0	59	110	0	0	0	0
C7	0	0	0	0	186	188	0	0
C8	0	0	0	0	0	0	108	30
C9	66	175	0	0	0	0	0	0
C10	0	0	29	55	0	0	0	0
C11	0	0	0	0	93	94	0	0
C12	0	0	0	0	0	0	108	31

Table 5.18: Vertical training set division for 12 clients.

Client	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
C1	66	175	0	0	0	0	53	0
C2	0	175	29	0	0	0	53	0
C3	0	0	0	55	93	94	53	0
C4	0	0	0	55	93	94	53	30
C5	133	175	0	0	0	0	0	0
C6	0	175	0	0	0	0	0	0
C7	0	0	0	55	93	94	0	0
C8	0	0	0	55	93	94	0	0
C9	66	0	0	0	0	0	53	0
C10	0	0	29	0	0	0	53	30
C11	0	0	29	0	0	0	53	30
C12	0	0	30	0	0	0	60	32

Table 5.19: Mixed training set division for 12 clients.

number of clients increases, the related execution costs decrease to the point that counterbalances the high communication costs.

Client	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
C1	66	175	0	0	0	0	0	0
C2	0	0	29	55	0	0	0	0
C3	0	0	0	0	93	94	0	0
C4	0	0	0	0	0	0	107	30
C5	66	175	0	0	0	0	0	0
C6	0	0	29	55	0	0	0	0
C7	0	0	0	0	93	94	0	0
C8	0	0	0	0	0	0	107	30
C9	66	175	0	0	0	0	0	0
C10	0	0	29	55	0	0	0	0
C11	0	0	0	0	93	94	0	0
C12	0	0	0	0	0	0	107	30
C13	67	175	0	0	0	0	0	0
C14	0	0	30	55	0	0	0	0
C15	0	0	0	0	93	94	0	0
C16	0	0	0	0	0	0	110	32

Table 5.20: Vertical training set division for 16 clients.

For this reason, differently by the classical FL-based solutions [18, 46, 57, 83], the proposed architecture can obtain excellent time performances in the presence of non-IID.

5.5 CONCLUSIONS AND FUTURE WORKS

In this Chapter, I investigated the capabilities of Markov chains and associations-based rules within an IoT environment in order to support a privacy-aware malware classification. To accomplish this, I enhanced the dynamic-based detector, presented in [23], through the federated logic discussed in Chap. 4. Therefore, I proposed a dedicated architecture capable of performing a federated training process in which end-users build a shared detector by sending the analyzed applications to a central server. Consequently, the interchange of sensitive information, and their protec-

Client	Air	DKF	Fus	Gen	Gin	Jis	Opf	Sms
C1	66	87	0	0	0	0	53	0
C2	0	87	29	0	0	0	53	0
C3	0	0	0	55	93	94	53	0
C4	0	0	0	55	93	94	53	30
C5	66	87	0	0	0	0	0	0
C6	0	87	0	0	0	0	0	0
C7	0	0	0	55	93	94	0	0
C8	0	0	0	55	93	94	0	0
C9	66	0	0	0	0	0	53	0
C10	0	0	29	0	0	0	53	30
C11	0	0	29	0	0	0	53	30
C12	0	0	30	0	0	0	60	32
C13	67	87	0	0	0	0	0	0
C14	0	87	0	0	0	0	0	0
C15	0	87	0	0	0	0	0	0
C16	0	91	0	0	0	0	0	0

Table 5.21: Mixed training set division for 16 clients.

tion against the most common data leakage threats, has been similarly guaranteed as done in traditional Federated Learning-based approaches. Next, I validated the effectiveness of the presented method by employing several famous malware families derived from the UMD dataset. More precisely, the obtained results have shown an average accuracy of 99% by outperforming the most famous DL and FL-based approaches and highlighting possible benefits in zero-day malware detection. Finally, I also provided a statistical and temporal performance assessment in the presence of non-IID data, in which I considered several dataset partitions, splitting criteria, and number of subscribed clients, respectively.

However, due to the high number of existing and yearly released malware applications, this study proposes some possible future works that might provide some benefits, such as well-suited mechanisms to improve the zero-day detection and the

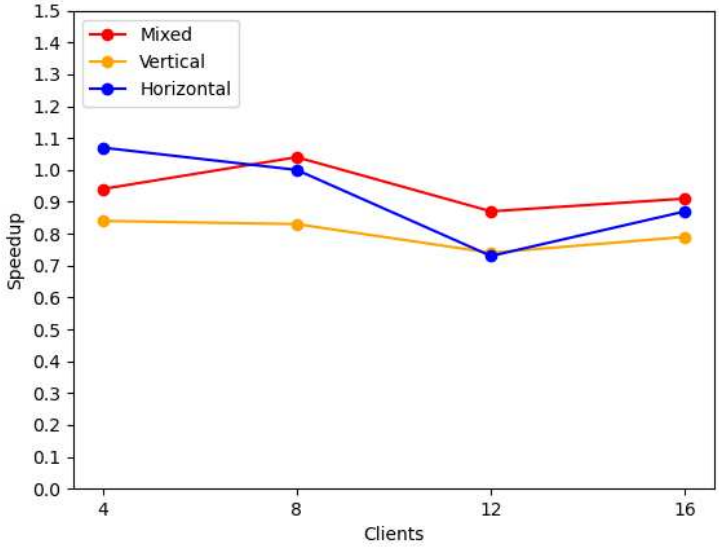

Figure 5.9: Speedup comparison.

building of new communication channels and environments for federated entities, respectively.

For this reason, I will investigate an extension of the proposed detector in order to detect new malware families and variants characterized by more intricate dynamic patterns. For instance, given an arbitrary rule, the employed pruning indexes might consider only the "nearest rules". Also, I will optimize the applications-related extraction process in order to improve the required temporal effort. For instance, several selection criteria could be employed (e.g. the proposed pruning indexes) to consider only the relevant rule sequences effectively mined during the application execution.

6

CONCLUSIONS AND FUTURE WORKS

In this thesis, I presented new enhanced DL-based strategies to spot malware threats in IoT network security scenarios. Therefore, the proposed approaches, derived from my PhD activities and related publications, have been designed with the aim of guaranteeing the success of the related early alerting facilities.

First, in Chapter 2, I started by focusing on Malware classification related to Android-based devices, which represent one of the most famous hostile activities sources. More precisely, I remarked on the effectiveness of dynamic features, arranged as API-Images, to classify several Malware families through a Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN), respectively. The derived results have shown an average accuracy of 99% for both employed DNNs by outperforming the most famous ML-based approaches.

Next, in Chapter 3, I presented an extension of such API-Images to face an enhanced Malware classification as Video-Classification tasks. The proposed features representation technique, named API-Streams, has been proven effective in classifying different Malware families through a CNN-LSTM Sparse Autoencoder (CNN-LSTM-SAE). The achieved results, also compared with those derived by the most famous static and dynamic-based approaches, have proven the effectiveness of the proposed model by obtaining an average accuracy of 98% in the presence of several unbalanced training datasets. In addition, such results also highlighted the existence of several malware sub-categories characterized by a well-distinct dynamic behavior.

On the other hand, in Chapter 4, I analyzed other Android malware families by analyzing the effectiveness of a static-based approach. More precisely, I presented a new representation technique called Permission Maps (Perm-Maps), which combines information related to Android permissions and their corresponding severity levels. The achieved results, carried out through a CNN enhanced by a training process based on federative logic,

have demonstrated the effectiveness of the employed detector by obtaining an average accuracy of 99.74%.

Finally, in Chapter 5, I combined the effectiveness of dynamic-based approaches with the cooperation of federated devices by providing a privacy-preserving Malware detector. In this direction, the capabilities of Markov Chains and associative rules have been investigated in order to improve the state-of-the-art solutions. More precisely, the achieved results have proven that the proposed approach cannot be affected by non-Independent and Identically Distributed (non-IID) data in terms of required time effort and classification results (with an average accuracy always of 99.20%), respectively.

On the basis of achieved outcomes, I firmly believe that new empowered detection strategies will play a more and more dominant role in effectively spotting many kinds of threats in Internet security scenarios, with particular attention to those IoT and Federated environments-related. For this reason, I would propose the following three future research directions.

First, I will investigate the abilities of CNN and LSTM Autoencoders in detecting malware sub-categories, which are often characterized by a well-distinct dynamic behavior. Indeed, the Autoencoders' unsupervised learning mechanism might improve the effectiveness of the Run-time malware detection and propose more targeted DL-based approaches, respectively. Second, I will enhance the proposed federated detectors in order to detect future malware families. More precisely, empowering these solutions might provide several benefits, such as new mechanisms capable of counteracting the malware-related spread, improving zero-day detection, and creating new privacy-preserving environments. Finally, since traffic anomaly detection activities represent another fundamental topic related to Internet security scenarios, I will investigate the effectiveness of the presented techniques in spotting illegitimate traffic flows in federated organizations. More precisely, the combination of such approaches might be helpful in detecting simultaneously hostile activities coming from multiple federated entities, which can also be seen as classic traffic observation points. Therefore, this future work might better provide the foundations for a new generation of Federated Intrusion Detection and Prevention Systems.

BIBLIOGRAPHY

[1] Abada Abderrahmane, Guettaf Adnane, Challal Yacine, and Garri Khireddine. "Android Malware Detection Based on System Calls Analysis and CNN Classification." In: *2019 IEEE Wireless Communications and Networking Conference Workshop (WCNCW)*. 2019, pp. 1–6. DOI: 10.1109/WCNCW.2019.8902627.

[2] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. "Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning." In: *Nature* 33.8 (2015), pp. 831–838. DOI: https://doi.org/10.1038/nbt.3300. URL: https://doi.org/10.1038/nbt.3300.

[3] Android. *Permissions on Android*. https://developer.android.com/guide/topics/permissions/overview. Last Access: 02/01/2023.

[4] Android. *R.attr*. https://developer.android.com/guide/topics/permissions/overview. Last Access: 02/01/2023.

[5] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. "Obfuscapk: An open-source black-box obfuscation tool for Android apps." In: *SoftwareX* 11 (2020), p. 100403. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100403.

[6] Simone Aonzo, Alessio Merlo, Mauro Migliardi, Luca Oneto, and Francesco Palmieri. "Low-Resource Footprint, Data-Driven Malware Detection on Android." In: *IEEE Transactions on Sustainable Computing* 5.2 (2020), pp. 213–222. DOI: 10.1109/TSUSC.2017.2774184.

[7] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. "Drebin: Effective and Explainable Detection of Android Malware in Your Pocket." In: *Proceedings 2014 Network and Distributed System Security Symposium*. Internet Society, 2014. DOI: 10.14722/ndss.2014.23247.

[8] Rowel Atienza. *Advanced Deep Learning with Keras: Apply deep learning techniques, autoencoders, GANs, variational autoencoders, deep reinforcement learning, policy gradients, and more*. Packt Publishing Ltd, 2018.

[9] Sandra Avila, Nicolas Thome, Matthieu Cord, Eduardo Valle, and Arnaldo de A. Araújo. "Pooling in image representation: The visual codeword point of view." In: *Computer Vision and Image Understanding* 117.5 (2013), pp. 453–465. ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2012.09.007. URL: https://www.sciencedirect.com/science/article/pii/S1077314212001737.

[10] Ritesh Bhagwat, Mahla Abdolahnejad, and Matthew Moocarme. *Applied deep learning with keras: Solve complex real-life problems with the simplicity of keras*. Packt Publishing Ltd, 2019.

[11] K. Houtman R. V. Zutphen C. Guarnieri A. Tanasi J. Bremer M. Schloesser and B. D. Graaff. *Cuckoo Sandbox - Automated Malware Analysis*. https://cuckoosandbox.org/. Last Access: 14/11/2022.

[12] K. Houtman R. V. Zutphen C. Guarnieri A. Tanasi J. Bremer M. Schloesser and B. D. Graaff. *CuckooDroid Book*. https://cuckoo-droid.readthedocs.io/en/latest/. Last Access: 14/11/2022.

[13] Gürol Canbek, Seref Sagiroglu, and Tugba Taskaya Temizel. "New Techniques in Profiling Big Datasets for Machine Learning with a Concise Review of Android Mobile Malware Datasets." In: *2018 International Congress on Big Data, Deep Learning and Fighting Cyber Terrorism (IBIGDELFT)*. 2018, pp. 117–121. DOI: 10.1109/IBIGDELFT.2018.8625275.

[14] Vincenza Carchiolo, Michele Malgeri, Mark Philip Loria, and Marco Toja. "An efficient real-time architecture for collecting IoT data." In: *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)* (2017), pp. 1157–1166.

[15] Francois Carrez, Tarek Elsaleh, David Gómez, Luis Sánchez, Jorge Lanza, and Paul Grace. "A Reference Architecture for federating IoT infrastructures supporting semantic

interoperability." In: *2017 European Conference on Networks and Communications (EuCNC)*. 2017, pp. 1–6. DOI: 10.1109/EuCNC.2017.7980765.

[16] François Carrez, Tarek Elsaleh, David Gómez, Luis Sánchez, Jorge Lanza, and Paul Grace. "A Reference Architecture for federating IoT infrastructures supporting semantic interoperability." In: *2017 European Conference on Networks and Communications (EuCNC)* (2017), pp. 1–6.

[17] Patrick P. K. Chan and Wen-Kai Song. "Static detection of Android malware by using permissions and API calls." In: *2014 International Conference on Machine Learning and Cybernetics*. Vol. 1. 2014, pp. 82–87. DOI: 10.1109/ICMLC.2014.7009096.

[18] Zachary Charles and Jakub Konečný. *On the Outsized Importance of Learning Rates in Local Update Methods*. 2020. DOI: 10.48550/ARXIV.2007.00878. URL: https://arxiv.org/abs/2007.00878.

[19] David Charte, Francisco Charte, Salvador García, María J. del Jesus, and Francisco Herrera. "A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines." In: *Information Fusion* 44 (2018), pp. 78–96. ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2017.12.007. URL: https://www.sciencedirect.com/science/article/pii/S1566253517307844.

[20] Ana Cholakoska, Bjarne Pfitzner, Hristijan Gjoreski, Valentin Rakovic, Bert Arnrich, and Marija Kalendar. "Differentially Private Federated Learningfor Anomaly Detection in EHealth Networks." In: *Adjunct Proceedings of the 2021 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2021 ACM International Symposium on Wearable Computers*. New York, NY, USA: Association for Computing Machinery, 2021, 514–518. ISBN: 9781450384612.

[21] Hsin-Yu Chuang and Sheng-De Wang. "Machine learning based hybrid behavior models for Android malware analysis." In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE. 2015, pp. 201–206.

[22] Gianni D'Angelo, Massimo Ficco, and Francesco Palmieri. "Malware detection in mobile environments based on Autoencoders and API-images." In: *Journal of Parallel and Distributed Computing* 137 (2020), pp. 26–33. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.11.001. URL: https://www.sciencedirect.com/science/article/pii/S0743731519302436.

[23] Gianni D'Angelo, Massimo Ficco, and Francesco Palmieri. "Association rule-based malware classification using common subsequences of API calls." In: *Applied Soft Computing* 105 (2021), p. 107234. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2021.107234.

[24] Gianni D'Angelo and Francesco Palmieri. "Enhancing COVID-19 tracking apps with human activity recognition using a deep convolutional neural network and HAR-images." In: *Neural Computing and Applications* (2021). ISSN: 1433-3058. DOI: 10.1007/s00521-021-05913-y. URL: https://doi.org/10.1007/s00521-021-05913-y.

[25] Gianni D'Angelo, Francesco Palmieri, and Antonio Robustelli. "A federated approach to Android malware classification through Perm-Maps." In: *Cluster Computing* 25.4 (2022), pp. 2487–2500. DOI: 10.1007/s10586-021-03490-2. URL: https://doi.org/10.1007/s10586-021-03490-2.

[26] Gianni D'Angelo, Francesco Palmieri, and Antonio Robustelli. "Effectiveness of Video-Classification in Android Malware Detection Through API-Streams and CNN-LSTM Autoencoders." In: *Mobile Internet Security*. Ed. by Ilsun You, Hwankuk Kim, Taek-Young Youn, Francesco Palmieri, and Igor Kotenko. Singapore: Springer Nature Singapore, 2022, pp. 171–194. ISBN: 978-981-16-9576-6.

[27] Gianni D'Angelo, Francesco Palmieri, Antonio Robustelli, and Arcangelo Castiglione. "Effective classification of android malware families through dynamic features and neural networks." In: *Connection Science* 33.3 (2021), pp. 786–801. DOI: 10.1080/09540091.2021.1889977. eprint: https://doi.org/10.1080/09540091.2021.1889977. URL: https://doi.org/10.1080/09540091.2021.1889977.

[28] A. Desnos and G. Gueguen. *Androguard*. https://github.com/androguard/androguard. Last Access: 04/01/2023.

[29] Maharshi Dhada, Amit Kumar Jain, and Ajith Kumar Parlikad. "Empirical Convergence Analysis of Federated Averaging for Failure Prognosis**This research was funded by the EPSRC and BT Prosperity Partnership project: Next Generation Converged Digital Infrastructure, grant number EP/R004935/1." In: *IFAC-PapersOnLine* 53.3 (2020). 4th IFAC Workshop on Advanced Maintenance Engineering, Services and Technologies - AMEST 2020, pp. 360–365. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.11.058.

[30] Gianni D'Angelo and Francesco Palmieri. "Network traffic classification using deep convolutional recurrent autoencoder neural networks for spatial–temporal features extraction." In: *Journal of Network and Computer Applications* 173 (2021), p. 102890. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2020.102890.

[31] Sannara EK, Francois PORTET, Philippe LALANDA, and German VEGA. "A Federated Learning Aggregation Algorithm for Pervasive Computing: Evaluation and Comparison." In: *2021 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2021, pp. 1–10. DOI: 10.1109/percom50583.2021.9439129.

[32] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. "A Survey on Automated Dynamic Malware-Analysis Techniques and Tools." In: *ACM Comput. Surv.* 44.2 (2008). ISSN: 0360-0300. DOI: 10.1145/2089125.2089126. URL: https://doi.org/10.1145/2089125.2089126.

[33] Massimo Ficco. "Detecting IoT Malware by Markov Chain Behavioral Models." In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 2019, pp. 229–234. DOI: 10.1109/IC2E.2019.00037.

[34] Gernot A Fink. *Markov models for pattern recognition*. en. 2nd ed. Advances in Computer Vision and Pattern Recognition. London, England: Springer, Jan. 2014.

[35] Rafa Galvez, Veelasha Moonsamy, and Claudia Díaz. "Less is More: A privacy-respecting Android malware classifier using Federated Learning." In: *CoRR* abs/2007.08319 (2020). arXiv: 2007.08319. URL: https://arxiv.org/abs/2007.08319.

[36] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[37] Guardsquare. *Dexguard | guardsquare*. Last Access: 03/01/2023.

[38] Guardsquare. *Proguard | guardsquare*. Last Access: 03/01/2023.

[39] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory." In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[40] Chet Hosmer. *Polymorphic and metamorphic malware - black hat briefings*.

[41] Ruei-Hau Hsu, Yi-Cheng Wang, Chun-I Fan, Bo Sun, Tao Ban, Takeshi Takahashi, Ting-Wei Wu, and Shang-Wei Kao. "A Privacy-Preserving Federated Learning System for Android Malware Detection Based on Edge Computing." In: *2020 15th Asia Joint Conference on Information Security (AsiaJCIS)*. 2020, pp. 128–136. DOI: 10.1109/AsiaJCIS50894.2020.00031.

[42] Idanr. *Droidmon - Dalvik Monitoring Framework for CuckooDroid*. https://github.com/idanr1986/droidmon. Last Access: 14/11/2022.

[43] Clayton Johnson, Bishal Khadka, Ram B. Basnet, and Tenzin Doleck. "Towards Detecting and Classifying Malicious URLs Using Deep Learning." In: *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 11 (2020), pp. 31–48.

[44] Giannis Karamanolakis, Kevin Raji Cherian, Ananth Ravi Narayan, Jie Yuan, Da Tang, and Tony Jebara. "Item Recommendation with Variational Autoencoders and Heterogeneous Priors." In: *Proceedings of the 3rd Workshop on Deep Learning for Recommender Systems*. DLRS 2018. Vancouver, BC, Canada: Association for Computing Machinery, 2018, 10–14. ISBN: 9781450366175. DOI: 10.1145/3270323.3270329. URL: https://doi.org/10.1145/3270323.3270329.

[45] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. "MalDozer: Automatic framework for android malware detection using deep learning." In: *Digital Investigation* 24 (2018), S48–S59. ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2018.01.007. URL: https://www.sciencedirect.com/science/article/pii/S1742287618300392.

[46] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. "SCAFFOLD: Stochastic Controlled Averaging for Federated Learning." In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 5132–5143.

[47] Kaspersky. *Sandbox*. Last Access: 05/12/2022.

[48] Dimitrios Kelaidonis, Angelos Rouskas, Vera Stavroulaki, Panagiotis Demestichas, and Panagiotis Vlacheas. "A federated Edge Cloud-IoT architecture." In: *2016 European Conference on Networks and Communications (EuCNC)*. 2016, pp. 230–234. DOI: 10.1109/EuCNC.2016.7561038.

[49] Latif U. Khan, Walid Saad, Zhu Han, Ekram Hossain, and Choong Seon Hong. "Federated Learning for Internet of Things: Recent Advances, Taxonomy, and Open Challenges." In: *IEEE Communications Surveys Tutorials* 23.3 (2021), pp. 1759–1799. DOI: 10.1109/COMST.2021.3090430.

[50] Hwankuk Kim. "5G core network security issues and attack classification from network protocol perspective." In: *J. Internet Serv. Inf. Secur.* 10 (2020), pp. 1–15.

[51] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. "Deep Learning for Classification of Malware System Call Sequences." In: *AI 2016: Advances in Artificial Intelligence*. Ed. by Byeong Ho Kang and Quan Bai. Cham: Springer International Publishing, 2016, pp. 137–149. ISBN: 978-3-319-50127-7.

[52] Ajit Kumar, K Pramod Sagar, K. S. Kuppusamy, and G. Aghila. "Machine learning based malware classification for Android applications using multimodal image representations." In: *2016 10th International Conference on Intelligent Systems and Control (ISCO)*. 2016, pp. 1–6. DOI: 10.1109/ISCO.2016.7726949.

[53] Check Point Software Technologies LTD. *MOBILE SECURITY REPORT 2021*. https://www.cybertalk.org/wp-content/uploads/2021/04/mobile-security-report-2021.pdf. Last Access: 28/11/2022. 2021.

[54] Check Point Software Technologies LTD. *SECURELIST | Mobile malware evolution 2020*. https://securelist.com/mobile-malware-evolution-2020/101029/. Last Access: 28/11/2022. 2021.

[55] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444. DOI: https://doi.org/10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[56] Chenglin Li, Keith Mills, Di Niu, Rui Zhu, Hongwen Zhang, and Husam Kinawi. "Android Malware Detection Based on Factorization Machine." In: *IEEE Access* 7 (2019), pp. 184008–184019. DOI: 10.1109/ACCESS.2019.2958927.

[57] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. "Federated optimization in heterogeneous networks." In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 429–450.

[58] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. "Android Malware Clustering Through Malicious Payload Mining." In: *Lecture Notes in Computer Science* (2017), 192–214. ISSN: 1611-3349. DOI: 10.1007/978-3-319-66332-6_9.

[59] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. *Android Malware Clustering through Malicious Payload Mining*. 2017. arXiv: 1707.04795 [cs.CR].

[60] T. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. "Focal Loss for Dense Object Detection." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.2 (2020), pp. 318–327. DOI: 10.1109/TPAMI.2018.2858826.

[61] Yongqiang Lu, Zhaobin Liu, and Yiming Huang. "Parameters Compressed Mechanism in Federated Learning for Edge Computing." In: *2021 8th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2021 7th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*. 2021, pp. 161–166. DOI: 10.1109/CSCloud-EdgeCom52276.2021.00038.

[62] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE." In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

[63] Marco Maggipinto, Chiara Masiero, Alessandro Beghi, and Gian Antonio Susto. "A Convolutional Autoencoder Approach for Feature Extraction in Virtual Metrology." In: *Procedia Manufacturing* 17 (2018). 28th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM2018), June 11-14, 2018, Columbus, OH, USAGlobal Integration of Intelligent Manufacturing and Smart Industry for Good of Humanity, pp. 126–133. ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.10.023. URL: https://www.sciencedirect.com/science/article/pii/S2351978918311399.

[64] Alireza Makhzani and Brendan Frey. *k-Sparse Autoencoders*. 2014. arXiv: 1312.5663 [cs.LG].

[65] Antonio La Marra, Fabio Martinelli, Francesco Mercaldo, Andrea Saracino, and Mina Sheikhalishahi. "D-BRIDEMAID: A Distributed Framework for Collaborative and Dynamic Analysis of Android Malware." In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 11.3 (2020), pp. 1–28.

[66] Alejandro Martín, Víctor Rodríguez-Fernández, and David Camacho. "CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains." In: *Engineering Applications of Artificial Intelligence* 74 (2018), pp. 121–133. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2018.06.006.

[67] Mohammad Masum and Hossain Shahriar. "Droid-NNet: Deep Learning Neural Network for Android Malware Detection." In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 5789–5793. DOI: 10.1109/BigData47090.2019.9006053.

[68] McAfee. *McAfee 2022 Consumer Mobile Threat Report*. https://www.mcafee.com/blogs/mobile-security/mcafee-2022-consumer-mobile-threat-report/. Last Access: 09/01/2023.

[69] McAfee. *McAfee Mobile Threat Report 2021*. Last Access: 09/01/2023.

[70] McAfee. *McAfee Mobile Threat Report Q1, 2020*. https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf. Last Access: 09/01/2023.

[71] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. "Communication-Efficient Learning of Deep Networks from Decentralized Data." In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Ed. by Aarti Singh and Jerry Zhu. Vol. 54. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1273–1282.

[72] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. "Communication-efficient learning of deep networks from decentralized data." In: *Artificial intelligence and statistics* (2017), pp. 1273–1282.

[73] Lingheng Meng, Shifei Ding, Nan Zhang, and Jian Zhang. "Research of stacked denoising sparse autoencoder." In: *Neural Computing and Applications* 30.7 (2018), pp. 2083–

2100. DOI: 10.1007/s00521-016-2790-x. URL: https://doi.org/10.1007/s00521-016-2790-x.

[74] Milad Mohammadi and Subhasis Das. *SNN: Stacked Neural Networks*. 2016. DOI: 10.48550/ARXIV.1605.08512. URL: https://arxiv.org/abs/1605.08512.

[75] MrMalware. *Malware Sample Sources - A Collection of Malware Sample Repositories*. https://github.com/Virus-Samples/Malware-Sample-Sources. Last Access: 10/04/2023.

[76] Lakshmanan Nataraj, S. Karthikeyan, and B.S. Manjunath. "SATTVA: SpArsiTy Inspired ClassificaTion of Malware VAriants." In: *Proceedings of the 3rd ACM Workshop on Information Hiding and Multimedia Security*. IH&MMSec '15. Portland, Oregon, USA: Association for Computing Machinery, 2015, 135–140. ISBN: 9781450335874. DOI: 10.1145/2756601.2756616.

[77] Cong-Danh Nguyen, Nghi Hoang Khoa, Khoa Nguyen-Dang Doan, and Nguyen Tan Cam. "Android Malware Category and Family Classification Using Static Analysis." In: *2023 International Conference on Information Networking (ICOIN)*. 2023, pp. 162–167. DOI: 10.1109/ICOIN56518.2023.10049039.

[78] Dinh C. Nguyen, Ming Ding, Pubudu N. Pathirana, Aruna Seneviratne, Jun Li, and H. Vincent Poor. "Federated Learning for Internet of Things: A Comprehensive Survey." In: *IEEE Communications Surveys Tutorials* 23.3 (2021), pp. 1622–1658. DOI: 10.1109/COMST.2021.3075439.

[79] Lan Huong Nguyen and Susan Holmes. "Ten quick tips for effective dimensionality reduction." In: *PLoS computational biology* 15.6 (2019), e1006907.

[80] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version)." In: *ACM Trans. Priv. Secur.* 22.2 (2019). ISSN: 2471-2566. DOI: 10.1145/3313391. URL: https://doi.org/10.1145/3313391.

[81] Giovanni Paragliola and Antonio Coronato. "Definition of a novel federated learning approach to reduce communication costs." In: *Expert Systems with Applications* 189 (2022), p. 116109. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2021.116109.

[82] R. Parmar. *Training Deep Neural Networks*. https://t.ly/ZKLC. Last Access: 10/04/2023.

[83] Reese Pathak and Martin J Wainwright. "FedSplit: an algorithmic framework for fast federated optimization." In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 7057–7066.

[84] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[85] Mauricio Perez, Sandra Avila, Daniel Moreira, Daniel Moraes, Vanessa Testoni, Eduardo Valle, Siome Goldenstein, and Anderson Rocha. "Video pornography detection through deep learning techniques and motion information." In: *Neurocomputing* 230 (2017), pp. 279–293. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2016.12.017. URL: https://www.sciencedirect.com/science/article/pii/S0925231216314928.

[86] Segun I. Popoola, Ruth Ande, Bamidele Adebisi, Guan Gui, Mohammad Hammoudeh, and Olamide Jogunola. "Federated Deep Learning for Zero-Day Botnet Attack Detection in IoT-Edge Devices." In: *IEEE Internet of Things Journal* 9.5 (2022), pp. 3930–3944. DOI: 10.1109/JIOT.2021.3100755.

[87] Nektaria Potha, V. Kouliaridis, and G. Kambourakis. "An extrinsic random-based ensemble approach for android malware detection." In: *Connection Science* 33.4 (2021), pp. 1077–1093. DOI: 10.1080/09540091.2020.1853056.

[88] T.N. Prabhu. *Exploratory data analysis in Python*. https://towardsdatascience.com/exploratory-data-analysis-in-python-c9a77dfa39ce. Last Access: 02/01/2023.

[89] Valerian Rey, Pedro Miguel Sánchez Sánchez, Alberto Huertas Celdrán, and Gérôme Bovet. "Federated learning for malware detection in IoT devices." In: *Computer Networks* 204 (2022), p. 108693. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2021.108693.

[90] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. "Contractive Auto-Encoders: Explicit Invariance during Feature Extraction." In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Bellevue, Washington, USA: Omnipress, 2011, 833–840. ISBN: 9781450306195.

[91] Md Nazmus Sadat, Md Momin Al Aziz, Noman Mohammed, Feng Chen, Xiaoqian Jiang, and Shuang Wang. "SAFETY: Secure gwAs in Federated Environment through a hYbrid Solution." In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16.1 (2019), pp. 93–102. DOI: 10.1109/TCBB.2018.2829760.

[92] Md. Nazmus Sadat, Md Momin Al Aziz, Noman Mohammed, Feng Chen, Xiaoqian Jiang, and Shuang Wang. "SAFETY: Secure gwAs in Federated Environment through a hYbrid Solution." In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16 (2017), pp. 93–102.

[93] Michael L. Santacroce, Daniel Koranek, and Rashmi Jha. "Detecting Malware Code as Video With Compressed, Time-Distributed Neural Networks." In: *IEEE Access* 8 (2020), pp. 132748–132760. DOI: 10.1109/ACCESS.2020.3010706.

[94] Cícero dos Santos and Maíra Gatti. "Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts." In: *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*. Dublin, Ireland: Dublin City University and Association for Computational Linguistics, 2014, pp. 69–78. URL: https://aclanthology.org/C14-1008.

[95] Sunny Sanyal, Dapeng Wu, and Boubakr Nour. "A Federated Filtering Framework for Internet of Medical Things." In: *2019 IEEE International Conference on Communications, ICC 2019, Shanghai, China, May 20-24, 2019*. IEEE, 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8761381. URL: https://doi.org/10.1109/ICC.2019.8761381.

[96] Jürgen Schmidhuber. "Deep learning in neural networks: An overview." In: *Neural Networks* 61 (2015), pp. 85–117. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0893608014002135.

[97] Selkie. *Parallel Speedup*. http://selkie.macalester.edu/csinparallel/modules/IntermediateIntroduction/build/html/ParallelSpeedup/ParallelSpeedup.html. Last Access: 15/04/2023.

[98] Chi-Sheng Shih, Ching-Chi Chuang, and Hsin-Yuan Yeh. "Federating public and private intelligent services for IoT applications." In: *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*. 2017, pp. 558–563. DOI: 10.1109/IWCMC.2017.7986346.

[99] Sanket Shukla, P D Sai Manoj, Gaurav Kolhe, and Setareh Rafatirad. "On-device Malware Detection using Performance-Aware and Robust Collaborative Learning." In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 967–972. DOI: 10.1109/DAC18074.2021.9586330.

[100] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. DOI: 10.48550/ARXIV.1409.1556. URL: https://arxiv.org/abs/1409.1556.

[101] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. *Unsupervised Learning of Video Representations using LSTMs*. 2015. DOI: 10.48550/ARXIV.1502.04681. URL: https://arxiv.org/abs/1502.04681.

[102] Statista. *Annual number of global mobile app downloads 2016-2021, by store*. https://www.statista.com/statistics/276602/annual-number-of-mobile-app-downloads-by-store/. Last Access: 21/11/2022. 2019.

[103] S. Sumit. *A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way*. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. Last Access: 10/04/2023.

[104] Symantec. *Connect Symantec Archives*. http://www.symantec.com/connect/blogs/yet-another-bunchmalicious-apps-found-google-play. Last Access: 21/11/2022. 2018.

[105] Laya Taheri, Andi Fitriah Abdul Kadir, and Arash Habibi Lashkari. "Extensible Android Malware Detection and Family Classification Using Network-Flows and API-Calls." In: *2019 International Carnahan Conference on Security Technology (ICCST)*. 2019, pp. 1–8. DOI: 10.1109/CCST.2019.8888430.

[106] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors." In: Jan. 2015. DOI: 10.14722/ndss.2015.23145.

[107] Jameson Thies and Amirhossein Alimohammad. "Compact and Low-Power Neural Spike Compression Using Undercomplete Autoencoders." In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 27.8 (2019), pp. 1529–1538. DOI: 10.1109/TNSRE.2019.2929081.

[108] Qiuting Tian, Dezhi Han, Kuan-Ching Li, Xingao Liu, Letian Duan, and Arcangelo Castiglione. "An intrusion detection approach based on improved deep belief network." In: *Applied Intelligence* 50.10 (2020), pp. 3162–3178. DOI: 10.1007/s10489-020-01694-4. URL: https://doi.org/10.1007/s10489-020-01694-4.

[109] VMRay. *Malware Sandbox*. https://www.vmray.com/glossary/malware-sandbox/. Last Access: 05/12/2022.

[110] R. Vinayakumar, K. P. Soman, and Prabaharan Poornachandran. "Deep android malware detection and classification." In: *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2017, pp. 1677–1683. DOI: 10.1109/ICACCI.2017.8126084.

[111] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. "Extracting and Composing Robust Features with Denoising Autoencoders." In: *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: Association for Computing Machinery, 2008, 1096–1103. ISBN: 9781605582054. DOI: 10.1145/1390156.1390294. URL: https://doi.org/10.1145/1390156.1390294.

[112] Hansong Wang, Xi Li, Hong Ji, and Heli Zhang. "Federated Offloading Scheme to Minimize Latency in MEC-Enabled Vehicular Networks." In: *2018 IEEE Globecom Workshops (GC Wkshps)* (2018), pp. 1–6.

[113] Hongyi Wang, Mikhail Yurochkin, Yuekai Sun, Dimitris Papailiopoulos, and Yasaman Khazaeni. *Federated Learning with Matched Averaging*. 2020. DOI: 10.48550/ARXIV.2002.06440. URL: https://arxiv.org/abs/2002.06440.

[114] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. "How to Use t-SNE Effectively." In: *Distill* (2016). DOI: 10.23915/distill.00002. URL: http://distill.pub/2016/misread-tsne.

[115] Fengguo Wei, Yuping li, Sankardas Roy, Xinming Ou, and Wu Zhou. "Deep Ground Truth Analysis of Current Android Malware." In: June 2017, pp. 252–276. ISBN: 978-3-319-60875-4. DOI: 10.1007/978-3-319-60876-1_12.

[116] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 0128042915.

[117] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. "DroidMat: Android Malware Detection through Manifest and API Calls Tracing." In: *2012 Seventh Asia Joint Conference on Information Security*. 2012, pp. 62–69.

[118] Niannian Xie, Fanping Zeng, Xiaoxia Qin, Yu Zhang, Mingsong Zhou, and Chengcheng Lv. "RepassDroid: Automatic Detection of Android Malware Based on Essential Permissions and Semantic Features of Sensitive APIs." In: *2018 International Symposium on Theoretical Aspects of Soft-*

ware Engineering (TASE). 2018, pp. 52–59. DOI: 10.1109/TASE.2018.00015.

[119] Xingyu Xu, Xiaoyu Wu, Ge Wang, and Huimin Wang. "Violent Video Classification Based on Spatial-Temporal Cues Using Deep Learning." In: *2018 11th International Symposium on Computational Intelligence and Design (ISCID)*. Vol. 01. 2018, pp. 319–322. DOI: 10.1109/ISCID.2018.00079.

[120] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. "Federated Machine Learning: Concept and Applications." In: *ACM Trans. Intell. Syst. Technol.* 10.2 (2019). ISSN: 2157-6904. DOI: 10.1145/3298981.

[121] Fanghua Ye, Chuan Chen, and Zibin Zheng. "Deep Autoencoder-like Nonnegative Matrix Factorization for Community Detection." In: *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. CIKM '18. Torino, Italy: Association for Computing Machinery, 2018, 1393–1402. ISBN: 9781450360142. DOI: 10.1145/3269206.3271697. URL: https://doi.org/10.1145/3269206.3271697.

[122] 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. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, 1105–1116. ISBN: 9781450329576. DOI: 10.1145/2660267.2660359. URL: https://doi.org/10.1145/2660267.2660359.

[123] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao yong Zhou, and XiaoFeng Wang. "Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 915–930. ISBN: 978-1-4673-6949-7.

[124] Peng Zhang, Shaoyin Cheng, Songhao Lou, and Fan Jiang. "A Novel Android Malware Detection Approach Using Operand Sequences." In: *2018 Third International Conference on Security of Smart Cities, Industrial Control System*

and Communications (SSIC). 2018, pp. 1–5. DOI: 10.1109/SSIC.2018.8556755.

[125] Tuo Zhang, Chaoyang He, Tianhao Ma, Lei Gao, Mark Ma, and Salman Avestimehr. "Federated Learning for Internet of Things." In: *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. SenSys '21. Coimbra, Portugal: Association for Computing Machinery, 2021, 413–419. ISBN: 9781450390972. DOI: 10.1145/3485730.3493444.

[126] Kunrong Zhao, Tingting He, Shuang Wu, Songling Wang, Bilan Dai, Qifan Yang, and Yutao Lei. "Research on Video Classification Method of Key Pollution Sources Based on Deep Learning." In: *J. Vis. Comun. Image Represent.* 59.C (2019), 283–291. ISSN: 1047-3203. DOI: 10.1016/j.jvcir.2019.01.015. URL: https://doi.org/10.1016/j.jvcir.2019.01.015.

[127] Ying Zhao, Junjun Chen, Di Wu, Jian Teng, and Shui Yu. "Multi-Task Network Anomaly Detection Using Federated Learning." In: *Proceedings of the Tenth International Symposium on Information and Communication Technology*. SoICT 2019. Hanoi, Ha Long Bay, Viet Nam: Association for Computing Machinery, 2019, 273–279. ISBN: 9781450372459. DOI: 10.1145/3368926.3369705.

[128] J. wENG. *Exploratory Data Analysis: A Practical Guide and Template for Structured Data*. https://towardsdatascience.com/exploratory-data-analysis-eda-a-practical-guide-and-template-for-structured-data-abfbf3ee3bd9. Last Access: 02/01/2023.