Formalization of Neural Network Applications to Secure 3D Mobile Applications

Paul Irolla

université PARIS-SACLAY

NNT: 2018SACLS585

Thèse de doctorat

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir

Formalization of neural network to secure 3D mobile applications

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

École doctorale n° 568 Biosigne
Spécialité de doctorat: innovation technologique

Thèse présentée et soutenue a l'ESIEA Laval, le 19 décembre 2018, par

Paul Irolla

Composition du jury:

Éric Filiol Directeur de recherche — Laboratoire CNS	Directeur de thèse
Jean-Philippe Deslys Directeur de recherche — Laboratoire CEA/DRF/iMETI/SEPIA	Co-directeur de thèse
Ludovic Apvrille Professeur — Laboratoire LabSoC (Telecom ParisTech)	Rapporteur
Antonella Santone Professeur — Università degli Studi del Molise	Rapporteur
Maroun Chamoun Professeur — Université Saint-Joseph (Beyrouth)	Président
Akka Zemmari Maître de conférences — Laboratoire LaBRI (CNRS)	Examinateur

université PARIS-SACLAY

ÉCOLE DOCTORALE
Signalisation et réseaux intégratifs en biologie (BIOSIGNE)

Titre: Formalisation et applications des réseaux de neurones à la sécurisation d'applications mobiles 3D.

Mots clés: Détection de Malware Android, Vulnérabilité, Data Mining, Apprentissage Artificiel, Séquence

Résumé: Ce travail de thèse fait partie du projet 3D NeuroSecure. C'est un Projet d'Investissement d'Avenir (PIA) qui vise à développer une solution collaborative pour l'innovation thérapeutique en utilisant des technologies de traitement haute performance (HPC) appliquées au monde biomédical. Cette solution permettra aux experts de différents domaines de naviguer intuitivement dans l'Imagerie Big Data avec accès via des terminaux légers 3D. La protection des données biomédicales contre les fuites de données est primordiale. En tant que tel, l'environnement client et les communications avec le serveur doivent être sécurisés. Cette thèse contribue à l'état de l'art dans la domaines suivants: analyse statique et dynamique d'applications Android; web crawling d'applications ; sélection de caractéristique pour l'apprentissage automatique; algorithme d'entrainement, d'initialisation et d'anti-saturation des réseaux de neurones; Algorithmes formels pour l'utilisation des séquences communes dans un groupe de séquences afin de le caractériser; des prototypes opérationnels ont été conçus pour chaque innovation.

Title: Formalization of neural network to secure 3D mobile applications.

Keywords: Android Malware Detection, Vulnerability, Data Mining, Machine Learning, Sequence.

Abstract: This thesis work is part of the 3D NeuroSecure project. It is an investment project, that aims to develop a secure collaborative solution for therapeutic innovation using high performance processing (HPC) technology to the biomedical world. This solution will give the opportunity for experts from different fields to navigate intuitively in the Big Data imaging with access via 3D light terminals. Biomedical data protection against data leaks is of foremost importance. As such, the client environnement and communications with the server must be secured. This thesis contributes to the state of the art in the following areas: Static and dynamic analysis of Android applications; application web crawling; Feature selection for Machine Learning; Neural Network training, initialization and anti-saturation algorithm; Formal algorithms to use common sequences in a sequence group for characterizing it; Production-ready implementations have been designed for each innovation.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

TABLE OF CONTENTS

List of Figures

LIST OF TABLES

Ackowledgements

I deeply thank Éric Filiol, my thesis supervisor, for giving me his trust to complete this thesis, providing support and critical analysis throughout these 5 years together. I thank Jean-Philippe Deslys, my co-supervisor, for giving me his trust to complete this thesis, his benevolent support and for his wise advices. I thank my colleagues, Baptiste David and Arnaud Banier, for their support, their humor, and valuable advices in mathematics. I thank all my current and former colleagues from ESIEA, for their benevolent presence, their humour and moral support. I thank Antonella Santone and Ludovic Apvrille for agreeing to be reviewers of my thesis.

I thank my parents, Jean-Marc and Élisabeth, for their unconditional love and support. I thank my family, who is a part of who I am. I deeply thank Marianne Thomas, without her I could never have done this long way.

CHAPTER 1

Introduction

Contents

This thesis work is part of the 3D NeuroSecure project [1]. It is an investment project that aims to develop a secure collaborative solution for therapeutic innovation using high performance processing (*HPC*) technology to the biomedical world. This solution will give the opportunity for experts from different fields to *navigate intuitively in the Big Data imaging* with access via 3D light terminals. Biomedical data protections against data leaks are of foremost importance. As such, the client environment and communications with the server must be secured. In the following parts, we detail the 3D NeuroSecure, then we summarize and develop the problematics of this thesis, last we present the research directions that has been chosen as solutions to these problematics.

1.1 3D NeuroSecure

The 3D NeuroSecure project aims toward the opening of high performance processing to the biomedical world, by combining the exploitation of exascale level numerical simulations and 3D preclinical models for the development of new medicines. The ability to process massive volume of data in a decentralized and secured way is a major concern for the years to come. The *Proof of Concept (PoC)* of the 3D NeuroSecure project is an application to the Alzheimer disease. We will use 3D images of whole brains from microscopic scale imagery with atomic scale numerical simulations to select and develop molecules against new therapy targets that have been identified in the Alzeimer disease. The rendering of these 3D images on tablet computer with stereoscopic screen will enable to grasp the abundance and the complexity of the results in an intuitive manner, with a collaborative work mindset on confidential data. These secured tablets will open the path for multiple serious game applications that are not limited to therapy monitoring.

The 3D NeuroSecure project is carried by *Neoxia* [2]. The industrial partners are, Archos *Logic Instrument* [3] and *TRIBVN* [4]. The research laboratory partners are *CEA-DAM* [5], *CEA-DRF* [6], *ESIEA-CVO* [7], *URCA-(CReSTIC* [8] *and MaSCA* [9]*)*.

1. http://www.3dneurosecure.com/
2. https://www.neoxia.com
3. http://logic-instrument.com
4. http://www.tribvn.com
5. http://www-dam.cea.fr
6. http://www.cea.fr/drf
7. https://www.esiea.fr/expertise-confiance-numerique-securite
8. https://crestic.univ-reims.fr
9. http://www.univ-reims.fr/recherche/plateformes-technologiques/maison-de-la-simulation,15923.html

3D NeuroSecure is a global secure solution that knocks down the different barriers that prevent the effective sharing of complex biomedical data and, more widely, of any restricted information. The end-to-end security enables to use the whole potential of decentralized high performance processing on secured light terminals made available to managers and experts that need to analyze critical information and to collaborate remotely. The security aspect of the 3D NeuroSecure *PoC* is obvious, as much for information regarding the evolution of cerebral performance as for the patient evaluations.

The technological models of system protections show at the moment severe limitations. For the protection against malicious codes and attacks, the main reasons of the inefficiency are:

- Regular viral detection based on patern matching requires manual analysis of malware samples. It places the solution in reaction to the attack. Therefore, there is a gap between the initial detection of a new attack and the means to prevent it.
- State-of-the-art techniques based on Machine Learning (for instance [1], [2] and [3]) are not implemented in production for several reasons. A high number of false positives (FP) — even a tiny percentage of FP induces a large amount of false positives, because of the number of benign data volume — and intensive processing times from the binary reversing to the execution of the trained Machine Learning.
- A rapid growth of the virus signature databases that makes the processing more difficult especially on light terminals.

Mobile antiviruses often rely on cloud processing for their detection mecanisms, which makes it dependent on internet. As these terminals could be used in a closed network space, it is a limitation. The nature of the data to be protected restrict the use of cloud processing.

Other security solutions are relying on a secured environment launched in a virtual machine from the host system. It enables — on paper — to have a public environment available and to use secured applications. These VMs often use API that are not mastered to communicate with an host system they have to trust. Regardless, if an attacker takes control over the host system, it can corrupt the VM system or leak data in the case of a targeted attack.

The current solutions, while having practical strengths — as low false positive rates [10] and a low processing time —, have severe algorithmic, design and practical limitations that we have highlighted.

This thesis carry on the work of the *DAVFI (Démonstrateur d'AntiVirus Français et Internationaux)* project made at the *ESIEA-CNS* lab. and funded by *FSN (Fonds national pour la Société Numérique)* [11]. This project ended in september 2014. Its objective is to develop antimalware solutions for Windows, Linux and Android. It is a concern of French national security to rely on sovereign security solutions. Actually, commercial solutions are opaque from a user point of view, they have access to the whole exploitation system and they communicate on a regular basis with foreign servers of editors. A security solution that is not mastered for national security concerns is not acceptable. The *DAVFI* project answers this problematic. The antimalware solution designed for Android relies on a hardened version of the *AOSP (Android Open Source Project)*. Security functions have been added on several layers.

- **Kernel**. protections against unknown code execution and physical attacks.
- **System**. integrity check, dynamic protection of sensitive resources.
- **Applications**. control of application installation. Only certified applications from the trusted store can be installed.
- **Data**. ciphering of user data and protection of authentication resources.
- **Communications**. ciphering of SMS and VoIP without central server (other than the signaling server).

All of these functionalities are extended and exploited in the 3D NeuroSecure project.

10. https://www.csoonline.com/article/2989137/linux/av-test-lab-tests-16-linux-antivirus-products-against-windows-and-linux-malware.html
11. https://www.caissedesdepots.fr/fonds-national-pour-la-societe-numerique

1.2 Research directions

The present work emphasizes the creation of new algorithms, methods and tools that bring advantages over the current state-of-the-art of malware detection and artificial intelligence, but more importantly that can be used effectively in a production context. It is why, what is proposed here is often a compromise between what can be done theoretically and its applicability. Algorithmic and technological choices are motivated by a relation of efficiency and performance results. This thesis contributes to the state-of-the-art in the following areas:

- ***Static and dynamic analysis of Android applications, application web crawling***
  First, to search for malicious activities and vulnerabilities, one needs to design the tools that extract pertinent information from Android applications. It is the basis of any analysis. Furthermore, any classifier or detector is always limited by the informative power of underlying data. An important part of this thesis is the designing of efficient static and dynamic analysis tools for applications, such as a reverse engineering module, a network communication analysis tool, an instrumented Android system, an application web crawlers etc.
- ***Feature selection for Machine Learning***
  The choice of features for Machine Learning is a crucial step for any classification algorithm. An optimized feature set reduces the overall running time and the accuracy of the classifier. We made extensive experimental study to choose the best feature selection/reduction method among a set of candidates. Moreover we observed that the normalization of the feature vector is really sensitive and crucial when it contains several feature type with their own range. We designed a method to treat these problematics, that is presented in this thesis.
- ***Neural Network initialization, training and anti-saturation techniques algorithm***
  Neural Networks are randomly initialized. It is possible to control the underlying random distribution in order to the reduce the saturation effect, the training time and the capacity to reach the global minimum. We have developed an initialization procedure that enhances the results compared to the state-of-the-art. We also revisited the *ADAM* [4] algorithm to take into account interdependencies with regularization techniques — in particular *Dropout* [5]. Last, we use anti-saturation techniques and we show that they are required to correctly train a neural network.
- ***FEA: a very fast classification / feature selection algorithm***
  We have designed an algorithm that enables very fast classification. While not having performance that outmatch state-of-the-art, it is useful to quickly assess the quality of a feature set.
- ***An algorithm for collecting the common sequences in a sequence group***
  The classic way of detecting a sample from a malware family is the manual reverse and analysis of malware samples. From this analysis, the common malicious parts are extracted to make a rule of detection. This algorithm is a new path of research toward the automatic creation of malware family detection rules. Malware code can be represented as multiple sequences of opcodes, and dynamic analysis of malware generate event sequences. Finding similarities in a group of sequences often involves studying their common subsequences or their common substrings. For this task, it is common to study their *Longuest Common Subsequence*. Nevertheless, for real case problems the solution rests upon the relevant common subsequences which are not necessarily the longest. In these cases, the sequence membership to a group is characterized by subsequences of any length. Heuristic algorithms for extracting short subsequences already exist, but no attempt to solve the problem systematically has ever been proposed. We propose a new algorithm for building the *Embedding Antichain* from the set of common subsequences. It is able to process and represent all common subsequences of a sequence set. It is a tool for solving the *Systematic Characterization of Sequence Groups*.

During the course of this thesis, the following tools have been developed:

- ***Glassbox: Android malware application dynamic analysis on real devices***
  Researchers rely on dynamic analysis to extract malware behaviors and often use emulators to do so. However, using emulators lead to new issues. Malware may detect emulation and as a result it does not execute the payload to prevent the analysis. Dealing with virtual device evasion is a never-ending

war and comes with a non-negligible computation cost. To overcome this state of affairs, we propose a system that does not use virtual devices for analyzing malware behavior. Glassbox is a functional prototype for the dynamic analysis of malware applications. It executes applications on real devices in a monitored and controlled environment.

- ***Smart Monkey: automated testing of Android Applications***
  Dynamic analysis of Android application requires user inputs to execute the application functions. Google shipped every Android phone with the *monkey* program that fire random user events to the selected application. Unfortunately, random events make it hard to reach deep functions of the application code. It is why we have developed a grey-box analysis tool that installs and test applications. It is able to identify several fields of interest (email/password etc.) and fill them with believable data. Furthermore it systematically tests all visible functions and do not test a function twice. We experimented Smart Monkey and we show that it covers more code than the monkey program.
- ***Libmla: Machine Learning Algorithms Library***
  Machine Learning starts to be, and will be a science used in every technological domain, without the requirement of an expert. The advancements of Deep Learning avoid the requirement of a feature extraction/selection process. However the learning algorithm and other internal parameters still need to be tweaked. Right now, Machine Learning still needs to be designed and configured by someone with experience to be effective. Last, while Deep Learning is very effective, it requires an astonishing amount of CPU/GPU power. Libmla is another direction of contribution to Machine Learning. We believe we can build IA algorithms with good performance but with far less resource. It is why *Libmla* is a growing open source project with the intent to deliver auto-configured, carefully optimized, efficient and easy to use machine learning algorithms. Right now, libmla is heavily focused of Machine Learning that fit the Malware Detection/Classification domain, but we intent to extend its original frame with actual use cases in other domains.

We presented our work through publications, conferences and articles of scientific vulgarization:

- **International articles**
  - Paul Irolla and Alexandre Dey, *The duplication issue within the Drebin dataset*, Journal of Computer Virology and Hacking Techniques, DOI 10.1007/s11416-018-0316-z.
- **International workshops and conferences with article**
  - Paul Irolla and Éric Filiol, *(In)Security of Mobile Banking... And of Other Mobile Apps*, Black Hat Asia 2015 [12].
  - Paul Irolla, *Glassbox : Dynamic Analysis Platform for Malware Android Applications on Real Devices*, ICISSP ForSE 2017, DOI 10.5220/0006094006100621
  - Paul Irolla. *Systematic Characterization of a Sequence Group*. ForSE 2019.
  - Abhilash Hota and Paul Irolla. *Deep Neural Networks for Android Malware Detection*. ForSE 2019.
- **International conferences without article**
  - Paul Irolla and Éric Filiol, *(In)Security of Mobile Banking*, Chaos Communication Congress 2014 [13].
  - Paul Irolla, *(In)Security of Mobile Banking... And of Other Mobile Apps*, C0c0n 2015 [14].
- **National publications and vulgarization articles**
  - Paul Irolla, *Applications bancaires pour smartphone : une sécurité bâclée*, SecuriteOff 2015 [15].

12. https://www.blackhat.com/docs/asia-15/materials/asia-15-Filiol-InSecurity-Of-Mobile-Banking-wp.pdf
13. https://media.ccc.de/v/31c3_-_6530_-_en_-_saal_6_-_201412272145_-_in_security_of_mobile_banking_-_ericfiliol_-_paul_irolla
14. http://is-ra.org/c0c0n/2015/speakers.php#Irolla_Paul
15. https://www.securiteoff.com/applications-bancaires-une-securite-baclee-12/

- Paul Irolla, *La surveillance mondiale du Wifi*, SecuriteOff 2015 [16].
- Paul Irolla, *User tracking et Facebook : l'ère de la surveillance globale*, SecuriteOff 2016 [17].
- Éric Filiol, Baptiste David and Paul Irolla, *Virus informatiques et autres infections informatiques*, Techniques de l'Ingénieur 2017, Ref H5440 V3.

The thesis is organized as follows. In the second chapter we present in detail the Android system and application internal mechanics, especially the functions that matters for our study. Then we discuss the current state-of-the-art static and dynamic analysis tools for Android applications, and the tools we have designed — namely *Egide*, *Glassbox* and *libmla-android-reverse* module. We expose current ways on collecting samples from public datasets to web crawling, and we present tools we have designed for this task namely *Tarentula* and *Cygea-crawler*. At last, we present un study of vulnerabilities and data leaks in legitimate applications, mobile banking to be precise and bring our conclusions. In the third chapter, we discuss the state-of-the-art of feature extraction, feature selection / reduction and neural networks classification. We present our choices for feature extraction and selection, supported with experimental studies. Then, we detail our neural network configuration, internal parameters and algorithms, supported by experimental studies. In the fourth chapter, we expose a new algorithm for studying the common subsequences between any group of sequences. We then apply it to the problem of malware family detection. Last, we present our full system of antimalware detection and its experimental results. In the fifth chapter, we present the limitations of the current study, the future evolution of the current work and we bring a conclusion.

16. https://www.securiteoff.com/dangers-dune-cartographie-mondiale-points-dacces-wi-fi/
17. https://www.securiteoff.com/user-tracking-facebook-lere-de-surveillance-globale/

BIBLIOGRAPHY

[1] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. Droiddetector : android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1) :114–123, 2016. 14

[2] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *Intelligence and security informatics conference (eisic), 2012 european*, pages 141–147. IEEE, 2012. 14

[3] Brandon Amos, Hamilton Turner, and Jules White. Applying machine learning classifiers to dynamic android malware detection at scale. In *Wireless communications and mobile computing conference (iwcmc), 2013 9th international*, pages 1666–1671. IEEE, 2013. 14

[4] Diederik P Kingma and Jimmy Ba. Adam : A method for stochastic optimization. *arXiv preprint arXiv :1412.6980*, 2014. 15, 53, 104, 118

[5] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1) :1929–1958, 2014. 15, 104

CHAPTER 2

Preliminary studies & background in Android Security

Contents

2.1 Android malware characterization

2.1.1 How to categorize Android malware

Malware is the general term for computer infection. It stands for *malicious software*. It describes a large variety of threats afflicting information and communication systems. Malware and its subcategories have been theoretically formalized by the work of Jürgen Kraus in 1980 [1], Fred Cohen [2], Leonard Adleman [3] and Éric Filiol [4]. However, due to the technical specificities of the Android operating system, we have established a classification dedicated to Android malware. It enables to grasp with precision the intentions of the malware author and the technical differences between malware programs. Motivations for writing a malware program is to obtain an advantage one way or the other. Often, this advantage is of financial aspects. It can be a direct way like tricking the user into purchasing a fake product or an indirect way like collecting personal information. Personal information can be monetized in three way mainly:

- It is internally used for one other activity such as targeted advertising, exploitation of account credentials, etc.
- Sold on the dark web. Actually, identity thieves buy anonymously personal information so we are able to evaluate the price for various pieces of information. The Experian company has released a study of the cost of personal information on the dark web and drawn a prize panel (Figure 2.1, source [1]).
- Sold to data brokers and advertising companies.

We expanded the classification made by Kadir et al. [5]. Their work is focused on financial malware attacks; we included other categories to characterize all malware kinds.

- **SMS malware**. In this category lies malware that sends SMS or call to premium numbers, that subscribes to paying services or that spreads by social engineering.
- **Spyware**. In this category lies malware that collect personal information once, on a periodical basis or in real time. Information collection often includes device technical information, contacts and messages, account credentials, microphone, camera, etc.
- **Scareware**. In this category lies malware that tries to scare users by social engineering and propose an answer to the threat in the form of a paying service or software.
- **Banking malware**. It is malware that steals one-time SMS of the two-factor identification or steals pin code — with tapjacking or a fake banking application.
- **Ransomware**. A malware that locks the phone, often encrypting personal data, and asks for a ransom to unlock and/or decrypt the phone.
- **Dropper**. A kind of malware whose goal is to install other malware either by social engineering or without the user's consent.
- **Botnet**. Malware that is able to control the phone through a backdoor. The infected phone becomes a part of a network — called roBOT NETwork — of controlled phones — called *zombies*.
- **Activism**. Often this type of malware uses SMS to spread propaganda messages to all contacts contained in the phone.
- **Adware**. This malware monetizes users through advertising in a way that is much more aggressive or sneaky than traditional advertising frameworks. For instance, ads for other malicious applications, aggressive advertising placements where the user is forced to click on them, or advertisements inserted in other applications. Often it is accompanied by an abnormally large collection of personal information without the consent of the user.

These are the main categories of malware that is currently found on smartphones. There are nevertheless other marginal motivations for the design of certain malware, as well as cases where the malware is not finalized and it is difficult to know its actual purpose. For these particular instances, we simply use the general category *malware*.

1. https://www.experian.com/blogs/ask-experian/heres-how-much-your-personal-information-is-selling-for-on-the-dark-web/

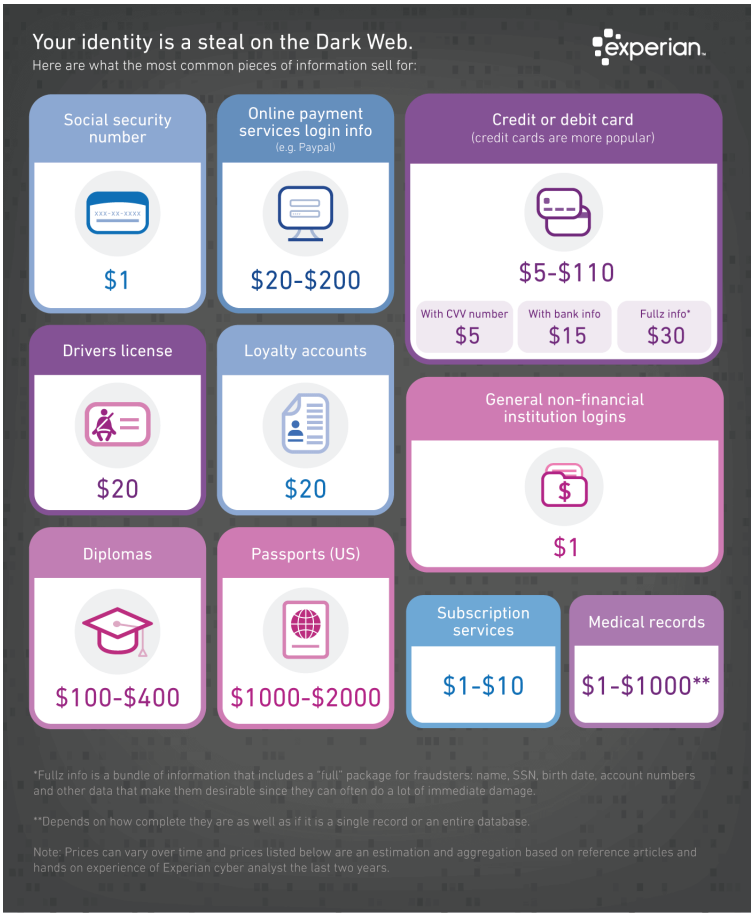


FIGURE 2.1 – Value of personal information on the dark web

We use a second way to categorize malware, through the technical levers it uses to accomplish their task. General trends emerge, so it is possible to draw up a limited list of these technical means.

- **Fake app**. A fake app is a seemingly legitimate app that does not provide any of the services it promises. In reality, it executes a malicious payload.
- **Drive-by download**. To prevent antivirus detection on phones or application markets, the payload is downloaded after installation.
- **Backdoor**. The malware sets up regular communication with a command and control server. This server gives orders following a panel of actions implemented in the application (collect data, send a message, contact a web site etc.), or else directly executes shell commands on the remote machine.
- **Repackaging**. The application is actually an existing legitimate application whose code has been modified to execute a malicious payload. Popular apps are mostly targeted. It is difficult for a user who has installed the application to realize that it is a corrupted version.
- **Root exploit**. The application takes control of the phone thanks to a vulnerability of the Android operating system, allowing it to obtain maximum rights (root) for a limited time or permanently. These vulnerabilities are corrected quickly when discovered. Nevertheless, many users do not update their system. In addition, some manufacturers take a long time to push security updates, besides those old versions of their model no longer evolve. In the latter case, the phone version is blocked and it will never receive security updates.
- **Obfuscation**. Some malware families try to slow the reverse engineering of their code by analysts. This slows down the creation of a detection rule or countermeasure. This set of techniques, called obfuscation, obscures code reading or execution.
- **AV evasion**. These malware implements techniques to pass the antiviral tests, or are in a position to disable or uninstall the antivirus on the user machines.
- **VM evasion**. Google Play has a dynamic application analysis procedure, called *Bouncer*[2]. During this process the application is executed on a virtual machine, and the traces of its execution are collected. These traces enable the detection of potentially malicious applications before they are made available to users. Some malware set up countermeasures to this system. They detect virtual environments and hides their real behavior in this case.
- **Tapjacking**. Some techniques allow to display overlapping windows in a pernicious way. Either the user is not able to clearly understand what is displayed on the screen, or the window is invisible. This allows the malware to force the user to click on another window, such as to grant additional rights to an application, or to record keystrokes.
- **Keylogger**. The malware is able to record the keystrokes. Generally it is used to exfiltrate passwords.
- **APT**. A malware setting up an *Advanced Persistent Threat* is a malware able to rewrite its payload even after deletion in order to maintain an access on the machine.

We classify 279 malware families from 2010 to 2018 with the classification proposed. This is the most important information compilation work on Android malware families to date, to our humble knowledge. Some malware families have unique or unknown motivations. In these cases, they are classified in the general category of *malware*. Our sources for the creation of the table 17 are[3], [4], [5], [6], [6] and [7].

2.2 Android and infection techniques

Android is a market opportunity for malware writers. Featuring an integrated application distribution platform, *Google Play*[7], and many alternative markets, Android has become a prime target. To assess the

2. http://googlemobile.blogspot.fr/2012/02/android-and-security.html
3. https://www.cyber.nj.gov
4. https://github.com/mingyuan-xia/AppAudit/wiki/Android-Malware-Analysis-Report-Collection
5. https://sites.google.com/site/androidmalrepo/
6. https://code.google.com/archive/p/androguard/wikis/DatabaseAndroidMalwares.wiki
7. https://play.google.com

trend of the evolution of Android malware, it is representative to look for the number of malware applications detected by antivirus companies. *Kaspersky Lab.* registered 10 million malware installation from 2004 to 2013, 2.5 million in 2014 and 8.5 million in 2016 [8].

The numbers are low because it is the number of malware installations detected by Kaspersky, it is therefore a subset of the malware applications in circulation (subset of users using kaspersky and subset of malware detected by kaspersky). What is important here is the evolution of the number of malware detections, and these figures are an index of this evolution.It shows that Android malware invasion is evolving in a fast pace.

In this section we explore the reasons for the full growth of malware, by what means they enter on the system and what they seek to obtain. Last, we expose an instance of a threat on the Android platform, the analysis of a sophisticated malware christened *Godless*, which had its moment of glory in 2016.

2.2.1 Malware growth

The use of Android OS has exploded and now is the most used platform in the world. Indeed for website access, Android has overcome Windows [8]. This is particularly due to Asian countries where Android is by far the most used OS, being the cheapest device offering an internet access.

Malware authors have business strategies similar to those of an entrepreneur: to reach a maximum of customers. Therefore, they have naturally turned to mobile apps. Whereas

Android has an official market for distributing applications — *Google Play* — and several alternative markets. It is therefore an opportunity for malware authors to reach customers. They distribute their malware application as any other developer and wait for users to install them. As there is no human verification in these markets, an application with an attractive presentation and a description using popular keywords is sufficient for the spread. Conversely, in the Windows world malware authors often need to rely on shady strategies to spread malware programs (*warez*, *scam*, pornography, etc.).

Moreover, the habits of smartphone users are going against the security of their environment. We are conditioned to install a program whenever we want to use a service. To the contrary, on a personal computer we tend to use web services, so programs that run either on distant servers, either in the browser enclosure. On mobile, an installed program has access, potentially, to the whole phone functions and data. Users therefore tend to the overconsumption of unnecessary applications, which favors the installation of malware because installation is a trivial act. Finally, a smartphone is the device that focuses the most private information: contact list, SMS, personal and professional mails, photos, GPS position, bank accounts, etc. It is a gold mine for malware writers.

2.2.2 Incentive for data leakage: facebook study case

The installation of application for every service, combined with optimization of the network requests for smartphones, leads to data leakage opportunities that are not existing on personal computers. For instance, mail boxes are saved locally, to reduce network requests at its minimum necessary. That is the case for Gmail accounts, a copy of the mails is kept locally and the application only synchronizes those copies with the mail server. The problem is that these mails are stored in plain text on the phone. Any malware application succeeding a privilege escalation is able to recover all mails by simply accessing and reading the mail application directory.

Facebook is a mail box, before being a social network. Mail messages are presented in a chat fashion. The problem that we find with Gmail, we find it with Facebook. We have analyzed in 2016 what was stored on the phone with the Facebook application. After connecting for the first time to the account, we collected data that have been created locally:

```
$> adb shell su -c 'cat /data/data/com.facebook.katana/**/*' > data.dump
```

Once the data is recovered on the host machine, here are some regular expressions that reveal sensitive information:

8. http://gs.statcounter.com/os-market-share

- Listing contacts with the relationship category

```
Regex : /"displayName":"([^"]+)"\}.*?friendshipStatus":"([^"]+)".*?
         "contactType":"([^"]+)".*?"cityName":"([^"]+)"/

Results :
[...]
Name : ###### #####
Status : ARE_FRIENDS
Type : USER
City : Toulouse

Name : #### ######
Status : CANNOT_REQUEST
Type : USER
City : Laval (Mayenne)

Name : ###### ## ###
Status : ARE_FRIENDS
Type : USER
City : Paris
[...]
```

- Listing contacts phone numbers

```
Regex : /(\+33[67][0-9]+)/

Result :
[...]
+33625######
+33628######
+33632######
+33644######
+33648######
+33659######
+33660######
+33663######
+33666######
+33668######
+33674######
[...]
```

- Listing contacts private messages

```
Regex : /ONE_TO_ONE:.+?_[a-zA-Z]+\.(.+?)\{/

Result :
[...]
1377866######:0c713c6e8ea6######/I managed to reverse candy crash for fun
1377866######:0c713c6e8ea6######/But whatsapp seem much harder
1377866######:0c713c6e8ea6######/So much obfuscation
[...]
b2ef924d389f6cf57e2b0ddb85ea######vous faites quoi Lundi soir et mardi soir ?
b2ef924d389f6cf57e2b0ddb85ea######Rien à priori
[...]
```

This availability of personal information on smartphones is an inward attraction for malware authors.

2.2.3 Android OS security measures

Android has several security measures, organized in successive layers. The first is *Bouncer*[9], a dynamic analysis system that run applications in a virtual environment for studying their behavior. It intervenes before the application publication on the Google Play and Google estimates

9. http://googlemobile.blogspot.fr/2012/02/android-and-security.html

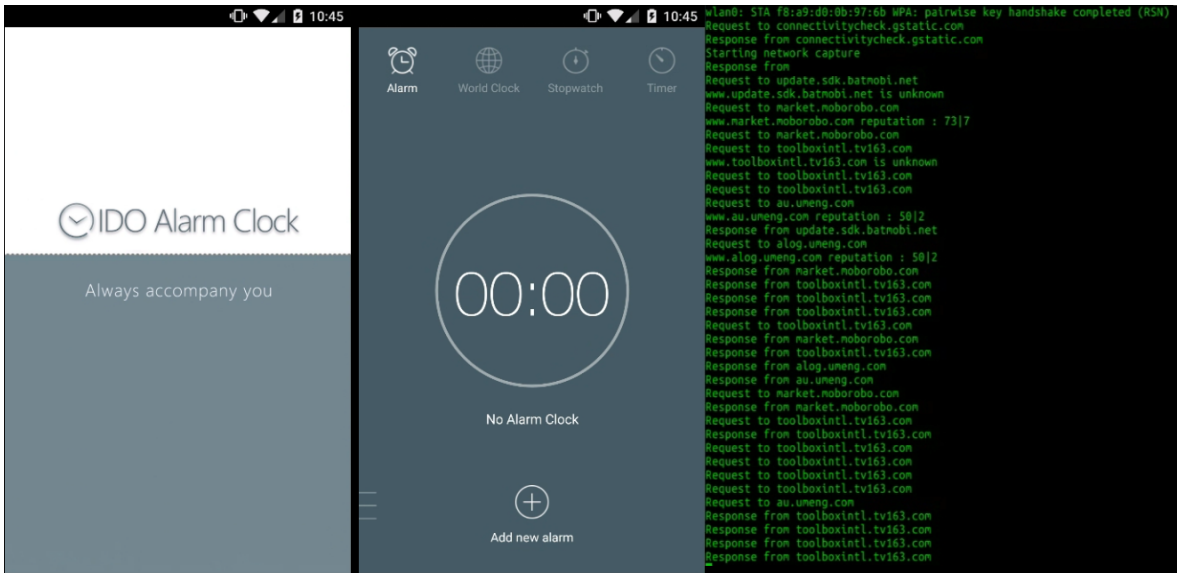


FIGURE 2.2 – (Godless) IDO Alarm Clock, analyzed with *Glassbox*

that it prevents 40% of malware (last information from 2012). Next, at installation the application shows requests for permission of using special or sensitive Android functions. Each permission must also be dynamically validated the first time the application needs it. Therefore, users have different tools at their disposal for access control. Finally, each application runs in a dedicated *sandbox*. This term, in the Android context is to be taken with a grain of salt because this sandbox is not a virtual machine or a container as its name suggests. Each application runs with a single Linux user, who has restricted rights. It is not possible for an application to read or write to system files or to other application files.

There is natively, on Android, a level of security and of control greater than on other OS for the non-specialist. However the flaw lies in the user behavior. Social pressure, comfort or entertainment to use some application narrow the decision-making. Facebook has more than one billion users according to Google Play statistics whereas the application requires quasi-full access on the phone: device and applications history, identity, agenda, contacts, location data, SMS, photos / multimedia / files, storage space, camera, mic, Wi-Fi connection information, device ID and information about calls etc. This ease of giving quasi-full access to application is a boon for malware because users are accustomed to submit themselves to the growing appetites of companies for our private data. Because of user-tracking, there is a interest convergence of companies and malware authors.

2.2.4 Godless analysis

Godless is a malware family that appeared in 2016. It is a interesting case because it is a sophisticated malware. It uses a panel of techniques that can accurately represents what can be a malware Android. Every Godless app is a quality product. People use them being satisfied — unlike most malware. We analyzed two applications of this family, an alarm clock (Figure 2.2) and a multifunction tool (wifi, file sharing, cleaner etc.).

This applications were created specifically for the purpose of distributing a payload. It does not contain exploits or payload, but it downloads them after installation (a.k.a. *Drive-by download attack*). Because of this it passes Google Play checks. The application waits until the phone screen turns off to download a root exploit library, *libtemproot.so* (Figure 2.3).

This library contains various known root exploits access [10].

10. https://github.com/android-rooting-tools

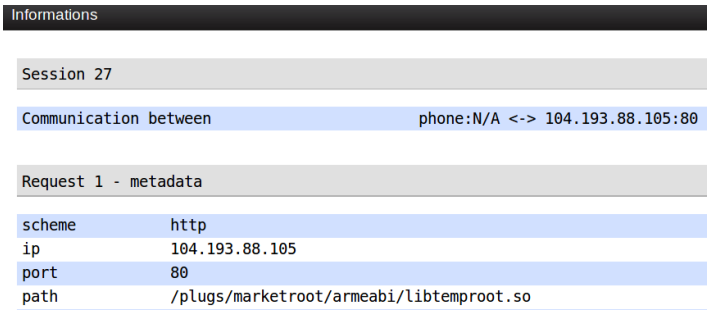


FIGURE 2.3 – (Godless) Drive-by download attack

```
$> readelf -sW libtemproot.so | grep exploit
39: 000038ad  1184 FUNC    GLOBAL DEFAULT    7 attempt_exploit
71: 00003d4d   240 FUNC    GLOBAL DEFAULT    7 attempt_mmap_exploit
72: 00003e3d   112 FUNC    GLOBAL DEFAULT    7 attempt_memcpy_exploit
78: 00003895    12 FUNC    GLOBAL DEFAULT    7 set_fb_mem_exploit_enable
80: 00004b31    18 FUNC    GLOBAL DEFAULT    7 acdb_run_exploit
82: 00004bd1    18 FUNC    GLOBAL DEFAULT    7 fj_hdcp_run_exploit
86: 000052f9    20 FUNC    GLOBAL DEFAULT    7 put_user_run_exploit
90: 00004701    84 FUNC    GLOBAL DEFAULT    7 perf_swevent_run_exploit
91: 00004649   168 FUNC    GLOBAL DEFAULT    7 diag_run_exploit
120: 00004969    52 FUNC    GLOBAL DEFAULT    7 perf_event_run_exploit
130: 0000499d    52 FUNC    GLOBAL DEFAULT    7 perf_event_run_exploit_with_offset
135: 00004edd    38 FUNC    GLOBAL DEFAULT    7 fb_mem_run_exploit
136: 000051d5    42 FUNC    GLOBAL DEFAULT    7 msm_cameraconfig_run_exploit
```

These are the following exploits:

```
CVE-2012-4220 (diag_run_exploit)
CVE-2013-2094 (perf_event_run_exploit)
CVE-2013-2595 (msm_cameraconfig_run_exploit)
CVE-2013-2596 (acbd_run_exploit)
CVE-2013-2597 (fb_mem_run_exploit)
CVE-2013-6282 (put_user_run_exploit)
hdcp_mmap - no CVE (fj_hdcp_run_exploit)
```

Overall this affects all Linux kernel versions less than 3.16.1. It is quite tedious to list with accuracy which version of Android for which manufacturer is vulnerable. But this list can be drawn up for the *AOSP* (*Android Open-Source Project*) which is the reference. [11].

```
Android Version    |API Level  |Linux Kernel in AOSP
-----------------------------------------------------
1.5   Cupcake      |3          |2.6.27
1.6   Donut        |4          |2.6.29
2.0/1 Eclair       |5-7        |2.6.29
2.2.x Froyo        |8          |2.6.32
2.3.x Gingerbread  |9, 10      |2.6.35
3.x.x Honeycomb    |11-13      |2.6.36
4.0.x Ice Cream San|14, 15     |3.0.1
4.1.x Jelly Bean   |16         |3.0.31
4.2.x Jelly Bean   |17         |3.4.0
4.3   Jelly Bean   |18         |3.4.39
4.4   Kit Kat      |19, 20     |3.10
5.x   Lollipop     |21, 22     |3.16.1
6.0   Marshmallow  |23         |3.18.10
7.0   Nougat       |24         |4.4.1
7.1   Nougat       |25         |4.4.1 (To be updated)
```

It can be inferred that — except in special cases for a manufacturer - all versions of Android below 6.0 (*Marshmallow*) are potentially affected. It represents the vast majority of devices. If one of the exploits succeeds the payload is downloaded, otherwise nothing happens for this application.

11. https://android.stackexchange.com/questions/51651/which-android-runs-which-linux-kernel

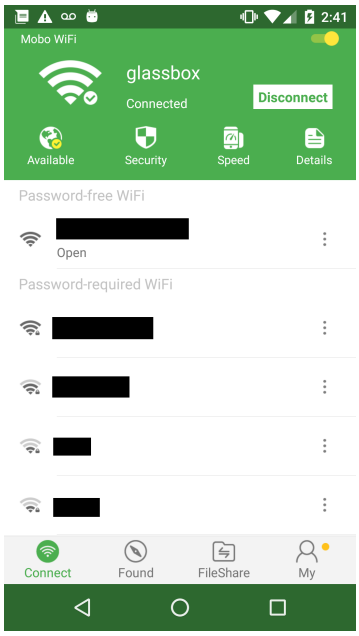


FIGURE 2.4 – (Godless) MoboWIFI app, connected to *Glassbox* access point

This is not the case for the second Godless sample, *MoboWIFI*.

MoboWIFI is a multifunction application with wifi control, file sharing and device cleaning functions (Figure 2.4).

When the user screen turns off, the application runs all exploits contained in *lib/x86/libgodlikelib.so*, which is a library multiple root exploits like *libtemproot.so* in the first sample, except that it is embedded in the application.

If none of the exploits succeed, then the application downloads 5 files:

```
u.smartchoiceads.com/sdk/HostDex_baidu_20170103101149.jar
u.smartchoiceads.com/sdk/pushsdk_20170308173151.jar
u.smartchoiceads.com/sdk/libDaemonProcess_20160520175142.so
u.smartchoiceads.com/sdk/so_20160520175426.png
download.moborobo.com/mobomarket/Apps/Jewel-v2.apk.apk
```

Both jar files are exploits (as java service) that enable the application to become device administrator and install other applications. The next two files are native libraries that enable to start linux daemons that are going to check continuously that the exploit services are started and they restart them if it is not the case. Then the application informs that an update is available (Figure 2.5).

MoboWIFI dynamically loads the Java code contained in the jar files, and the jar files execute the code from native libraries. Then the user sees an Android update popup. The update is actually making an installation request for *Jewel-v2.apk.apk* which is the malware payload. If the user installs it, immediately there is a request to pass the false update as as an administrator application. For a user who has already accepted to install this fake update, there is only a small step to make it admin. In the next section, we address another security point of view which is development flaws that enable a third party to gain advantages for malicious purpose.

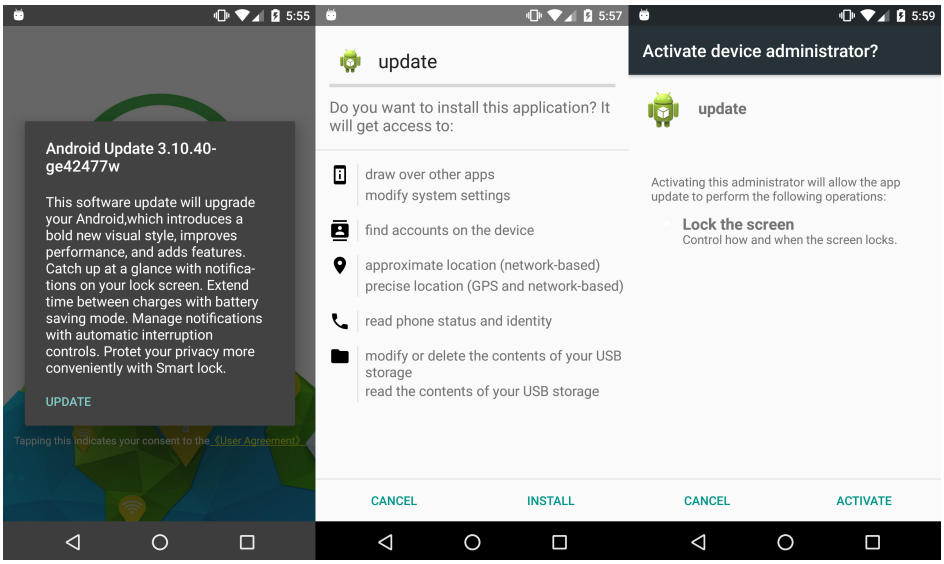


FIGURE 2.5 – Social engineering exploit

2.3 Malware applications in Google PlayStore

We have realized a experimental study to evaluate the number of malware in the PlayStore. It has been achieved with the help of Alexandre Dey, Loic Beheshti, Marie-Kerguelen Sido, Jeannaud Quentin and Sami Hammami; M1 engineer students at *ESIEA* by the time of the study. This work has been the subject of a publication at ForSE 2018 [9]. Despite the protections put in place by Google, it is certain that malware applications exist on the Playstore such as Viking Horde [10], DressCode [12] or FalseGuide [13] attacks have demonstrated. The aim of this study is to determinate the proportion of malware applications on the Google Play. We consider as malware any application intentionally causing harm or subverting the system intended function (this includes *Adware*), as well as manipulating information without user's express consent. To perform this study, we have designed a crawler able to simulate the download of an application and to bypass the downloading restrictions of Google Play. Then, with data from static analysis, we trained an Artificial Neural Network to discover new malware on the PlayStore. This neural network bases its detection on features that are extracted by reversing these applications. We call it Cygea, the complete system that allows you to download applications from the PlayStore, reverse applications, extract features, train a neural network and detect new malicious applications. This study has been realized in 2016.

2.3.1 Cygea PlayStore crawler

To collect set of applications from the Google PlayStore that is large enough to be statistically representative, we developed web crawler designed to massively download applications for the Google Play. This crawler is an open source project available online [14]. The Google PlayStore sets up a number of protections to prevent it from being cloned or analyzed. We explain here what are those protections and how we bypassed them.

First of all, there is no exhaustive list of the applications available on the PlayStore. Moreover, Google tends to bring out the most popular applications and to hide those less successful. Secondly, even though the communication with Playstore servers go through HTTPS, to allow only Android devices with a valid Google account, a proprietary protocol is used on top of it. This pro-

12. https://blog.trendmicro.com/trendlabs-security-intelligence/dresscode-potential-impact-enterprises/
13. https://blog.checkpoint.com/2017/04/24/falaseguide-misleads-users-googleplay/
14. https://github.com/AlexandreDey/cygea-playstore-crawler

```
message AndroidAppDeliveryData {
  optional int64 downloadSize = 1;
  optional string signature = 2;
  optional string downloadUrl = 3;
  repeated AppFileMetadata additionalFile = 4;
  repeated HttpCookie downloadAuthCookie = 5;
  optional bool forwardLocked = 6;
  optional int64 refundTimeout = 7;
  optional bool serverInitiated = 8;
  optional int64 postInstallRefundWindowMillis = 9;
  optional bool immediateStartNeeded = 10;
  optional AndroidAppPatchData patchData = 11;
  optional EncryptionParams encryptionParams = 12;
}
```

FIGURE 2.6 – Google PlayStore download protocol

tocol is based on Google protobuf [15], and has not been officially described as of now, but has been, for the most part, reverse engineered (Figure 2.6).

Finally, any behaviour considered as not "human" (such as downloading a large amount of applications in a short time, or following the same pattern at fixed interval, etc.) is to be detected and leads to an account ban. Furthermore, the criteria to define whether or not a behaviour is normal seems to be changing every few weeks or months. Other researchers have described how they prevent account banning. *Playdrone* [11] crawls the entire Store using Amazon EC2 cloud services and thus connections come from multiple IP addresses. As for *SherlockDroid* [12], they limit the number of downloads by pre-filtering potential malware, and also, each connection goes through *Tor*. The first one require to use Amazon EC2 and the second one does not respond to our problematic, crawling the whole PlayStore. It is the reason why we have developed our own tool. On GitHub, we can find a Python script [16] that is able to communicate with the PlayStore. This script implements the basic functionalities of the store (search for applications, get the details, download, etc.) and needs only two elements to work with: a valid Google account and an Android Device ID (a unique token generated for each Android device). However, the tool is not designed for mass downloading and we modified it to suit our needs. To avoid being banned, we crowd-sourced the creation of 30 activated Google accounts (an Android device is required to validate an account before it is allowed to download from the PlayStore, and you can only create a few accounts from one device) and we try to put as much time as possible between two downloads from the same account (multiple minutes). We also generate an Android ID for every account thanks to *android-checkin* [17]. When crawling, we keep the authentication token as long as it is possible to download applications and we renew the token each time it is invalidated. When an account is banned, the PlayStore servers return an error message when attempting to login. We use a list of HTTPS proxies to prevent our IP address from being blacklisted. To download an application knowing its package name is required, so we builded the list of package names hosted by the PlayStore. To do so, we use the search function with random words selected from a multilingual dictionary. Each search gives at most 250 results. From this point, we download applications that are free (we cannot download paid apps) and compatible with our fake device (we cannot download tablet specific apps without a compatible Android Device ID). To speed up the whole process, we split the dictionary between multiple processes and each of these spawns a thread provided with an account/device ID pair for each app to download (see Figure 2.7).

The crawler is designed to be scalable. Provided that we have enough accounts and Device IDs, it is possible to distribute the crawling across multiple machines. In our case, to synchronize the different processes, we store the applications inside a Cassandra NoSQL database [18] and before downloading a new one, we check if the application is absent from the database or not. The crawler has been used for two purposes. First, in October 2016, we generated a list of more than 2 millions

15. https://developers.google.com/protocol-buffers/
16. https://github.com/dflower/google-play-crawler
17. https://github.com/nviennot/android-checkin
18. http://cassandra.apache.org/

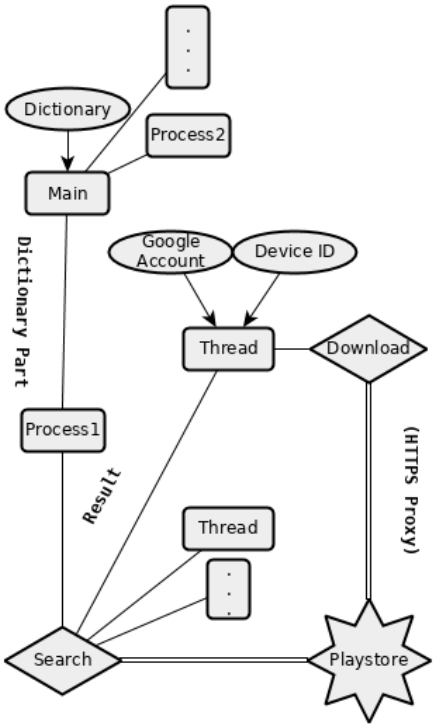


FIGURE 2.7 – Cygea crawler architecture

applications. In January 2017, we used it to download and extract features of 3650 applications, which constitute our sample set for this study. For this second usage, we used a laptop with a 2 core 4 thread intel CPU and the process took 2 weeks. Depending on the application, extracting the features can take up to 15 min while downloading takes less than one minute.

2.3.2 Feature analysis

We perform static analysis, and therefore, we extract all the features that can be interesting for classification directly from the APK file. To reverse applications, we use *Androguard*[19] written in Python. During the static analysis, we process the following information:

- **General Information**
  This information is not directly used to perform the analysis, but more as a way to identify and contextualize it. We extract these information mainly from the application manifest. First, we retrieve the application common name (ex: *Facebook Messenger*) the package name (*com.orca.facebook*), and the version number. It enables to identify the application, but considering it is easy to repackage an Android application, we also use the SHA-256 hash of the APK. We extract the certificate used to sign the application. From this certificate, we can extract interesting information on the developer. We also collect information relative to the SDK version (minimal/maximal/target) and all intents statically defined.
- **Classification Information**
  These are the features on which the detection is based. They are retrieved by decompiling the DEX file in the application. This file store the java bytecode that will be executed by *Dalvik* (or *Android RunTime*). The state-of-the-art of static analysis feature is discussed in the next chapter, so here we expose our process for information purpose. Most of the information we extract is based on the opcodes, without the operands. We store the trigram frequencies. To use several Android API methods, Android applications need to ask for certain permissions (use network, bluetooth, access user information, etc.). Malware apps also

19. https://github.com/androguard/androguard

need to ask for these permissions, and some permissions give access to more potentially malicious behaviour, and are therefore a good way to discriminate applications. Thus we collect permissions from the manifest. Finally, we extract the Android API call sequence. The Android API is a software interface used to access device related capabilities (sending SMS, manipulating the UI, etc.). Some of these tasks can be maliciously employed.

There is a total of 228 dalvik valid opcodes. We observed that applications were constituted on average of 80,000 to 100,000 opcodes. But a few applications go beyond a million opcodes. From these opcodes, we found a list of 500,000 unique trigrams. For the API calls, we recorded a total of 100,000 distinct ones. Finally we found 250 unique permissions. Therefore, if we wanted to represent applications inside vectors, the dimension of them would be greater than 600,000.

Even with neural networks being powerful tools, we must limit the number of entries to have an affordable solution. A common approach is to select the most relevant features based on statistical indexes. To optimize results on malware detection, we took the features most represented in malware while being the least represented in benign applications. A well known statistical method allows us to rank those features with this criterion, the *TF-IDF* indicator [13]. This method, originally used for document searching — especially in search engines —, can be adapted for our purpose. The *TF* or *term frequency* is the frequency of a word in a given text. The *IDF* or *inverted document frequency* the frequency of documents where the term appears. The more a word is represented, the less pertinent it becomes as a matter of selection, the lower the IDF value is. The *TF-IDF* is the product of these two values, thus, the higher the value is the most pertinent the result will be. In our case, things are slightly different. *TF* is the feature frequency in our malware database and *IDF* is the feature frequency in the benign dataset. We have then an indicator that grows if the feature is well represented in malware samples and diminishes if the feature is well represented in benign samples. Overall we chosed the top 12,175 features that exhibit the best *TF-IDF* score.

2.3.3 Classification

We use an artificial neural network to label new applications as malware or benign. To build it, we use a set of applications that are already labeled. We use the Drebin Dataset [14]. The Drebin Dataset is widely used in researcher community. At first we used the 5,585 malware and 2,201 benign applications as our training dataset. We have conducted a study on this initial set (see chapter 3) that shows that 42.5% of these applications can be regrouped into 592 (584 malware, 8 benign) sets of repackaged applications with the same code and only resources and a few strings that differ from one another. Therefore, we removed the duplicates from the Drebin Dataset, and we end up training on a set constituted of 2,891 malware APK and 2,178 benign ones.

Our neural network is a multilayer perceptron. A perceptron is one of the simplest structures in the Neural Network domain but, from previous experimentation using a single perceptron applied to malware detection, we deduced that this structure was sufficient for our problematic. The state-of-the-art of neural networks is presented in chapter 4, in the following paragraph we present the parameters we have used. More detail can be found on chapter 4.

The final neural network parameters are: 1 input and 1 hidden layer set to a sigmoid symmetric activation function , 12,175 entry nodes, 25 nodes on the hidden layer, 2 nodes on the output, expected results for a malware sample are (-0.95;0.95) and (0.95;-0.95) for a benign sample. The learning method is incremental (classic *backpropagation*), the learning rate is set to 0.25 and we use the FANN library on this study [20]. Last, we use the decision rule *winner takes all*. To cope with overtraining we use cross-validation with 20% of the dataset. 20% of the dataset is used to assess the performance of the trained neural network. The remaining 60% of the dataset is used to train the neural network. We selected the neural network state that produced the best results on the validation set.

To assess the performance of our neural network, we use the *accuracy*:

20. https://github.com/libfann/fann

For all experiments, we use the *accuracy* as the main performance measure. It expresses the ratio of samples that are well classified, regarding their expected class (malware or benign):

- tp is the number of *True Positive*, i.e. the number of malware samples classified as such by the model.
- tn is the number of *True Negative*, i.e. the number of benign samples classified as such by the model.
- fp is the number of *False Positive*, i.e. the number of malware samples classified as benign by the model.
- fn is the number of *False Negative*, i.e. the number of benign samples classified as malware by the model.

$$\text{True positive rate}: \quad tpr = \frac{tp}{tp+fn}$$

$$\text{False positive rate}: \quad fpr = \frac{fp}{fp+tn}$$

$$\text{Accuracy}: \quad acc = \frac{tp+tn}{tp+tn+fp+fn}$$

In chapter 3 (part 3.2.2) we present the state-of-the-art of static analysis with sequences of opcodes. Our result accuracy is on the top of the range of the state-of-the-art values (96.88%-98%).

Among the 3650 applications, the neural network classified 92 as being malicious. This represents 2.52% of our sample set. From this, and considering that our classifier is selected to be the most accurate and that false positive rate is roughly equal to false negative rate, assuming proportion of malware follows a normal distribution, using the normal estimation of the interval, we calculate the confidence interval of the proportion of malware on the PlayStore:

$$\mathrm{X} - 1.96\sqrt{\frac{\mathrm{X}(1-\mathrm{X})}{n}}, \mathrm{X} + 1.96\sqrt{\frac{\mathrm{X}(1-\mathrm{X})}{n}}$$

With X our measured proportion and n the sample number. We can estimate with a 95% confidence level that the proportion of malware on the PlayStore sits between 2.01% and 3.03%. A few of these applications were manually tested and proved to be malicious. For instance, the application *Battery King* (now removed from the store) requires many permissions and uses them to leak information. The application also writes binary file on the machine and executes them afterwards. By allocating more resources (more accounts, Device IDs, CPU power, time, etc.) to the crawler, it would be feasible to crawl most of the Google PlayStore applications. The main bottleneck here is the time needed to extract the features from the applications. This is explained by the fact that Androguard is a really complete tool that does more than what we actually need. Considering the classifier, it shows great accuracy for only light computation time. The *TF-IDF* variant employed here also allowed us to target more precisely the important features without the need of deep knowledge in the Android malware domain. Classification capabilities could be enhanced by using a larger training dataset constituted of more recent and complex malware. Finally, we are sure that Google PlayStore houses a certain number of malware, and we estimate that this number was between 40,000 and 60,000 applications in 2017.

2.4 Application vulnerabilities

The security of the 3DNeuroSecure project relies on the security of user Android environment as a whole. Malware is a major part of attack sources, but the security of legitimate applications must also be acknowledged. Threats are multiples: development flaws leading to vulnerability exploitation, aggressive user tracking, backdoors, etc. We draw here a portrait of conceivable threats

by presenting a study of the security of mobile banking application. Mobile banking is supposedly a domain where we expect a special attention to security. Mobile devices are the main intermediary between the user and Internet and there is no single aspect that is not impacted by mobility. Mobile devices are becoming more and more the unique entry point of our life. In this respect two main issues are critical:

- The security of our data and mobile environments. Vulnerabilities can be exploited to enable illegimitate access to them by an attacker.
- The protection of our privacy since personal data may naturally leak to providers/companies that are selling/proposing not only the devices but also different services. Most of the time, these companies and us, as customers, do not share the same interests.

In a mobile device, apps are in fact the most critical parts. We install them more or less voluntarily. Services are becoming available only via mobile application. Moreover as stated in the previous section, social pressure to use some application exists. They can have extensively access to the operating system and our data. They are suspected to do far more than the official actions they are supposed to do. In a word, can we trust them? As far as mobile banking is concerned, those issues are, of course, even more critical. It relates not only to our money and assets but also to all the aspects of our life that are impacted by money: what we buy, what we do, etc. This is the reason why — without loss of generalities — we mainly focused on banking apps in this study. However the reader must keep in mind that from the code perspective there is no fundamental difference with other application domains. Insecurity and privacy are common issues which concern all apps. Vulnerabilities, data leaks, backdoor, agressive user tracking are sometimes hard to define exactly. The line between an acceptable and a unacceptable behavior in the world of legitimate application is not obvious and subjective. To frame how an application can be trustworthy, we designed a Trust Policy that application must follow to be considered trustworthy:

- It does not contain hidden functionalities. *threat covered: backdoors.*
- User information collection must be motivated by explicit functionalities. *threat covered: aggressive user-tracking.*
- Web communications and local data involving personal user information or cryptogragic secrets must be encrypted. *threat covered: data leaks.*
- The app does not contain known vulnerabilities. *threat covered: vulnerabilities.*

This study has been made in 2015 with application analysis that designed, namely *Egide* for static analysis and *Panoptes* for dynamic analysis. These are the precursors of the tools of this current thesis, details can be found in [15]. We first present the results regarding the app which is likely to be the most used application: Facebook, to illustrate our methodology. It represents the worst-case situation in terms of privacy violation and infringement. Then, we present the detailed results regarding the security of banking apps we have analyzed throughout the world. Overall it exposes a panel of threat for legitimate application that we consider as representative. It is worth mentioning that most banks have been contacted to provide (for free) all technical details. Up to now, only a very few have answered but did not take our recommendations into account. As a general observation is very difficult to identify the suitable contact point in a bank. Vulnerabilities of French applications have been corrected. The cases we present here are entry points to highlight wider scale problems in the world of smartphones.

2.4.1 Aggressive user tracking: Facebook case

What is *user tracking*? It involves identifying and characterizing a user and quantifying all his interactions with a point of sale, a website or a mobile application. This optimizes content for users, usually for the purpose of improving sales. User tracking is the set of procedures implemented to collect personal information about consumers and their habits. This information classifies consumers to offer products they are more likely to buy. In addition, it allows companies to measure changes in behavior in the face of a change of environment in order to draw conclusions

FIGURE 2.8 – Facebook user information collection

about human psychology. In other words, *user tacking* enables companies to predict users behaviors.

In order to illustrate how an application may undermine our privacy, a representative example is likely to be the Facebook application. This application is one of the most used app in the world and Facebook by nature contains a lot of user private information. However they are totally unaware of how their privacy is undermined. Facebook collects user-submitted data but what about information collection without the users knowledge? From Facebook security policy[21]: *What kinds of information do we collect? Things you do and information you provide. Things others do and information they provide.* ***Your networks and connections****. Information about payments.* ***Device information****. Information from websites and apps that use our services. Information from third-party partners. Companies owned by Facebook.*

In the previous section we have showed that sensitive personal data are stored in plain text. To expose what exactly Facebook collects, we performed dynamic analysis of *http* and *https* requests with *Panoptes* [15]. The first time the application is launched, a *https POST* request is made to Facebook servers 2.8.

Whereas with *Panoptes* we are able to get the plaintext of *https* requests, the data format is still under a cryptic form. Here is the reverse procedure to apply to determine what is going on:

```
1) Unescape url codes recursively.
2) Parse the output string as a JSON object.
3) Until the data superstructure is entirely reversed
    a) Try to parse each string in the JSON object as a JSON object.
    b) Try to decode each strings which seems to be in a base64 format, then:
        i) Try to unzip the result with gzip if the magic number is 1F8B.
       ii) Read the result string with a Windows minidump reader like WinDBG.
```

At the end, we listed all variables that are exfiltred with their values. Let us summarize what information is leaking towards the Facebook servers:

- Bootloader used.
- Device model/manufacturer/serial/hardware/ROM.
- CPU model/architecture/version + Kernel version.
- Screen settings.
- The complete list of system applications.

21. https://www.facebook.com/about/privacy/

- All environment variables.
- Open file descriptors count.
- Software and hardware file descriptors limits.
- Locations settings, developer settings, lock pattern settings.

```
LOCK PATTERN ENABLED=1
LOCK PATTERN SIZE=3
LOCK PATTERN VISIBLE=1
```

- Application settings.
- Security settings.
- Sound used for alarm alert.
- Spell checker settings and Screensaver settings.
- Notification settings (including used sound).
- Battery settings (including current energy level).
- Sounds/music settings, camera settings, Wifi connection settings.

```
connection = WIFI
connection class = POOR
network extra info = Panoptes-AP
```

- Sdcard and memory size/free space/used space.
- Usual user tracking info (timestamp for each user action)

Reports have been made that Facebook applications (including *Messenger* and *Facebook Lite*) are also collecting SMS, call history[22]. This report from *Ars Technica* appears credible because of actual data proving it. Other reports states that Facebook collects private photos from phones, but we could not find soundproofs in theses reports or by ourselves. So we are not able to confirm or deny this statement. In conclusion, the user tracking of Facebook is very aggressive. From another point of view, if Facebook was not a known company, its application would be labeled as malware. We already show in a previous study that the Facebook application stores personal information in plaintext. Here we expose its aggressive user tacking strategy. This application is definitely not compliant with our trust policy.

2.4.2 Geolocation implications: the Sberbank case

Smartphones are able to locate us even indoors. A smartphone seeks to connect to the three closest satellites with GPS and locates itself by trilateration. When it cannot find them or does not have a GPS transmitter, it uses the three closest base transmitter stations. But if nothing is available, for instance in a building with thick walls, it relies on Wifi trilateration. In order for this last option to work, location providers like Google need to know the location of Wifi access points — but it is relevant for Apple, Microsoft and Yandex as well. How do location providers get these data? We have analyzed the Sberbank application, and it is through this analysis that we got our answer.

This application offers additional services such as the location of the nearest automatic ATM that enables cash withdraw at a Sberbank account. But unlike the apps we know, it does not use Google Maps to display the destination. It is its Russian counterpart that it uses, Yandex Maps. The network communications analysis revealed a plaintext request, on the Yandex servers, containing the MAC addresses and the signal strength of the surrounding wifi access points (Figure 2.9). The server response is a file whose header is *"FoundByWifi"* followed by coordinates corresponding to the location of the nearest ATM.

22. https://arstechnica.com/information-technology/2018/03/facebook-scraped-call-text-message-data-for-years-from-android-phones/

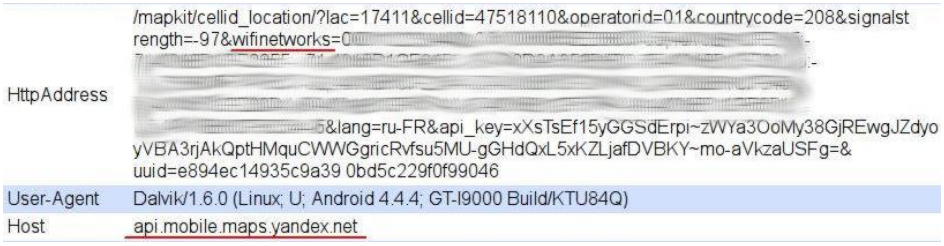


FIGURE 2.9 – Yandex wifi collection

It was inside a building that the analysis was done. Looking at the server response, the location has not been made by GPS or Cell ID. It is by trilateration of nearby wifi access points (*AP*) that the location happened. This raises two questions. Firstly, does it make sense to scan surrounding wifi AP and to send this information in plaintext over the network? This question being rhetorical, let us move on to the second question, how is it possible to locate a phone with wifi AP?

With the received signal power and three wifi APs it is possible to approximate the position of the user — i.e. by trilateration. However the position of these wifi APs must be previously known. In our case we strongly doubt that Yandex came to France to locate the building AP. Google Maps uses the same process for indoor geolocation, with the exception that the application also retrieves SSIDs from access points and communications with Google servers are encrypted.

In 2012, a report [23] from the *Federal Communications Commission* claiming that Google Cars, these self-driving cars that roam the world to build Google Street View, were also scanning wifi AP and unencrypted web communications in their wake.

One would think that this is how Google built a gigantic database of wifi access points. However, it seems unlikely that this is the main method. Indeed other localization operators such as Apple, Combain, Microsoft, Mozilla or Yandex do not have access to the same methods as Google and yet they have also managed to build a colossal database of APs. On the other hand, Google Cars do not make it easy to update this database, while the global wifi mapping is constantly evolving.

The solution is simpler, cheaper and smarter. Cartographic applications like Maps actually collect data from wifi AP. Coupled with the latest reliable GPS or Cell ID data from smartphones of other users and ours, it is possible to trilaterate these access points. Since almost all smartphones contain a geolocation application, some of which are provided with the phone and cannot be uninstalled, we always have a agent that maps the surrounding APs on behalf of a large company.

The dangers of such a practice lie in information overlapping. This is what Glenn Wilkinson shows us with the *snoopy-ng project* at *BlackHat* in a presentation entitled *"The Machines That Betrayed Their Masters"* [16], rightly so. The speaker shows his audience photos of streets, houses and workplaces; these are places regularly frequented by spectators. He is able to map the countries where the audience comes from, who has attended BlackHat for many years and more. How is it possible?

Spoopy-ng is a program installed on a drone swarm, collecting Wi-Fi information, like AP and users wifi communications. The speaker actually uses this program on his personal machine to listen to wireless communications from smartphones in the room.

When a smartphone or laptop is not connected to an access point, it actively searches for access points that it knows, and automatically connects to it if it finds one. The device therefore continuously sends out a list of its known access points and awaits a response from one of them. Anyone listening to this wifi noise is able to retrieve this AP list (SSID). By storing this kind of information over time, and thanks to overlaps, it is possible to deduce from a smartphone — and by extension a person — its address, its place of work, its travel habits, its social circles, family, etc. It all depends on the duration and the perimeter of listening to wireless communications.

23. http://transition.fcc.gov/DA-12-592A1.pdf

In the case of *snoopy-ng*, this is a proof of concept and what it can achieve is already impressive. Now back to our initial problem, location operations like Yandex, Google, Apple and Microsoft have a sort of *snoopy-ng* agent for every of their user. The intelligence power on every user they have at their disposal by cross-checking information is absolutely colossal. And this only covers information about wifi AP, which are just a drop in the ocean of data collection related to the use of connected devices.

It is possible to exclude APs from the automatic cartography of several agents. For this purpose, we must add *"_nomap"* to SSID of its access point, it is a code recognized by Google and Mozilla that will not use these access points in their mapping. Since there are currently no laws regulating this information and standards, this code is not recognized by other operators. For Android users with Google Maps, wifi location can be disabled in phone settings. Finally, to prevent our smartphone from displaying the list of known AP, there are two solutions: turn off the wifi when not in use and / or remove the dispensable AP from the list.

On a global scale, we have entered a society where interactions with a connected device is recorded, quantified and automatically analyzed. This case shows that even with legitimate application it is sometimes hard to draw a line on what is considered as acceptable behaviors and what is not. Data leakage and spying is also a matter of viewpoint.

2.4.3 Backdoor: JPMorgan case

Backdoor — in a legitimate application — is a flaw in the development process of the application that enables its designer to take a partial or total control of the smartphone where it is installed. Most of the time when we are talking about a backdoor, it means that the designer of the application has deliberately inserted a piece of code that can subvert the underlying system. In some cases it is hard to tell if that kind of development flaw is a *feature*, an *error* or a *dramatic consequence of a bad architecture choice*, but we must consider it as a backdoor — i.e. a deliberate choice to own a remote access to the system.

The present case is the account management application of the JPMorgan investment bank. When the app is first launched, it sends queries to servers owned by JPMorgan. The application receives the content of a variable named *signature* as response. The content length seemed too long to be the signature of an object because commonly signatures are cryptographic hash with a fixed length that usually range from 128 to 512 bits. So, we intrumented the application to see how this signature is used. To achieve this task, we used *APImonitor*[24]. It reverses the application, adds monitoring routines around targeted methods then it compiles back the application. Monitored methods have their arguments and return value dumped onto the *logcat*, which is the centralized logging system of Android. The results is shown on Figure 2.10.

In fact, a decryption routine is called on the message. It contains several sub-messages separated a specific pattern. The first sub-message is actually a signature, and then comes a list of application names that use root rights. This information enabled to find with precision the part the source code that process this data, by seeking the split pattern. This revealed an interesting discovery, one of his sub messages is used and executed as a shell command if it is present. In addition, if the phone is rooted, this command is executed with root privileges (Figure 2.11).

These shell commands are actually used by a security framework that check if the phone is rooted, which could indicate that a malware is present. The ability to execute shell commands allows JPMorgan to quickly respond to new threats by performing additional tests. However this process can be easily be used to take control of the phone remotely and in a targeted way. That is why it must be considered a backdoor.

A few months after JPMorgan was informed of this discovery, the application received an equally interesting update that is supposed to correct the backdoor. So we analyzed this new application:

24. https://github.com/pjlantz/droidbox/tree/master/APIMonitor

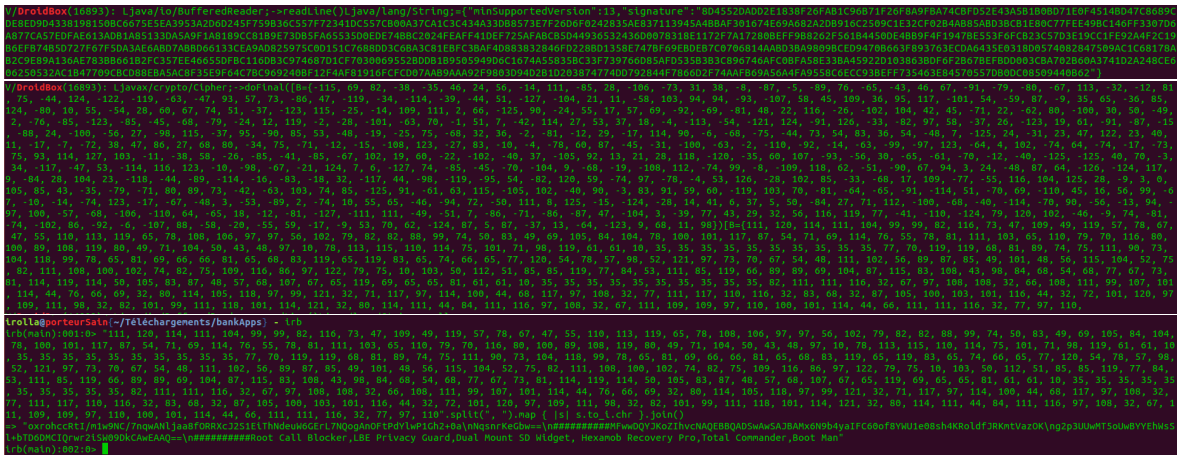


FIGURE 2.10 – Interception of an encrypted message

```
String[] arrayOfString = str2.split("###########");
[...]
RootTools.log("Executing sh Commands :  " + arrayOfString5[0] +
arrayOfString5[1]);
[...]
List localList = runcmd(arrayOfString6);
```

FIGURE 2.11 – Abitrary execution of shell commands received from JPMorgan servers

- Our https communication analysis tool does not work anymore on this version. It is therefore impossible to know if the application still receives encrypted messages.
- The APImonitor tool no longer works either, when the application is instrumented it crashes.
- The java code related to the security framework has disappeared but a native library (cross-compiled c code for Android) has appeared. The security code has been deported to the library, making it more difficult to read.

The reverse engineering of the library has not produced anything significant, however, it is difficult to be 100% sure that the backdoor has not taken another form. Many efforts have been made to prevent us from analyzing this application, which maintains the doubt. We would not install this application on our phone, but everyone will make one own opinion.

2.4.4 Vulnerability to MITM attacks: Budgea, Bank of China cases

MITM — stands for *Man In The Middle* — is a category of attack where the attacker secretly has found a way to act as intermediary between two parties communicating. At the most basic stage of *MITM* attacks, the attacker listens to communications and transmits requests and responses to the original recipients. In the most advanced stages, it corrupts messages to execute a payload on the recipient computers. These kinds of attacks are countered by a proper implementation of cryptography in communications.

One the first banking application, Budgea, is an app that aggregates bank accounts from different companies and countries. That is to say that with a single login and password, the information from all accounts is centralized. We noticed, when studying how the app is handling user accounts, that the *ALLOW_ALL_HOSTNAME_VERIFIER* option is used for https connections. This option serves to bypass the domain verification of the contacted website. This means that the application accepts any valid certificate and therefore an attacker able to position himself between

the application and the server — *Man In The Middle attack* — can redirect the connection to his server and thus recover the login and password of the target. This option is supposed to be used only during the development phase of the application, to facilitate functional testing. Officials have been warned of the problem and the flaw is now corrected.

The Bank of China case is interesting because the exploitation of the flaw is not as obvious as in some other cases we present here. The application does not use the playstore updates — either Google Play or Baidu App Store, its chinese counterpart — instead, it relies on its own update mechanism. When the app is started, it checks for updates on a server owned by Bank of China. If the app is not at it last version, it displays a link on an official market to download the last version. The app retrieves the link from the server response. The issue here is that all this process is done entirely in plaintext. A *MITM* attacker could corrupt the communication in order to send arbitrary links to the user. In particular, it could exploit the confidence of using a banking application to trick the user to install a malware that is visually similar to the bank application. It is called *social engineering*, i.e. the manipulation of people into performing unintended action — here installing a malware — or divulging information.

2.4.5 Vulnerable third party code: BNP, Canara Bank cases

Bad implementations of secure connections or account management between the application and the server is very rare for mobile banking. Banks know for long time how to implement this kind of functionalities, securely. Problems arise when a part of the development is contracted out.

For instance, the BNP bank promotes its additional services — insurance, alarm systems, loans, etc. — through its application. The problem is to entrust this work of advertisement distribution to an external company, *Ad4Screen*. The development of Android is still young and this kind of development is well mastered for websites. Hence, developers reuse the website code directly in Android application, through a web interface — the *WebView* — that enables the execution of *html*, *css* and *javascript*. Javascript code has the possibility to call Android API methods with the *addJavascriptInterface* function of the *WebView*. The problem here is that the web code is dynamically loaded from the *Ad4Screen* servers in plaintext. An *MITM* attacker is therefore able to corrupt the communication and execute arbitrary javascript code that have restricted access to the Android API. Moreover, obsolete versions of Android (API < 4.2, which is about 20% of users currently according to the latest Google report on distribution statistics), the function *addJavascriptInterface* contains a vulnerability that allows the code javascript to execute arbitrary shell commands or call any function of the Android API (Camera, SMS, Calls, app install etc.). After contacting the bank about this vulnerability, it is now fixed.

One of the oldest banks of India, Canara Bank, is surfing the wave of *passbook* / *bankbook* applications. This is an app that summarizes account information by including the balance and the latest moves quickly. However, it does not make it possible to generate bank tranfers. According to Google Play, the app counts between 100k and 500k downloads and the last update is from June 2014. What is wrong with this app? All, absolutely all communications are in clear! Anyone monitoring the network can collect login credentials, among others. Even better, when connecting to the server it displays a tutorial page that lists the commands for which the server responds, with the arguments to use and even an example (Figure 2.12).

The available functions range from *AccountSummary* to *getCreditCardNumbers* and *Get_Balance*. It is perfect for a hacker wishing to automate the process. A whois on the IP address of the server reveals that it does not even belong to the bank, but to a service company in Mumbai. The company offers solutions ranging from dedicated server service management to mobile application development. This is what happens when a serious work is entrusted to an unscrupulous service company without checking the result.

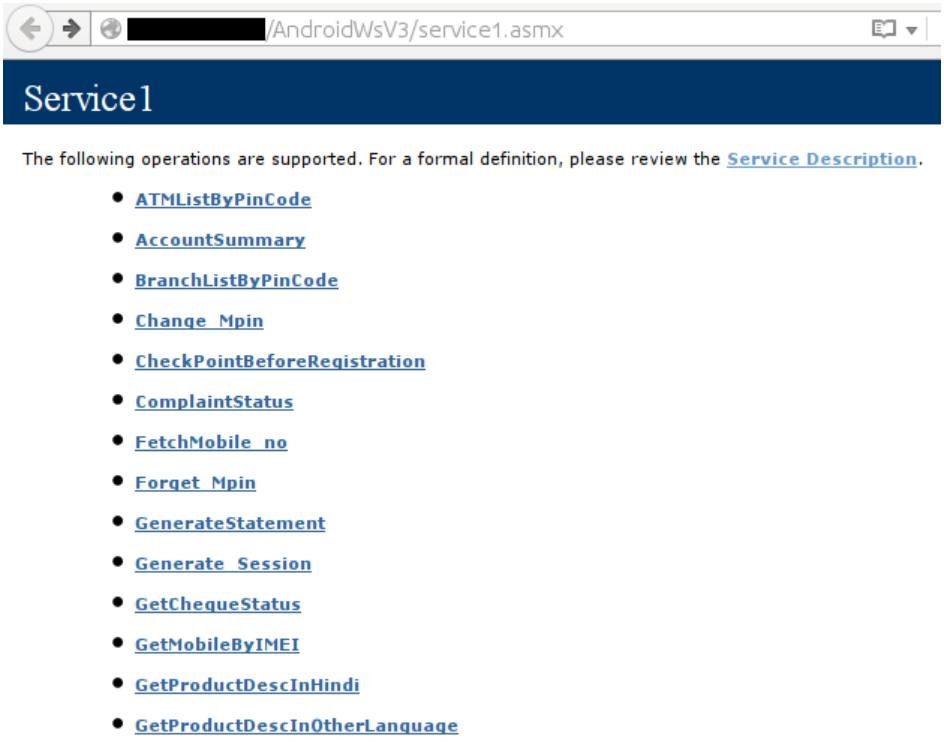


FIGURE 2.12 – Canara Bank: vulnerable server

```
D/SessionHandlerRSA( 2968):  body to encrypt:
[username=7388484&password=1353&include_spbi_accounts=true
&requestId=6d087e0a-1160-4aa4-b3af-b4ae0e03a994]
```

FIGURE 2.13 – Debug mode enabled in production application

2.4.6 Plaintext sensitive data: Norwegian mobile banking case

The use of cryptography, for sensitive application, should be used not only for communications but also for data locally stored. The user environment should be considered as a hostile one, and as a consequence no trust should be accorded to it.

We had the opportunity to analyze almost all Norwegian banking applications. They are built with few exceptions around a handful of models. The differences between several applications of the same model are the logo and the connection address to the bank server. All models are safe except for one that is used by 6 applications: *Bank2*, *Vekselbanken*, *Storebrand*, *Cultura Bank*, *Totens Sparebank*, *Drangedal Sparebank*. When a connection error to the server happens, the application writes the login, the password and the encryption key of the password in plaintext in the *logcat*! (see Figure 2.13).

To provoke a network error not being particularly complicated, the question is: how to recover this data in the log files? It is possible for applications to read the logs of other applications on Android versions below 4.1. For other versions, the most direct way is to flash the phone, install a root access (by installing a Cyanogen ROM for example) and use tools to recover deleted files. It is strongly recommended to encrypt the phone data to protect against attackers who may have physical access to the device (this option is available in the security settings of Android).

2.4.7 Conclusions

It is still more interesting to talk about a vulnerable application than a well-implemented and well-secured application. The vast majority of mobile banking application that we analyzed were entirely safe to use. We expect high standard of security for mobile banking, so as we found anyway a handful of serious vulnerabilities, we expect finding a greater amount of them in regular applications. In terms of geographical distribution, overall we see a slightly higher security awareness of Asian banks compared to European and American banks. Indeed we could see more regularly among Asian banks:

- Additional encryption measures for the password. Usually the only encryption used is that of communication (https). We suspect that Asian countries do not trust entirely trust the *https* protocol.
- A particular care to protect themselves from retro-engineering and dynamic analysis tools. For instance, we could note more often the use of *certificate pinning*. This is a set of strategies to ensure that the *SSL* certificate of the contacted server is signed by a trusted authority recognized by the server and not by an authority that could have been maliciously embedded in the client system list. These techniques put a stick in the wheel of tools we rely on for interception of *https* communications.

For managing bank accounts, we recommend using a regular computer through a web browser. Smartphones should be considered a less secure environment than a computer and therefore avoid using services / data whose security is vital. The phones are not less safe because they suffer from a bad design, on the contrary the permission system of Android has no equivalent on other OS and it limits most of the capacities of malicious actions.

However it is our use of the phone that makes it less secure:

- It is constantly moved, forgotten and manipulated by others.
- It concentrates a lot of personal information and therefore arouses covetousness.
- We install applications when we could use online services. However, many services are only available through an application.
- It becomes mainly an object of entertainment and consumption and incidentally remains an object of communication, then it undergoes the inconsistent behaviors of its users — compulsive installation of gadget applications. The false impression of applications being free is also a problem in this sense: the real cost is measured here in bytes of personal information and available brain time.

There are still technical solutions to strengthen the security of our phone:

- Encrypt phone data.
- Use an antivirus product.
- Use open-source alternative applications when they exist. There is an alternative Android app market of Open Source applications called *Fdroid*[25].
- Common applications are not particularly aware of security issues, and may be the vector of attacks. It is difficult to expect a serious security on a phone that also mixes entertainment, consumption and social networks. Hence, a careful selection of what should be installed reduce infection vectors.

Last, it seems to us highly challenging to hope automating the process of vulnerability discovery as there is no obvious similarity of shapes between them or apparent recurring patterns. The process of application validation can be supported by semi-automatic tools that organize information for a fast look up of anomalies, but we chose to keep it manual.

25. https://f-droid.org

BIBLIOGRAPHY

[1] Jürgen Kraus. Selbstreproduktion bei programmen. *University Dortmund (Feb 1980) http ://vx. netlux. org/lib/mjk00. html as of 21 oct 2007*, 1980. 22

[2] Fred Cohen. Computer viruses. *Computers & security*, 6(1) :22–35, 1987. 22

[3] Leonard M Adleman. An abstract theory of computer viruses. *Advances in Crypto*, 1998. 22

[4] Eric Filiol. Viruses and malware. In *Handbook of Information and Communication Security*, pages 747–769. Springer, 2010. 22

[5] Andi Fitriah Abdul Kadir, Natalia Stakhanova, and Ali A Ghorbani. Understanding android financial malware attacks : Taxonomy, characterization, and challenges. 22

[6] Yajin Zhou and Xuxian Jiang. Dissecting android malware : Characterization and evolution. In *security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012. 24, 59

[7] Ken Dunham, Shane Hartman, Manu Quintans, Jose Andre Morales, and Tim Strazzere. *Android Malware and Analysis*. Auerbach Publications, 2014. 24

[8] Kaspersky Lab. Mobile malware evolution. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07180734/Mobile_report_2016.pdf, 2016. 25

[9] Alexandre Dey, Loic Beheshti, and Marie-Kerguelen Sido. Health state of google's playstore. 2018. 30, 63

[10] Andrey Polkovnichenko, Oren Koriat, und Check Point Research Team, et al. Viking horde : A new type of android malware on google play. *Blog Post*, 2016. 30

[11] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 221–233. ACM, 2014. 31

[12] Axelle Apvrille and Ludovic Apvrille. Sherlockdroid, an inspector for android marketplaces. *Hack. lu, Luxembourg*, 2014. 31, 60

[13] Jeremy Z Kolter and Marcus A Maloof. Learning to detect malicious executables. In *Machine Learning and Data Mining for Computer Security*, pages 47–63. Springer, 2006. 33

[14] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin : Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014. 33, 48, 49, 59, 117, 137, 152, 154

[15] Paul Irolla and Éric Filiol. (in)security of mobile banking...and of other mobile apps, 2015. 35, 36

[16] Glenn Wilkinson. The machines that betrayed their masters, 2014. 38

CHAPTER 3

Sample & Feature Collection

Contents

3.1 Sample Collection

Heuristic methods for malware detection are generally coming from the Data Mining domain. So they need to be based on sound statistical data to generalize on new data effectively. In other words, we need a representative sample set to have an excellent statistical reference set. So we worked on collecting of Android malware massively. This topic is rarely discussed in scientific articles on malware detection or just flown over. However this is a central topic in the context of heuristic detection, especially if we intend to build a heuristic antivirus which is competitive with the state of art in Android malware detection. The question that arises from this observation is *how AV companies collect their malware samples?* Their main sources are most likely sample submission by customers/users/Virus Total and database sharing with other antiviral companies. We note that the databases of the tool Andrubis were fed 70% through the exchange of samples with the antiviral companies [1] (Table II, page 15). The second-largest source is labeled *Source Unknown* and is actually made of applications uploaded by individuals and companies. These are sources we, unfortunately, cannot access. Several academic institutions have malware sample share programs. We have identified and contacted each institution offering these programs. In addition, we have also collected samples from nonacademic datasets. Finally, we collected new samples by ourselves with web crawling techniques.

3.1.1 Places to Find Android Samples

Most of the datasets we present here did not exist at the beginning of this thesis. They appeared mostly between 2015 and 2017. Currently, it is quite easy to gather enough samples to conduct a study even on a large scale. We started with a few thousand malicious samples from *Contagio-Dump*[1] and around two thousand benign samples from *Fdroid*[2]. It was difficult to access more malware samples because IT security companies were reluctant to share their viruses. So we chose at the time to find samples that had not yet been discovered on the web through a mass collection program. We were able to get about 300,000 applications and we estimate that there are about 20,000 malware programs in this set. This estimate is based on average infection rates of different alternative markets. By the time we got these results, public malware datasets appeared whose samples were perfectly labeled. So we abandoned this direction to use these datasets instead. Here is an updated list of these datasets:

- (Academic) **Drebin dataset**[3] [2]
  5585 labeled malware samples.
- (Academic) **AndroTracker dataset**[4] [3]
  4554 unlabeled malware samples, 40292 benign samples.
- (Academic) **M0Droid dataset**[5] [4]
  200 malware samples, 200 benign samples.
- (Academic) **Android Malware Dataset**[6] [5]
  24553 malware samples divided into 71 malware family.
- (Academic) **AndroZoo**[7]
  Millions of unlabeled malware and benign samples. We downloaded up to 500000 malware samples and 500000 benign samples.
- (Public) **Virus Share**[8]
  35397 unlabeled malware samples.
- (Public) **Contagio**[9]
  1168 malware samples with known malware family (but it requires manual reorganization). We designed scripts to automate mass download from the contagio dropbox, and to mass unzip the password-protected archives.
- (Public) **Koodous**[10]
  Millions of malware samples. It requires the use of an API and sample download is limited. We do not have used this source for this thesis.

3.1.2 Our contribution: Tarentula

It is possible to find Android apps on medias much less recommendable than the Google Play. And the goal of Tarentula tool is to explore these areas where it is assumed that the risk of encountering a malicious application is greater. We list the following channels of application sharing:

- **Alternative application markets**. These application repositories do not usually have virus countermeasures, test procedures, or special constraints for developers. Thus, it is obvious that the risk of encountering malware is higher than Google Play.

1. http://contagiodump.blogspot.com/
2. https://f-droid.org/en/
3. https://www.sec.cs.tu-bs.de/~danarp/drebin/
4. http://ocslab.hksecurity.net/andro-tracker
5. http://cyberscientist.org/m0droid-dataset/
6. http://amd.arguslab.org/
7. https://androzoo.uni.lu/
8. https://virusshare.com/
9. http://contagiodump.blogspot.com/
10. https://koodous.com/

- **Wild FTPs**. A significant amount of FTP are indexed by search engines. With a Google search phrase like `[intext:"index of" intext:"parent directory" intext:"name" intext:"last modified" intext:"size" intext:"description" intext:"apk" -site:com -inurl:php]`, we have access to tens of thousands of public FTPs that contain Android apps. There is obviously no control over the content here. It is therefore a source of choice for us.
- **Torrents**. On torrent sharing sites, it is possible to find paid Android games, lots of applications of all kinds (usually warez). It is on this channel that we hope to find many malware applications, for obvious reasons.

Tarentula launches four simultaneous threads.

- The first one processes free proxy sharing sites like *Hide My Ass*[11]. Each of the proxies is tested. In particular, what is verified is that the public ip has been changed and is not leaked in the metadata of the web requests, and that the bit rate is acceptable. Some proxies do not allow access to all websites — especially Chinese proxies. When web crawling, if more than three times a page is not accessible, another proxy is tested. If this new proxy works, the old proxy is rejected from the viable proxy list.
- The second thread collects application download links on public ftps and alternative markets such as *appchina*, *hiapk*, *nduoa*, *3gyu* or *angeeks*. The third thread collects Android application sharing torrents from sites like *Kickass Torrent*.
- The last thread continuously downloads applications from the list of web crawling links with a rotation on viable proxies. Tarentula ran on one month on a regular computer, allowing us to download around 300,000 applications.

3.1.3 Biases of Academic Datasets: the Drebin Study Case

We noted that academic datasets suffer from biases. It is important for the rest of the study to discuss the quality of our sample collection. The *Drebin dataset* [2] is the most supplied academic dataset of Android malware. Therefore it is the most used dataset in research papers on Android malware detection. The research community is using it for evaluation and comparison of their algorithms. We discovered that 49.35% of samples in this dataset has at least one other sample that is a repackaged version containing exactly the same sequence of opcode. The only differences between the original malware and the duplicated ones, in all cases, are the resources embedded and some strings in the code. For assessing the performance of malware detectors or classifiers, a part of the dataset is used for this purpose. So a major part of the testing set end up being the same samples that have been used in the training set.

This situation can lead us, the research community, to overrate the performance of algorithms we are designing. In the worst case, it leads us to wrong conclusions and wrong directions for future research. In this section, we present in detail the issue of duplication the *Drebin dataset*. Then we conduct an experiment where we test several classification algorithms on the *Drebin dataset* with and without the duplicates. Our results show that depending on the classifier the full dataset can lead from moderately (124%) to strongly (172%) *underrated inaccuracy*, and the order of performance of the algorithms is modified. Finally, we provide the list of unique malware samples from the *Drebin dataset*, available on *Github*[12]. This study has been the object of a publication in 2018 in JICV [6].

Duplicates analysis

According to Google Scholar statistics, the Drebin paper has been cited 705 times. Moreover, it has the second position in the ranking of the most cited papers for the *"Android malware"* keywords. It appears to be one of the major papers in Android malware detection, and the *Drebin*

11. https://www.hidemyass.com
12. https://github.com/paul-irolla/drebin-nodup/tree/master

TABLE 3.1 – Distribution of application sequence sets

Set size	Number of sets	Set size	Number of sets
1 (unique sample)	2181	2	280
3	119	4	39
5	35	6	25
7	12	8	7
9	8	10	4
11	4	12	5
13	7	14	5
16	3	17	3
18	2	19	2
20	1	21	3
22	2	25	1
29	1	34	1
35	1	36	1
37	1	38	1
42	1	44	1
53	1	54	1
55	1	62	1
83	1	92	1
96	1	120	1
128	1	-	-

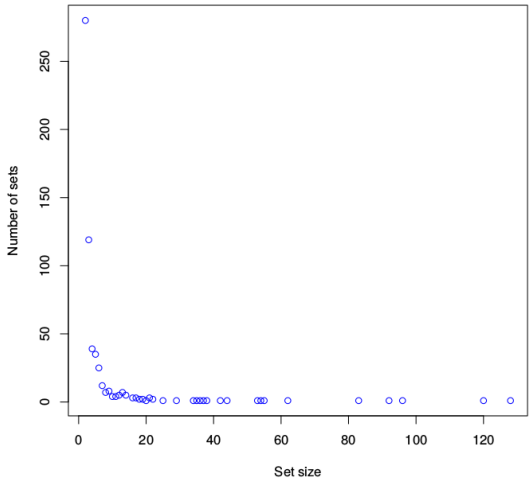


FIGURE 3.1 – Distribution of application sequence sets (size > 1)

dataset is the most widely used Android malware dataset in the research community. Researchers rely on this dataset to evaluate and compare the algorithms they are designing. Analyzing the opcodes sequences of Drebin samples led us to a discovery. A large part of the dataset is samples that share the exact same opcodes sequence. From a statistical point of view, it means that samples are interdependents. Therefore all performance measures made on the *Drebin dataset* are flawed and cannot represent the capacity of the evaluated algorithm to generalize on new data. The link between repackaged applications in the *Drebin dataset* has been studied in [7], so the fact that it contains repackaged is known. Otherwise, this fact has not been raised as an issue for previous or actual research.

Our dataset contains 5,585 malware from the *Drebin dataset* and 2,201 benign samples from *Fdroid*[13], a free and open source repository of Android applications. We use data from static analysis to feed the machine learning algorithms, namely the *n-gram* frequencies of opcodes sequences. The static analysis of Android application implies the reverse of Android applications to Java byte codes. These Java byte codes constitute a sequence of instructions that can be executed by the *Android RunTime*[14], the Android Java virtual machine. We distinguish the operator byte code, called opcode, from the operand.

- Thanks to *androguard*[15], the applications from the whole dataset were decompiled (excluding a few which raised unsolved exceptions). Once decompiled, it was possible to collect each opcode in order of appearance. For each application, the sequence was stored in a file.
- The program *fdupes*[16] was used to find the duplicates among the sequence files.
- For each set of duplicates, a script was used to compare the first element of the set to all others. To do so, applications were unzipped and *diff*[17] was used to determine which files are common within all archives.

Decompilation with androguard generated 7,625 files (5,459 malware, 2,166 benign) containing opcodes sequences for applications in the database. We grouped together equal sequences, to constitute a set. To summarize the results we gathered the sequence set sizes and quantity for the *Drebin dataset* (Table 3.1 and Figure 3.1).

13. https://f-droid.org/
14. https://source.android.com/devices/tech/dalvik/index.html
15. https://github.com/androguard/androguard
16. https://github.com/adrianlopezroche/fdupes
17. https://www.gnu.org/software/diffutils/

More precisely, *fdupes* indicated that 3,278 malicious applications and 31 benign ones are concerned by the duplication problem. These applications were sorted into 584 sets for the malware and 8 sets for the benign applications.
A list of apk files whose codes are different was generated and is available on *Github*[18]. At last, 2,765 malware apps are unique (2,181 unique samples and 1 app chosen for each 584 malware duplicate set), which is 50.65% of the *Drebin dataset*. It means that 49.35% of the *Drebin dataset* is actually applications that have a duplicated source code.

Moreover, the analysis with *diff* highlighted three common cases:

- Applications share everything but their certificates.
- Only images and configuration files change.
- Binary files appear or disappear from the application.

Note: it may happen that *diff* found a difference between the *.dex* files (the file containing the dalvik bytecode) from two duplicated applications. We investigated and discovered that, despite sharing the exact same sequence of opcodes, classes in the code are named differently and therefore, the *.dex* are different.

Impact of the duplicates

The scope of this study is not about assessing the optimality of the data pre-preprocessing step, so the details are given for information purposes only. The features used for training the machine learning algorithms are the normalized frequencies of opcodes trigrams.

$$f_n = \frac{f}{f_{max}}$$

Where f is the frequency of a trigram, f_{max} is the maximum trigram frequency in the sample and f_n is the normalized frequency. The normalization spreads the feature values between 0 and 1 and increases the accuracy of machine learning.

This produces millions of unique features. We use the *Weka*[19] software to test the performance of several algorithms. As the *Weka* implementation only accepts the full feature vectors (sparse matrix processing is not available), it is necessary to reduce the input data dimensionality. We select the most pertinent features according to their *Mutual Information* (a.k.a. *Information Gain*) [8] and their appearance frequency in the sample set. The first criterion is defined as follows:

$$\mathrm{MI(X;Y)} = \sum_{y \in \mathrm{Y}} \sum_{x \in \mathrm{X}} p(x,y) \log \frac{p(x,y)}{p(x), p(y)}$$

Where $p(x,y)$ is the *joint probability distribution function* of X and Y, and $p(x)$ and $p(y)$ are the *marginal probability distribution functions* of X and Y respectively.

In our case X and Y are two variables defined, as follows:

$$\forall f \in \mathrm{F},\ \forall s \in \mathrm{S},\ \mathrm{X} = \begin{cases} 1 & \text{if } f \in s\mathrm{F} \\ 0 & \text{else} \end{cases}$$

$$\forall s \in \mathrm{S},\ \mathrm{Y} = \begin{cases} 1 & \text{if } s \in \mathrm{M} \\ 0 & \text{else} \end{cases}$$

Where:

- F denotes the features set of all samples.
- S denotes the samples set.
- sF denotes the features set of the sample s.

18. https://github.com/paul-irolla/drebin-nodup/tree/master
19. https://www.cs.waikato.ac.nz/ml/weka/

- M denotes the set of malware samples.

MI, in our case, quantifies the correlation between the presence of a feature and the fact that a sample is a malware. A feature can be strongly *MI-correlated* with maliciousness but rarely be found, in which case it does not globally contribute to the classification. It is why we select the top 5% *MI-correlated* and the top 5% most frequent features. After feature selection, each sample has a feature vector size of 21379.

Lastly, we need to define the performance measures we are using:

For all experiments, we use the *accuracy* as the main performance measure. It expresses the ratio of samples that are well classified, regarding their expected class (malware or benign):

- tp is the number of *True Positive*, i.e. the number of malware samples classified as such by the model.
- tn is the number of *True Negative*, i.e. the number of benign samples classified as such by the model.
- fp is the number of *False Positive*, i.e. the number of malware samples classified as benign by the model.
- fn is the number of *False Negative*, i.e. the number of benign samples classified as malware by the model.

$$\text{Accuracy}: \quad acc = \frac{tp+tn}{tp+tn+fp+fn}$$

The last percentages of accuracy is the most important and the most difficult to get. Between a first solution that cures 98% of a disease and a second solution that cures 99% of the same disease, the difference is that with the first solution, twice the number of patients die. So we defined a measure that characterizes this point of view. *Inaccuracy underrating* denotes the loss of *accuracy*, for a machine learning algorithm, from training with the duplicates to training without the duplicates. It is defined as follows:

$$inaccuracy\ underrating = \frac{1-accuracy_{nodup}}{1-accuracy_{dup}}$$

Where $accuracy_{dup}$ is the $accuracy$ with the duplicates and $accuracy_{nodup}$ is the $accuracy$ without the duplicates. In our illustrated example, if the second solution was inaccurate and had the same performance as the first, its *inaccuracy underrating* would be 200%.

In order to test the impact of the duplication problem, we use *weka* to classify sequences from the Drebin Dataset. *Weka* allows us to test various classification algorithms. For each algorithm, we validate the corresponding model using *k-fold cross validation* [9] with k set to 10. To determine the impact of the duplication problem, we focus on the accuracy rate (percentage of correctly classified instances). Our goal is not to provide a comparison between the various classification algorithms and thus, we did not try — more than necessary — to maximize accuracy by optimizing the machine learning hyperparameters. We use the following algorithms:

- Random Forest [20]
- Instance Based [21]
- SMO [22]
- C4.5 Decision Tree [23]
- REPTree [24]

20. http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/RandomForest.html
21. http://weka.sourceforge.net/doc.dev/weka/classifiers/lazy/IBk.html
22. http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/SMO.html
23. http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/J48.html
24. http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/REPTree.html

TABLE 3.2 – Accuracy with and without the duplicates

Algorithm	Accuracy (no dup.)	Accuracy (dup.)	Inaccuracy underrating
RF	97.34%	98.15%	143.78%
IBk	88.08%	92.41%	157.04%
SMO	95.50%	97.39%	172.41%
C4.5	93.65%	95.16%	131.61%
REPTree	96.25%	96.98%	124.17%
ANN	96.27%	97.47%	147.43%
Average	94.51%	96.26%	146.07%

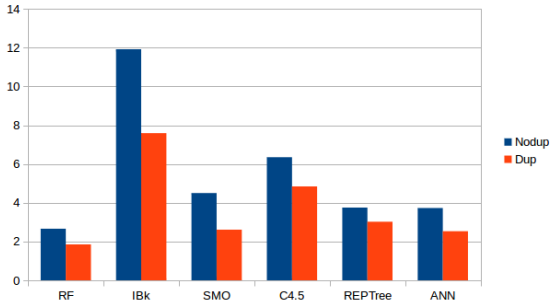


FIGURE 3.2 – Inaccuracy (%) with and without the duplicates

- Artificial Neural Network:
  Custom implementation. Configuration (information purposes): perceptron 3-layers, ADAM [10] learning algo. (same hyperparameter values as the reference), Dropout 50% on hidden neurons only, $2 * \sqrt{n}$ hidden units (n is input size), L2 regularization, custom weight initialization, sigmoïd transfert function (all neurons), stochastic gradient descent, early stopping.

The results are presented in Table 3.2 and Figure 3.2.

As we can see from the table, removing the duplicates from the training dataset multiplies the inaccuracy from 124.17% to 172.41% depending on the classifier (146.07% average). The Support Vector Machine with Sequential Minimal Optimization underrates the inaccuracy the most because of the duplication, with an underrating of 172.41%. Moreover, it changes the performance order of the algorithms.

With duplicates we have:
RF > ANN > SMO > REPTree > C4.5 > IBk

Without duplicates we have:
RF > ANN > REPTree > SMO > C4.5 > IBk

This result shows a few algorithms which seem to outperform others when it is not the case in reality. It could change the conclusions of an article, because some algorithms are more affected by the duplicates than others.

Our experiments over the *Drebin dataset* show that 49.35% of the applications share their code with one or more application(s) in the dataset. We found that the differences between applications with the same code can be classified into three categories:

- The signature has been modified.
- Images or layout modification.
- Binary files.

The classifiers are trained to recognize data from the learning set, so keeping the duplicates can have an impact on the performance (erroneously high accuracy), because they will be on the test set as well. We conducted tests in order to determine this impact. We discovered that using the dataset with all the duplicates underrates the inaccuracy up to 172%.

This experiment brings a reasonable doubt over research results on the *Drebin dataset*. Most must have overrated performance on new applications, and some may bring invalid conclusions. Furthermore, it means that we need to assess the datasets we are using. We need to question their pertinence if we want realistic performance results. We have provided a list of unique applications from the dataset. It is available on *Github*[25]. We recommend using this list for any research

25. https://github.com/paul-irolla/drebin-nodup/tree/master

using the *Drebin dataset*, and for those who want to test the impact of the duplication on their algorithms. Secondly, it shows that academic datasets must be assessed and their limits must be questioned.

3.2 Static Analysis

Static analysis is the study, often automated or semi-automated, of an executable or a source code without executing it. In the domain of software development, static analysis often involves automated tools that review code correctness, optimize it and warn about common development mistakes. In the domain of malware detection, executables are compiled source code. Hence in our domain, static analysis always implies the reverse engineering of Android executables. Then static analysis can be very different whether it is used for extracting features for machine learning or if it is used as a support for manual analysis. We cover both aspects in the following subsections, because our objectives are twofold: malware detection and vulnerability discovery. To reverse an Android application, we need to detail its construction. In the first subsection, we expose the composition of an Android package. In the second subsection we present the state-of-the art of static analysis and Android reverse engineering. In the last subsection we present our contributions, with the static analysis feature we use for the rest of this thesis.

3.2.1 Application Package

The **A**ndroid **P**ac**K**age (*APK*) is the package file format used by the Android operating system for the installation of applications. It contains the compiled Java source code, resources, assets, certificates and the manifest file. An Android application package presents itself as a zip archive file with the *.apk* extension.

Source Code

While is possible to develop Android applications with multiple languages (HTML5 [26], Ruby [27], Python [28], C# [29] etc.), the generated code will always be Java code as it is what the Android system can execute. The Android Java code has nonetheless the capacity to execute native code via compiled shared libraries (*.so*) via the **J**ava **N**ative **I**nterface (*JNI* [30]) or HTML/CSS/JavaScript via the WebView [31] class. This normalization of the Android source code is an opportunity for malware detection as we can rely on a specific format and constraints for application analysis. Source code is compiled into Java Bytecode (*.class*) by the Java compiler then the dexer tool — *dx*, which is part of the Android SDK — compile the class files into a binary file in the DEX format. This DEX file contains Dalvik Bytecodes, it is a Java Bytecode variant optimized for the Android System. The whole source code is packed into one file, decreasing the overall size [32]. All strings, especially package/class/method names are indexed. The DEX file format is illustrated in Figure 3.3 (source [33]).

Dalvik is register-based contrary to the standard *JVM* which is stack-based. As processors also use registers, the Dalvik code is more easily mapped to the CPU registers. Moreover the stack of the *JVM* need a memory access, which is orders of magnitude slower than registers. This effect is balanced by capacity of the **J**ust **I**n **T**ime (*JIT*) compiler to turn stack operations into register operations.

26. https://cordova.apache.org/
27. http://www.rubymotion.com
28. http://pyzia.com/technology.html
29. https://github.com/dot42/api
30. https://developer.android.com/training/articles/perf-jni
31. https://developer.android.com/reference/android/webkit/WebView
32. http://pallergabor.uw.hu/common/understandingdalvikbytecode.pdf
33. https://www4.comp.polyu.edu.hk/~csxluo/DexHunterHITCON15.pptx

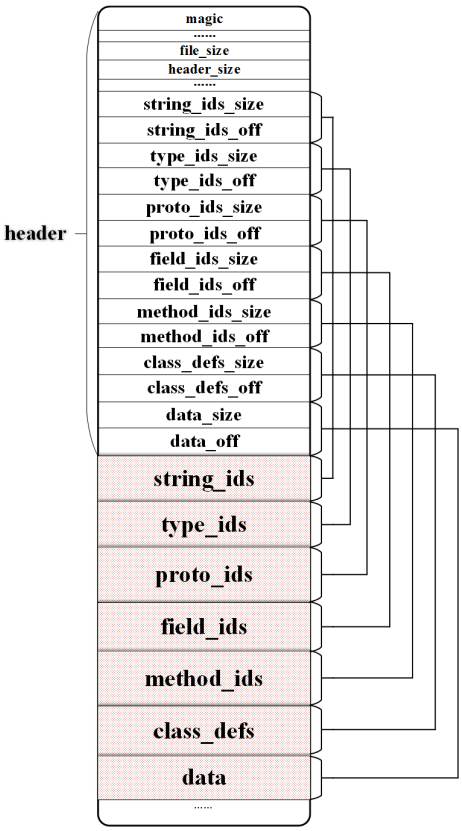


FIGURE 3.3 – DEX file format

Manifest

The Android manifest file — *AndroidManifest.xml* — expresses meta information about the application that the Android system requires before any execution. It contains the following important information:

- **Package Name**
  Every application is uniquely identified with a package name. Only one instance a package name can be installed on the Android OS. The naming convention is the following[34]: *The name should be unique. The name may contain uppercase or lowercase letters ('A' through 'Z'), numbers, and underscores ('_'). However, individual package name parts may only start with letters.*
- **Version Name / Version Code**
  The version code identify the application version on the Google Play but what is actually displayed for the user is the version name.
- **Permissions**
  Permissions are shown to the user before the installation and it must accept them. It enables the application to access to specific API calls that often require to use peripheral devices of the phone (camera, micro, etc.), system applications (contacts, google services, etc.) or network/GSM capabilities. Misused or maliciously used, these API call can be harmful for the users.

  Prior to Android API 23 (a.k.a. Android 6.0 Marshmallow), all permissions were treated the same way. From API 23 until now permissions are classified into two categories: normal and dangerous permissions. Normal permissions are granted on installation and dangerous permissions are granted on usage — once granted, it is permanent. However, most of the users

34. https://developer.android.com/guide/topics/manifest/manifest-element.html#package

(83%) do not pay attention to the permission, and only a tiny fraction of users have a comprehension of what are the permission and the consequences to grant them (3%) as shown in this study [11]. This system definitely helps to frame application behaviors to the contrary of programs running on a regular computer, but it is a weak security protection as the user is responsible for it usage. However it is an important feature for classification as permissions put a frame on application behaviors.

- **Activity**
  An activity is a visible window of the application. Android emphasizes the design of applications by activities. When an activity is not visible anymore, is it put in a paused state. Applications declare a main activity that will be started when the application is lunched.
- **Intent Filters**
  Intents are signals sends to the Android system or to the application itself in order to execute a component. For instance, taking a photo or visiting a link can be done with an intent, requesting the capabilities of external applications. There are *explicit* and *implicit* intents. Explicit intents make the Android system call the specified activity or service directly. The target and source of the intent must a component of the same application. Implicit intents are used to call a component of another application as shown in Figure 3.4 (source [35]).

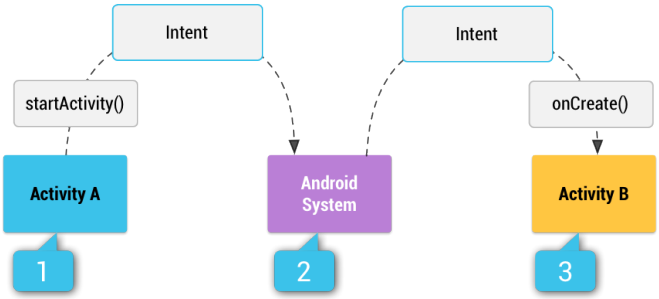


FIGURE 3.4 – How an implicit intent is delivered through the system to start another activity: (1) Activity A creates an Intent with an action description and passes it to *startActivity()*. (2) The Android System searches all apps for an intent filter that matches the intent. When a match is found, (3) the system starts the matching activity (Activity B) by invoking its *onCreate()* method and passing it the Intent.

The Android system receives the intent and target the first app component that have the corresponding *intent-filter*. An intent can be sent with arguments. It is why it is often the entry point of a vulnerability, as the intent parameters must be sanitized. Parameters are considered as unsafe values from the point of view of the targeted application.

- **Receivers**
  A receiver — short for BroadcastReceiver — is a component that is executed when a broadcast intent is received. Some receivers are symptomatic of malware behaviors like *BOOT_COMPLETED* which is triggered when the phone boots.
- **Services**
  A service is a component that can run tasks in the background — when the UI is not visible. It can be triggered by broadcast intents like a receiver.
- **Providers**
  A provider is a component that is able to provide data that is managed by this application to another application. It encapsulates the data and provides an interface to share it. It can come along with a custom permission that another application must declare to use it. Providers can also be triggered with a broadcast intent.

35. https://developer.android.com/guide/components/intents-filters

Like the DEX file, the manifest file is binary encoded into a format named *AXML* to optimize the size and the processing time.

Assets & Resources

Assets and resources are two way of managing external files. Resources are able to provide alternative form of the same file — for different languages, OS, screen sizes and orientations. Resources are indexed with a unique identifier, and their paths are checked at compile time. They are classically *XML* configuration files and images. Last, resources are packaged into one binary file in the ARSC format. It is a binary *XML* — strings are indexed to save space. Assets, to the contrary of resources, are not indexed and are used only at the execution. Files can be stored in a free directory hierarchy — which is not possible with resources.

Certificates

Android applications are numerically signed to ensure that it has been packaged by its developer. It copes with the cases of developer identity theft and application corruption — voluntary or not. The *MANIFEST.MF* and *CERT.SF* files list all package files with *SHA1* fingerprint by default. The *MANIFEST.MF* file also contains meta information about the *.apk* package as developer information, package version, main class to be run, etc. The last part that constitutes the certification system of Android application is the *CERT.RSA* file that contains the signed content of *CERT.SF* along with the certificate chain of public keys used to sign the content. If the content is modified, the signature in *CERT.RSA* will not match content of *CERT.SF*.

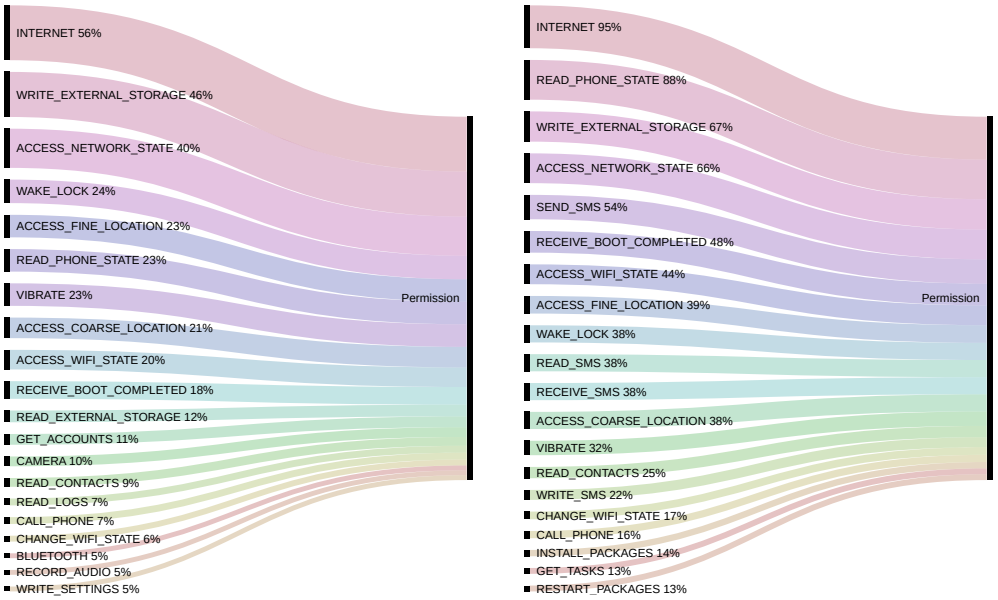
Knowing how applications are made in detail enables to reverse engineer them. In the next part, we see what are the state-of-the-art tools for reversing Android applications and for static analysis applied to the security domain — not to be confused with code review. Last, we present our contribution to the state-of-the-art.

3.2.2 Android Reverse Engineering and Static Analysis Features

An objective of this thesis is the detection of Android malware. What features enable a machine learning algorithm to differentiate benign and malicious applications? We give a review of the state of the art here, with motivations for our static feature choices. We distinguish two categories of static features: *raw* and *complex* features.

- *Raw features* are values extracted directly from the executable without any post-processing. As an instance, presence of permissions, strings or a sequence of opcodes are raw features. They are the most efficient features to process as it involves only to read the executable.
- *Complex features* values obtained after the processing of an abstract representation of the executable or by inference. For instance, features from *taint analysis*, a *control flow graph* (*CFG*), *call graph*. Features captured from these abstract representations are often high-level information as structural or behavioral information.

What static analysis features are effective for classification and for what cost? The processing cost of complex feature is taken into account because we want to design an efficient process. Studies use different classification algorithms and datasets to assess the effectiveness of their method. It is why, here, we are not able to give a definite answer to the question. We compare results mostly by their final *accuracy*, but its absolute value is not fully correlated with the feature choice — because of other algorithmic choices as previously stated. Instead we try to grasp a tendency for each feature to select the most promising ones. This tendency can be balanced with implementation constraints. For instance, we often rejected graph based complex features that have a high asymptotic complexity. Last, we test the selected features against a fixed classification algorithms and a fixed feature selection method. The chosen classification algorithm and feature selection method are detailed in the next chapters. Review of raw static features:



(a) Benign applications

(b) Malware applications

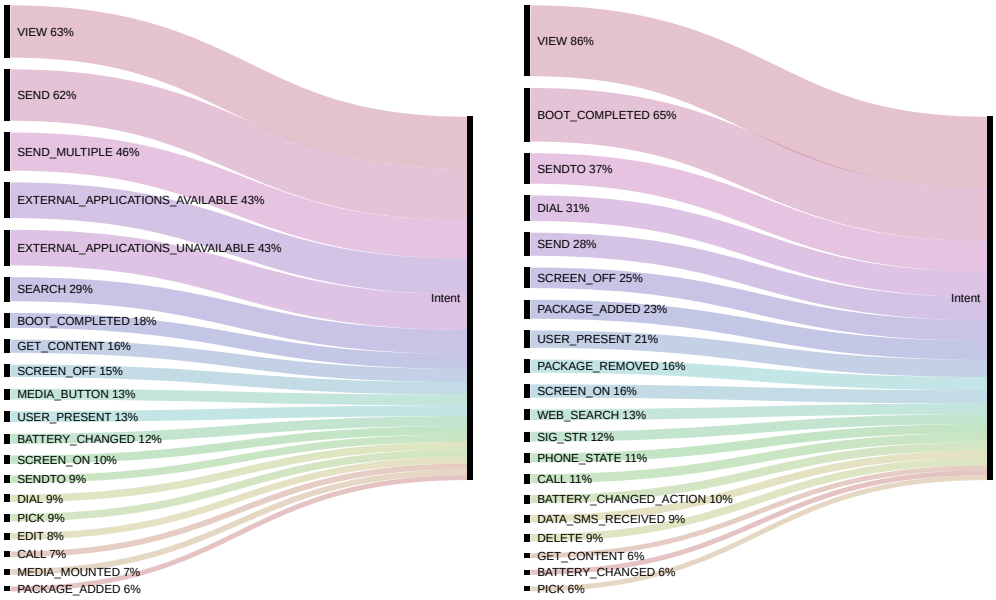
FIGURE 3.5 – Top 20 permissions

- **Presence of a permission** (type: Boolean). A permission request is a clear sign of a behavior in an action range defined by the Android system. Generally these behaviors are sensitive from a security point of view. Used in a malicious way, this can harm the user. This is why the user must validate these permissions. Malware sometimes exploits a vulnerability to gain additional rights, but most of the time, they simply ask for the permission to act.

  The list of requested permissions seems to be an important feature that allows to deduce a potentially malicious behavior. We list previous studies than only use permissions as classification feature. Leeds and al. [12] achieved a 80% accuracy averaged over 10 neural network training results. Feizollah and al. with *AndroDialysis* [13] achieved 83% accuracy with *leave-one-out cross validation* on a dataset of 7,406 applications (1,846 benign, 5,560 malicious) with a Bayesian network. Gunjan and al. [14] achieved a 81.32% accuracy [36] with a *J48 classifier*. Sanz and al. with *PUMA* [15] obtained 86.41% accuracy with a *random forest* classifier, evaluated with 10-fold cross validation on a dataset of 606 applications (357 benign, 249 malicious). 87.23% accuracy with a neural network on a 10-fold cross-validation dataset of 650 applications (325 benign, 325 malicious). Permissions appear to be effective to detect malware, but it is clearly not enough. We made statistics on our own dataset (5,585 malware samples, 2,206 benign samples) on Figure 3.5. It clearly shows that there is in fact great difference between the malware set and the benign set.

- **Presence of an intent** (type: Boolean). Intents are actions that an application asks the Android system to resolve. It can be the opening of a file, a web page or the sending of a block of data to another application. The system itself sends broadcast messages during special events. Applications can then declare components that react with these intents. It is the basis of inter-application communication. The components that react to intents are declared in the manifest. They can also be dynamically declared in the source code. We will also find the sending of intents in source code. Often malware abuses this system to perpetuate the execution of their malicious routines. With intents, malware can run as soon as the phone is turned on or when the screen is off. Studies that use only intents as classification feature achieve honorable accuracy. [13] achieved 91% accuracy with *leave-one-out cross validation*

36. We could not find information about a test set or cross validation and there is no dataset details which is highly dubious. However, the results are coherent with the state-of-the-art.



(a) Benign applications

(b) Malware applications

FIGURE 3.6 – Top 20 intents

on a dataset of 7,406 applications (1,846 benign, 5,560 malicious) with a Bayesian network. This is the only study that only uses intents as features. It is to be noted that Drebin [2] use intents as a part of their feature set with SVM on a split dataset (117,893 benign samples, 5,560 malware samples; 66% learning, 33% testing). On our own dataset, the statistics of intent usage or declaration is shown in Figure 3.6. There are significant discrepancies between those distributions, making intents a good feature candidate.

- **Presence of a certificate signature** (type: Boolean). Only [16] use certificate signatures in their detection scheme. They show that, among their dataset of 4,554 malware samples, 70% of them use 24 unique signatures. This feature is a good candidate for creating a first filter that act as a blacklist.
- **Opcode N-Gram frequency** (type: Numeric). This method enables to identify malware with their code structure. Any byte code operation is constituted by an opcode (the operator code) and one or more operands (operator values). Values can vary arbitrarily from family sample to another due to string change and obfuscation. However it is difficult to change the structure of operators that constitutes the core logic. According to the literature, features based on the opcode sequence are very effective for both classification and detection problems. A common strategy to process the opcode sequence is to sum its n-gram occurrences. A n-gram is a sliding window of size n over the sequence. Shabtai et al. [17] obtained 96.88% using to top 1,000 opcode bigrams on the Malgenome dataset [18]. BooJoong Kang et al. [19] reported a 98% F-measure on a dataset of 2,520 with a SVM classifier. Niall McLaughlin et al. [20] obtained the accuracy of 98% with a *CNN* (Convolutionnal Neural Network) on the raw opcode sequence and 98% with bigrams and trigrams. The state-of-the-art clearly shows that opcode sequences produce potent features for classification.
- **Android API call frequency** (type: Numeric). The Android API is often the only entry point for accessing certain phone features. Information that identifies the phone or the user, access to SMS, calls, contacts, etc. Using these calls to the Android API in an unusual way may reveal mischievous behaviors. Researchers have already studied the usage of these API call sequences to the malware detection problem. DroidAPIminer [21] achieve 99% accuracy with KNN [22], on a dataset of 3,987 malware samples and 500 benign samples with split testing (66% learning, 33% testing). Feng Shen et al. [23] achieve 94.5% accuracy with a SVM

classifier on a dataset of 3,899 malware samples, 3,899 benign samples with split validation (90% learning, 10% testing). It appears that API calls are also a feature with potent discriminative power.

- **File / code sections size** (type: Numeric). The goal of malware is to spread quickly. This is facilitated by a small file size, a reduced code that does not clutter useless libraries, etc. Statistical studies conducted by Ludovic Apvrille et al. [24] show that 30% of malware samples has a size lesser than 70000 bytes as opposed to 21% for benign samples. Ludovic Apvrille et al. [25] use the file size as part of their feature vector, with a combination of classifiers they obtain more than 99% accuracy with split testing. However it is hard to know the impact of the file size on the overall detection. File and sections sizes are probably a weak indicator, but is a feature easily collected.
- **Whole source code as a byte sequence** (type: N/A). The binary DEX file is used as a whole in the case of Deep Learning. A Deep Learning algorithm uses raw input data and learns which features are predominant. McLaughlin et al. [20] use the raw opcode sequence with a convolutional neural network, the results range from 80% to 98% accuracy depending of the size of the dataset. The disadvantage of this technique is its weakness to obfuscation. A simple padding of the DEX file or a manipulation of the DEX so that the malicious code is found at the end of the file and the detection is lapsed. Indeed, Deep Learning algorithms cannot take arbitrary sizes of input objects, they need fixed sizes. So in general, the DEX file is truncated if it is too big or padded if it is too small. Another drawback is the learning and execution time, that constraints the use cases.

Review of complex static features:

- **Presence of a behavior defined by a rule** (type: Boolean). Axelle Apvrille et al. [26] use the presence of predefined behaviors as part of their feature vector, with a combination of classifiers they obtain more than 99% accuracy with split testing. However it is hard to know the impact of the presence of predefined behaviors on the overall detection. We have experimented the use of static rules for detecting risky behavior. These rules were designed manually with the observation of malware source code. The list of detected behaviors is as follows:
    - Reading SMS.
    - Sending SMS.
    - Phone calls.
    - Application installation (social engineering).
    - Application uninstallation (social engineering).
    - Execution of a shell command (more specifically chown / chmod / mount / remount).
    - Attempt to escalate privileges.
    - Dynamic code loading.
    - Using Java reflection.
    - HTTP requests.
    - File system parsing.
    - Using native code.
    - Track the position of the phone.
    - Access to Android account information.
    - Retrieving the number of this phone.
    - Using a resource.
    - Execute code when the phone starts.

TABLE 3.3 – Rate of benign and malware samples exhibiting risky behaviors

Feature	Benign samples rate	Malware samples rate
AdvancedPHONEFeatures	1,1%	1.9%
APNSettings	0,6%	26.9%
AskForAPKUninstall	1,5%	12.6%
ASEExecute	0,9%	1.7%
ChmodString	4,8%	51.3%
DynamicCodeLoading	35,4%	12.8%
DynamicCodeLoadingFromDEX	0,2%	2.9%
ForceAPKInstall	3,4%	77.3%
GetACCOUNTS	1,4%	1.5%
GetFromAssets	31,3%	77.5%
GetFromResRaw	17,7%	39.7%
GetPhoneNumber	1,6%	72.1%
HTTPRequests	44,1%	96.6%
LocationService	10,2%	40.1%
MailSender	5,4%	29.6%
MountString	2,2%	2.1%
NativeCode	12,1%	50.8%
PhoneCall	2,4%	12.8%
Reflection	38,0%	29.8%
RunatBOOT	7,0%	71.6%
SMSReceiver	5,9%	57.4%
SMSSender	3,1%	33.4%
SMSSenderbyURI	4,5%	57.4%
ShellExecution	17,5%	81.9%
SuString	9,7%	68.5%
ShString	9,5%	39.1%

- Sending mails.
- Modification of wifi settings.
- Access to advanced telephony features.

These behaviors, when found, do not mean that an application is malicious. Some of these behaviors clearly indicate that it is a malware, others indicate a potential malignancy. A statistical study on a base of 600 malware samples and 600 benign samples gives an idea of the nuance these characteristics bring (Table 3.3).

We can see that certain behaviors are frequent in malware and infrequent in benign applications like *SuString*, *ShString*, *Runatboot*, *APNSettings*, *ChmodString*, *NativeCode*. Other criteria are rarer but when present, indicate a high probability of malware like *DynamicCodeLoadingFromDEX* and *AskForAPKUninstall*.

- **Dependency graph**. A dependency graph is an oriented graph representing the object dependency contained in the application. The nature of the objects studied can be diverse, so there are many different dependency graphs. In the article *Profiling user-trigger dependence for Android malware detection*, Elish et al. [27] study the relation between user entry points — UI actionnable items — and the Android API calls that are triggered. This study relies on the observation that malicious applications run malicious code without user intervention, whereas benign applications typically use Android API calls when the user does a specific action. The dependency graph starts with the entry points of a user action — ex: *onClickListener* — and follows calls in the form of a tree until all calls to the Android API from these entry points are counted. Some metrics are processed from the graph. Simple thresholds on these metrics enable the classification of applications as malware or benign software. The final accuracy is around 98% but it is unsure that proper statistical tests have been made (no mention of split or cross testing). In the article *Structural detection of android malware using embedded call graphs*, Gascon et al. [28] use call graphs to detect Android malware. Only Android API calls are captured to reduce the distinct subgraphs. These graphs are then indexed to constitute a feature vector. To analyze a new sample, for each of its subgraphs

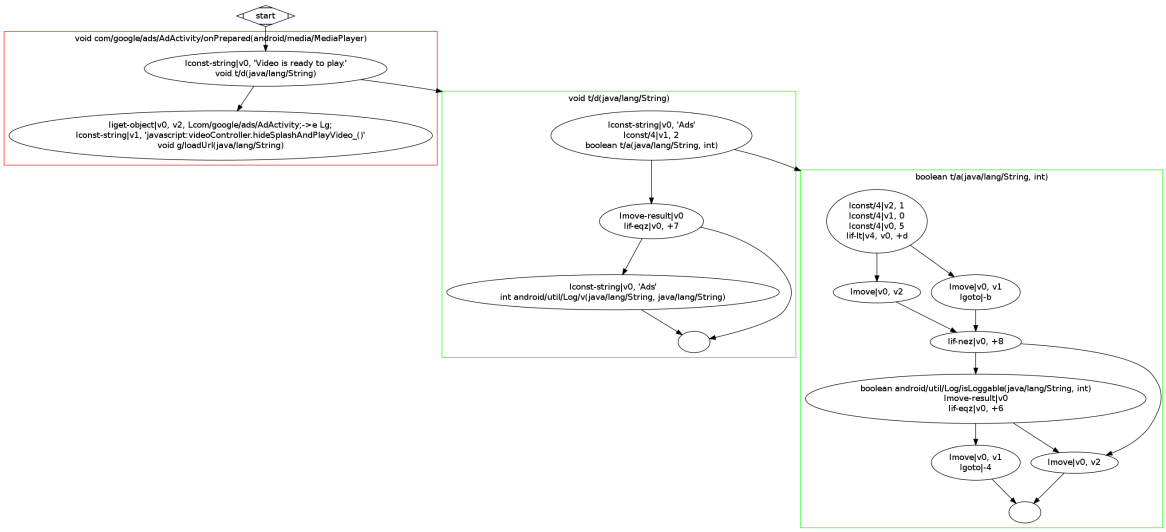


FIGURE 3.7 – Egide Control Flow Graph

the most similar is queried from the subgraph database. It sets then the corresponding index in the feature vector 1 to indicate the presence of this subgraph. Then the feature vector is used on a anomaly detector to single out malware samples from benign samples. The authors do not precise which anomaly detector is used. They obtain 93% accuracy on malware samples with 2% false negatives and 5.15% false positives on a test dataset of 2,200 malware samples and 13,500 benign samples. Dependency graphs seem to be a powerful tool for detecting malicious applications. However, there is still no consensus on the best way to use these graphs, what they should contain, as well as how to extract data suitable for Machine Learning. Added to this a relatively long graph generation and processing time compared to the other features mentioned above, it is a feature that we finally rejected for these different reasons after a preliminary study. *Egide* [29], a software that we developed prior to this thesis, allows to extract control flow graphs starting from every entry points. Initially planned for manual analysis of Android applications, we realized their inapplicability as an analysis support as well as the complexity to use them as a feature for a machine learning algorithm. Noting that the final performance of these dependency graphs are good but far from exceptional compared to other features, we chose efficiency and rejected its use. In Figure 3.7 we show an instance of a control flow graph from an application analyzed with *Egide*.

Existing tools for reversing Android application have different objectives. None of them have been designed for feature extraction, which is a crucial step for malware detection. The main tools to reverse apk are *baksmali*[37], *dexdump* (tool avalaible in the Android SDK), *Androguard*[38], *radare*[39] and *Dedexer*[40].

- *Baksmali* is a disassembler for the DEX format. It transforms the whole source code into *smali* format which is a human readable version of the Dalvik Bytecode. The reverse is done on the hard drive and there is no API.
- *Dexdump* is a tool available in the *AOSP*. It is a frontend program on the *libdex* library which is used by the Android OS to process applications. It dumps the DEX file on the terminal with a human readable format. There is no API to control the program and it cannot be used for targeted reverse.
- *Androguard* is a tool developed in the *CVO* lab. at *ESIEA* that allows to reverse and analyze applications. It is a complete tool that offers an API. It contains many functions for manual

37. https://github.com/JesusFreke/smali
38. https://github.com/androguard/androguard
39. https://github.com/radare/radare2
40. https://sourceforge.net/projects/dedexer/files/dedexer/

code analysis, allows the creation of malware signatures, and reverse engineering. However it cannot be used for targeted reverse, but the whole process can be done in memory. Last, it transforms the code into *smali*.

- *Radare* is a set of programs for the analysis and reverse of binaries. *Radare* is able to process Android applications. It can reverse the DEX file and the Manifest. Moreover it can be used for targeted reverse of source code parts. It has numerous of functionalities for static analysis like control flow graph. However there is no prorammable API and the reverse is done on the hard drive.
- *Dedexer* can reverse the DEX file. However it is not suitable for our needs as there is no API, no targeted reverse and the reverse is output on the hard drive. Finally, as there is no tool that suits all our needs so we developed our own. This tool is part of the *Libmla* library and can reverse the whole APK in memory, make targeted reverse and extract exactly the features we need for Machine Learning.

3.2.3 Our Contribution: libmla-android-reverse

We realized that the existing tools would not allow us to build efficient technologies in a production context. *Androguard* is the fastest tool we have tested — cf. *Cygea* [30] — and it turned out to be the bottleneck of the project. That is why we chose to develop our own library of reverse engineering. We apply here a strategy adapted to our problem. Rather than designing a generalist tool which reverses and analyzes the applications, we are focused on extracting only the features we need. This constraint allow the development of an efficient tool, because only useful information is extracted. We use the *libdex* library (version 7.1 of Android), distributed with the AOSP. This library is used to read DEX files and that is what the Android *dexdump* tool uses. No documentation is provided with AOSP or available elsewhere, and no public project do use *libdex* to our knowledge. We extracted this library and made corrections for recompiling it on the *libmla* project. The code has been analyzed to understand how to master it in standalone mode. To process the Manifest file that is encoded, we were inspired by the *AXML* parser written in *C++* of the project *Jitana*[41]. A final element that we thought was essential is the certificate that allows in certain cases to identify the author of the application. For this we used the *OpenSSL c* library. The overall architecture of the reverse engineering module is detailed in Figure 3.8.

A large part of the application source code is composed of external Java libraries. These libraries are, for the vast majority, benign code. Including it in the analysis is a waste of time and a dilution of the characteristics of malicious applications. That is why we have a list of common package names. We use it as a filter in the analysis. This method is effective for both the analysis time and the quality of the malware detector, but it will have to be improved for production. Indeed, malicious software can spoof the package name of a common librarie and thus bypass the detection. This is why we need to store into a database, fingerprints of codes from common libraries. This is a substantial work because it is necessary to follow the updates of libraries and to be able to accept small modifications of the code. In fact, according to the machine and the version of the software which compiled the application, which can lead to small differences in the code obtained during the reverse engineering. From an application we extract:

- The number of activities, receivers, providers, services and permissions.
- Declared permissions, static and dynamic intent-filters, intents used in the source code.
- Bigrams of strings defined in the source code. Using trigrams is impractical because in our dataset we get over a million unique trigrams, which borders the limits of the structures that manage features.
- Trigrams of method opcodes. We observed that using trigrams of method opcode regardless of basic blocks is more effective than using trigrams from basic block opcodes (cf. last chapter).

41. https://github.com/ytsutano/jitana

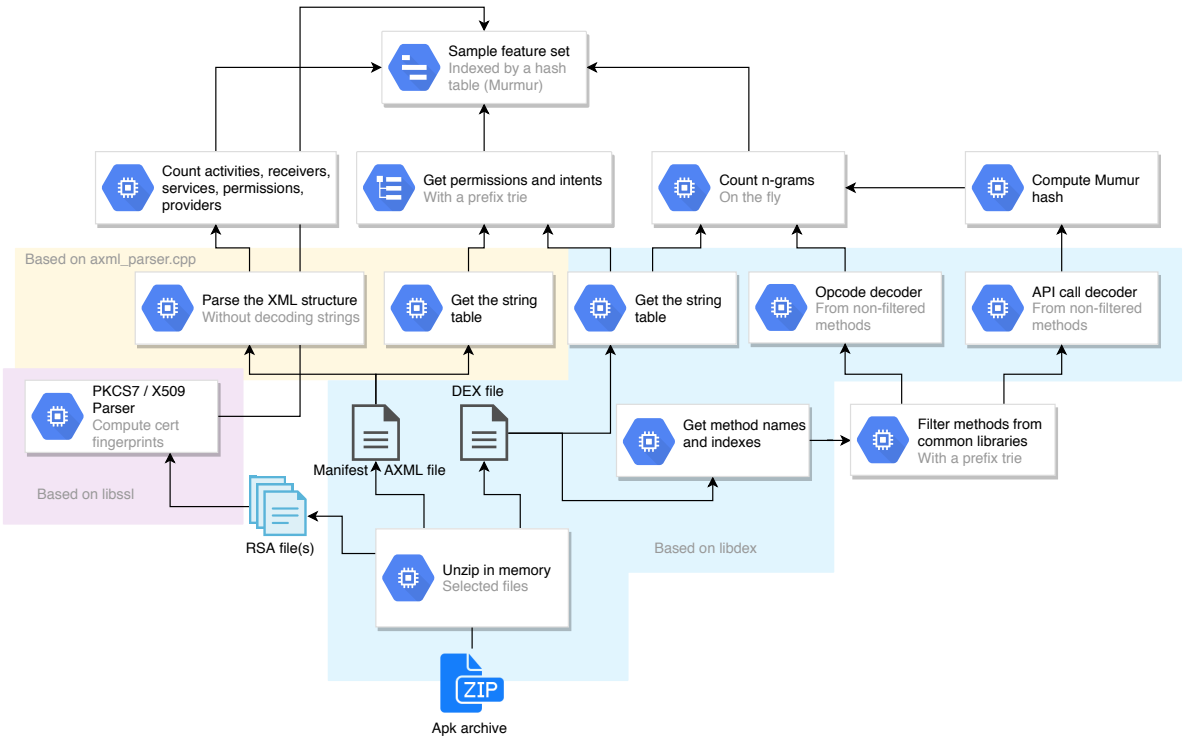


FIGURE 3.8 – Surgical extraction of features in memory

- Trigrams of common library calls. Here, we do not restrict calls to the Android API, but calls from any library that has been filtered. Otherwise parts of the code logic would not be taken into account. Its effectiveness is also backed by the experimentation (cf. last chapter).
- SHA Fingerprints of all certificates declared in the *META-INF* directory.

On a regular computer with an *Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz* CPU, it takes $2m$:$59s$:$526ms$ to extract the features of 7,791 applications. On average the reverse and feature extraction process take $23ms$. For our use case there is no competition with the available tools.

3.3 Dynamic Analysis

3.3.1 Application Execution

We developed a dynamic analysis tool that instruments the Android OS. Therefore having an in-depth understanding of the Android OS and how it executes the application is required. The first time an application is launched, it is translated into an optimized format depending on the CPU architecture. Prior to Android v4.4 the runtime system was the ***D**alvik **V**virtual **M**achine* (*DVM*). This system has then been replaced with the ***A**ndroid **R**un**T**ime* (*ART*). These two systems have different ways of optimizing the application as shown in Figure 3.9 (source [42]).

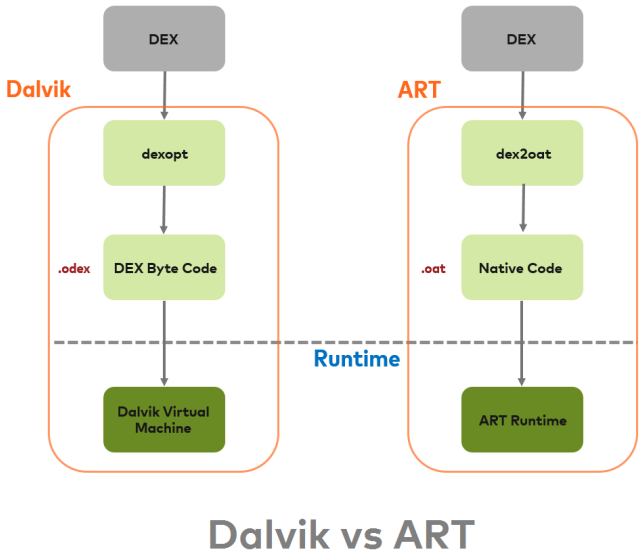


FIGURE 3.9 – Application pre-execution *DVM* vs *ART*

The *DVM* system performs bytecode verification, deducing code paths, mapping registers and deducing the Garbage Collector. An ***O**ptimized **DEX*** (*ODEX*) header is added to the code. It optimizes the data and dependency tables. Last, it changes some instruction with faster versions that eliminate table lookup. These instructions are unsafe to use outside of the frame of *ODEX* optimizations. The author of *Abusing Dalvik Beyond Recognition* [43] shows how to bypass the *ODEX* optimization routine and how to use these fast bytecode version to dynamically execute arbitrary code, hide the real code from analysts and break common reverse-engineering tools. The *ODEX* code is interpreted. Some *hot* portions of the code are compiled to native bytecode and/or inlined by the ***J**ust **I**n **T**ime* (*JIT*) compiler. Hot portions of the code are methods that are often used.

42. https://android.jlelse.eu/closer-look-at-android-runtime-dvm-vs-art-1dc5240c3924
43. http://archive.hack.lu/2013/AbusingDalvikBeyondRecognition.pdf

Hence, the more an application is running, the more it will be optimized — in theory. The *ART* system enhances the performance of application execution by compiling them to a native binary, in *OAT* format. This routine is called ***A**head **O**f **T**ime* (*AOT* compilation. An *OAT* file is an *ELF* shared object file with sections containing *OAT* data:

- *oatdata*, that contains the *OAT* header and the original *DEX* file.
- *oatexec*, that contains the native code of the compiled method.
- *oatlastword*, a special symbol that marks the end of the native code section.

A detailed explanation of the *OAT* format can be found in [31]. Their is no in-depth documentation on *ART* system, but its functioning is essential for the rest of our study. Therefore, we analyzed the ***A**ndroid **O**pen **S**ource **C**ode* (*AOSP*) version 5.1 to understand exactly how *ART* executes applications. *ART* exhibits three behaviors that are important for our study:

- The first time Android is launched, Java Android API libraries and applications are optimized and compiled to OAT.
- Each time a new application is installed, it is optimized and compiled to OAT format.
- Java methods can be executed in three ways: by an OAT JUMP instruction to the method address, by the ART interpreter for non-compiled methods (debugging purposes mostly), or via the Binder for invoking a method from another process or with Java Reflection. Details on the Android Binder can be found in [32] chapter 4.

Last note on application execution, almost every applications use calls to the Android API. It would be costly to load an instance of Android API with every applications. This is a reason why all applications are started with a fork of the same process, named *Zygote*. *Zygote* loads the Android API in memory and replaces every calls to the Android API with their actual address in memory.

3.3.2 Android Application Dynamic Analysis and Features

Dynamic analysis systems started being designed since 2010 [33] by the academic research, to circumvent the limitations of static analysis — namely, code morphism and obfuscation. Since that time, many systems have been released. For this study we have built a classification of a part of these systems, presented in Table 3.4. The classification takes into account three categories: the dynamic features collected by the analysis, the strategies set in order to automate application testing and finally the use of real devices in dynamic analysis systems history. We discuss the results on the following sub-sections.

Features Analyzed

Since the rise of *Android* dynamic analysis systems, the use of system calls have been the leading approach. System calls are the functions of the kernel space, available to the user space. It gives the capacity to manipulate hard drive files or to control processes. System calls can describe a program behaviors, from a low-level perspective. The collection of those calls can be achieved in two ways, mainly:

- **Virtual Machine Introspection** — This is a technique available for emulators, which enables the host to monitor the guest. It cannot be detected by the guest since it is out of its reach and it is therefore convenient for security analysis. *Andrubis* [1], *CopperDroid* [40] and *DroidScope* [36] take advantage of VMI to retrieve, unseen by the target malware, all systems calls done by the guest *Android* virtual machine.
- **Strace/ptrace** — *Strace* is a Linux utility for debugging processes. It can monitor system calls, signal deliveries and changes of process state. *Strace* use the *ptrace* system call to monitor another process memory and registers. This second method is by far the simplest and the most straightforward one as the only task here is the automation of the *strace* execution. Moreover, it targets the system calls of the application directly. That is why this method has

TABLE 3.4 – Comparative state of the art of dynamic analysis systems

Reference	Tool name	Dynamic features used	App testing strategies	Objectives & comments
Thomas Bläsing et al. 2010 [33]	AASandbox	System calls (name)	Monkey	Data for malware/benign classification # Virtual device
Iker Burguera et al. 2011 [34]	Crowdroid	System calls (name)	Crowdsourced app interactions	Data for malware/benign classification # Real device
Cong Zheng et al. 2012 [35]	SmartDroid	Taint tracking, +?	UI brute force Restriction of execution to targeted activities	Data for classification or manual analysis # Virtual device
Lok Kwong Yan et al. 2012 [36]	DroidScope	System call (all content) Java calls (all content) Taint tracking	-	Data for classification or manual analysis # Virtual device
Vaibhav Rastogi et al. 2013 [37]	AppsPlayground	Taint tracking Targeted Android API Java calls	Monkey UI brute force Broadcast events Text fields filling	Malware/benign classification # Virtual device
Martina Lindorfer et al. 2014 [1]	Andrubis	App Java calls (all content) System calls (name, +?) Shared libraries targeted calls (name, +?) Taint tracking DNS/HTTP/FTP/SMTP/IRC (all content)	Monkey Broadcast events All possible app services All possible app activities	Data for classification or manual analysis # Virtual device
Mingyuan Xia et al. 2015 [38]	AppAudit	Taint tracking	~	Malware/benign classification Data leak detector # Symbolic execution
Vitor Monte Afonso et al. 2014 [39]	-	Targeted *Android* API Java calls (name) System calls (name)	Monkey Broadcast events	Malware/benign classification 96.66% accuracy # Virtual device
Kimberly Tam et al. 2015 [40]	CopperDroid	System calls (all content) Binder data	Broadcast events Text fields filling, +?	Data for classification or manual analysis # Virtual device
Gerardo Canfora et al. 2015 [41]	-	System call (name)	Monkey	Malware/benign classification 94.9% accuracy (unseen applications) # Virtual device
Marko Dimjašević et al. 2016 [42]	Maline	System call (name)	Monkey Broadcast events	Malware/benign classification 96% accuracy # Virtual device
Michelle Y. Wong et al. 2016 [43]	IntelliDroid	Taint tracking	Targeted inputs leading to suspicious Android API calls	Data for classification or manual analysis # Virtual device
Gerardo Canfora et al. 2016 [44]	-	Measures of resource consumption (CPU, Network, Memory, Storage I/O)	Monkey	Malware/benign classification 99.52% accuracy # Real device
This work	Glassbox	Java calls (name) System calls (name) HTTP/HTTPS requests (all content)	Monkey UI brute force Broadcast events Real SMS/Call Text fields filling	Data for malware/benign classification # Real device

TABLE 3.5 – Legend

+?	The paper is not clear enough on those details and we cannot be sure that it is an exhaustive list
call (name)	Only the name of the call is used, in order to get the appearance frequency
#	Comment
-	No data
~	Data exists but is irrelevant for this study

been adopted in most of the literature, namely *Crowdroid* [34], *Maline* [42], [41] and [39]. We have also chosen to use the *strace* utility for system calls monitoring. It is theorically possible that an application with root privileges tries to detect that its process is beeing debugged by *strace*. By calling *strace* on its own processes and issuing an error, the malware application can deduce it is under debug. Despite of this theoretical possibility, we found malware application currently able to do so.

System calls seem to give great results for classification. *Maline* reported 96% accuracy rate, and [41] reported 94.9% accuracy rate on unseen applications with syscalls frequencies only. Actually syscalls capture low level behaviors of both Java code and native code.

The second most collected feature is taint tracking information as it reveals data leakage. It works by the instrumentation of the *Dalvik VM* interpreter. The information we do not want to leak is called a *source*. Some *sources* of personal data are tainted, like the phone number or the contacts list. Each time a tainted *source* or value is used in a method call, the *DVM* interpreter taints the returned value. With this simple mechanism, we can observe the propagation of the tainted information regardless of its transformations. A function that enables to transmit information outside of the system is called a *sink*, like network requests or SMS. If a tainted value is used in a *sink*, it means data *source* has leaked. It enables to detect data leakage even if this data have been ciphered or encoded. An application that leaks data is not necessarily a malware, as data leakage is the business of both malware and user tracking frameworks in commercial applications — which constitutes essentially a large part of goodware applications. Whereas this feature gives useful insights on the application behaviors for manual analysis, its utility for automatic malware detection needs to be proved. Moreover the implementation and execution of taint tracking are costly, which leads us not to choose this feature for now. Java calls is another feature of interest as it captures an explicit behavior of the application. There are several ways to collect them:

- **Application instrumentation** — This strategy does not need any modification of the *Android* source code and is not dependent on *Android* version. The application can be modified in order to dump targeted method parameters and return values. *APImonitor* [45] is a tool that enables the instrumentation of targeted Java calls. It reverses the application into *smali*, a human friendly format equivalent to the Java bytecode, with the *baksmali* [46] utility. Then, it adds monitor routines around the targeted calls and it recompiles the code with the *smali* [46] utility. This strategy is used by the authors of [39].
- **DVM/ART instrumentation** — The *DVM* (*Dalvik Virtual Machine*) or *ART* (*Android RunTime*, *Android* API version $\geq$ 4.4) is the system that interprets and executes all the application instructions. All Java calls converge to this component. Hence, by hooking the execution of *DVM/ART*, one can monitor and control all Java calls, their arguments and their return values. That implies the modification of *Android* source code and its compilation to a custom *ROM*. This is the strategy we chose to use for collecting Java calls. We prefer this method for keeping the application behaviors pristine, and particularly not inducing additional bugs. *Andrubis* [1] and *DroidScope* [36] use similar approaches for tracing method calls.

For the last features, they are highly marginal. Here, *Andrubis* reported the retrieval of targeted shared library calls. Another data are the network communications. Only *Andrubis* reported the utilization of features from network communications, but without any further details. Our system makes use of *Panoptes* [47] for gathering plain text and encrypted web communications. A recent study, [44], shows that the measures of resource consumption for malware detection is a promising trail. It reports 99.52% accuracy with global measures of CPU usage, Network usage, RAM usage and Storage I/O usage.

Automated Testing Strategies

Dynamic analysis does not consist of launching the application and waiting the malware to show its malicious behaviors off. Malware are using logic bombs for hiding the payload. Logic bombs are a malicious piece of code that is executed after a condition is triggered. It means we

need to test each application as a real user could have done it. For achieving this objective, several strategies have been used in the past:

- **Black Box testing strategies** — This class of strategies does not take the application source code into account, it focuses on sending inputs in the application without any prior information. This is the commonly used strategy. *Monkey*[44] is a dedicated tool created by *Google* for this task. It generates random events in a fast pace. Events range from system events (home/wifi/bluetooth/sound volume etc.) to navigation events (motion, click). Because of its capacity to quickly explore applications activities, it has been used by most of dynamic analysis systems (*AASandbox* [33], *AppsPlayground* [37], *Andrubis* [1], [39], [41], [44], *Maline* [42]). *Monkey* is sometimes confused with *Monkey Runner*[45] in the literature, which is a python library for writing *Android* test routines.
- **White Box testing strategies** — This class of strategies takes the application source code into account. It focuses on sending specific inputs in the application for triggering targeted code paths. It requires the information from the static analysis of the application. Parsing the code is needed, to find the target methods and all their triggering conditions. *SmartDroid* [35] and *IntelliDroid* [43] determine all paths to sensitive API calls, then execute one of the paths to the target with dynamic analysis. Another kind of *White Box* testing strategy is *symbolic execution* where dynamic analysis is done by simulating the execution of the application static code. *AppAudit* [38] uses this technique for finding data leaks with symbolic taint tracking.
- **Grey Box testing strategies** — This class of strategies partially takes the applications source code into account. It focuses on testing all visible inputs the application declares or displays (UI). It usually takes the output of the application to generate the next inputs. *Andrubis* uses a *Grey Box* strategy when it tests all possible application services and activities, because they get the information from the application manifest. *AppsPlayground* also uses *Grey Box* testing with its *Intelligent Execution* where windows, widgets, and objects are uniquely identified to know when an object has already been explored. *Grey Box* testing have the advantages of using tests that applications are susceptible to respond, contrary to *Black Box* testing, without the requirement of processing a Control Flow Graph, as *White Box* testing. It is then the most efficient way of testing applications.

Real Devices

The use of real devices for dynamic analysis started with *Crowdroid* [34], a crowdsourced based analysis. Whereas this approach gives good results, one cannot ask users to execute real malware on their personal device. So this system can only be an option, when we have already a trained machine learning algorithm, to find malware in the wild.

BareDroid [48] is a system which manages real devices in large scale for dynamic analysis. Whereas *BareDroid* cannot be considered as a dynamic analysis system, because it does not analyze applications, it brought two major results for our study. First, real devices for dynamic analysis systems are a scalable solution financially and in execution time compared to virtual devices. Second, using real devices drastically improves features detected for malware families that often rely on emulator evasion like *Android.HeHe*, *Android Pincer*, and *OBAD*.

3.3.3 Our Contribution: Glassbox & Smart Monkey

Google reacted to the rise of malware with a dynamic analysis platform, named *Bouncer* [49], that analyzes applications before their release on Google Play. This security model is centralized and acts before the distribution of applications. Whereas this system suffers from limitations — like virtual device evasion — it has helped to reduce the malware invasion by 40% [49].

44. https://developer.android.com/studio/test/monkey.html
45. https://developer.android.com/studio/test/monkeyrunner/index.html

Android antivirus companies use another centralized security model which acts after the distribution of applications. Because applications have access to restricted resources and permissions, antivirus programs cannot perform their analysis — as it often requires root permissions and extensive resources. Hence, the static analysis is externalized onto the company servers. As a result, it can give quick responses — each application being analyzed just once. This is a shift of security model for the common user toward centralization.

Users have been used to decentralized security model for their personal computer, i.e. antivirus programs. This security model does not allow much room for maneuvre because any antivirus needs to be quick enough for not bothering the user — otherwise another quicker antivirus will be chosen. Whereas antiviruses have implemented heuristic algorithms, they are rather limited by the security model. Hence, the security model shifting is an opportunity for building more complex systems that require more resources to run. It enables security systems to use advanced research techniques like behavioral detection with dynamic analysis, or detection with features similarity from static analysis.

Malware authors made their strategy evolve with the rise of *Bouncer* and other dynamic analysis systems. They have started to hide the payload execution with emulation detection or/and the requirement of a user interaction. For example, the reverse of the sample [50] shows that malware applications are currently using emulation evasion. Emulator settings can be modified to mimic the appearance of a real device but there are a wide number of ways to detect *Android* emulation. Actually the tool *Morpheus* [51] proves us this war is already lost as the authors found around 10 000 heuristics to detect *Android* emulation. So trying to modify the emulator to look real is probably a waste of time. In such condition, we need to redefine the problematic.

This is why we are presenting *Glassbox*, a dynamic analysis platform for *Android* malware applications on real devices. *Glassbox* is an environment for the controlled execution of applications, where the *Android* OS and the network are monitored and have the capacity to block some actions of the analyzed application. This environment is paired with a program that automates the installation, the testing of applications and the cleaning of the environment afterwards. We called this program *Smart Monkey*, as a reference to *Monkey*, the *Android* tool for generating pseudo-random UI event. The objective of *Glassbox* is to collect features for machine learning algorithm, to classify applications as malware or as benign. *Glassbox* (Figure 3.10) is a modular system distributed among one or several phones and a computer. Glassbox has been published in 2017 at ForSE [52]. Each part is detailed in the following sections.

Android Instrumentation

A custom *Android* OS has been made, based on the *Android Open Source Project (AOSP)*[46]. The objective here is to log dynamically each Java call of a targeted application. This involves hooking these calls, at a point where all of them pass through. We instrumented *ART (Android RunTime)*[47], the *Android* managed runtime system that executes application instructions. A straightforward way of hooking Java calls is to instrument the *ART* interpreter. Unfortunately only a few calls are executed through it because most of the code is compiled into *OAT* and therefore it is not interpreted. We forced all calls to be interpreted by disabling several optimizations. The first one is the disabling of the compilation to *OAT*. That leads calls to be interpreted before being executed. But other optimization mechanisms come into play, namely *direct branching* and *inlining*.

The boot classpath contains the *Android* framework (Figure 3.11) and core libraries. They are always compiled in *OAT* resulting in a *boot.oat* file. This file is mapped into memory by the *Zygote* process, started at the initialization of *Android*. For launching an application, the main activity is given at *Zygote* in parameters. When *Zygote* is called that way, it forks and starts the given activity. It means that any application has access to the same instance of the *Android* framework and core libraries. *Direct branching* is an optimization that replaces framework/core method calls by their actual address in memory. So the calls do not pass through the interpreter. That optimization is

46. https://source.android.com/
47. https://source.android.com/devices/tech/dalvik/index.html

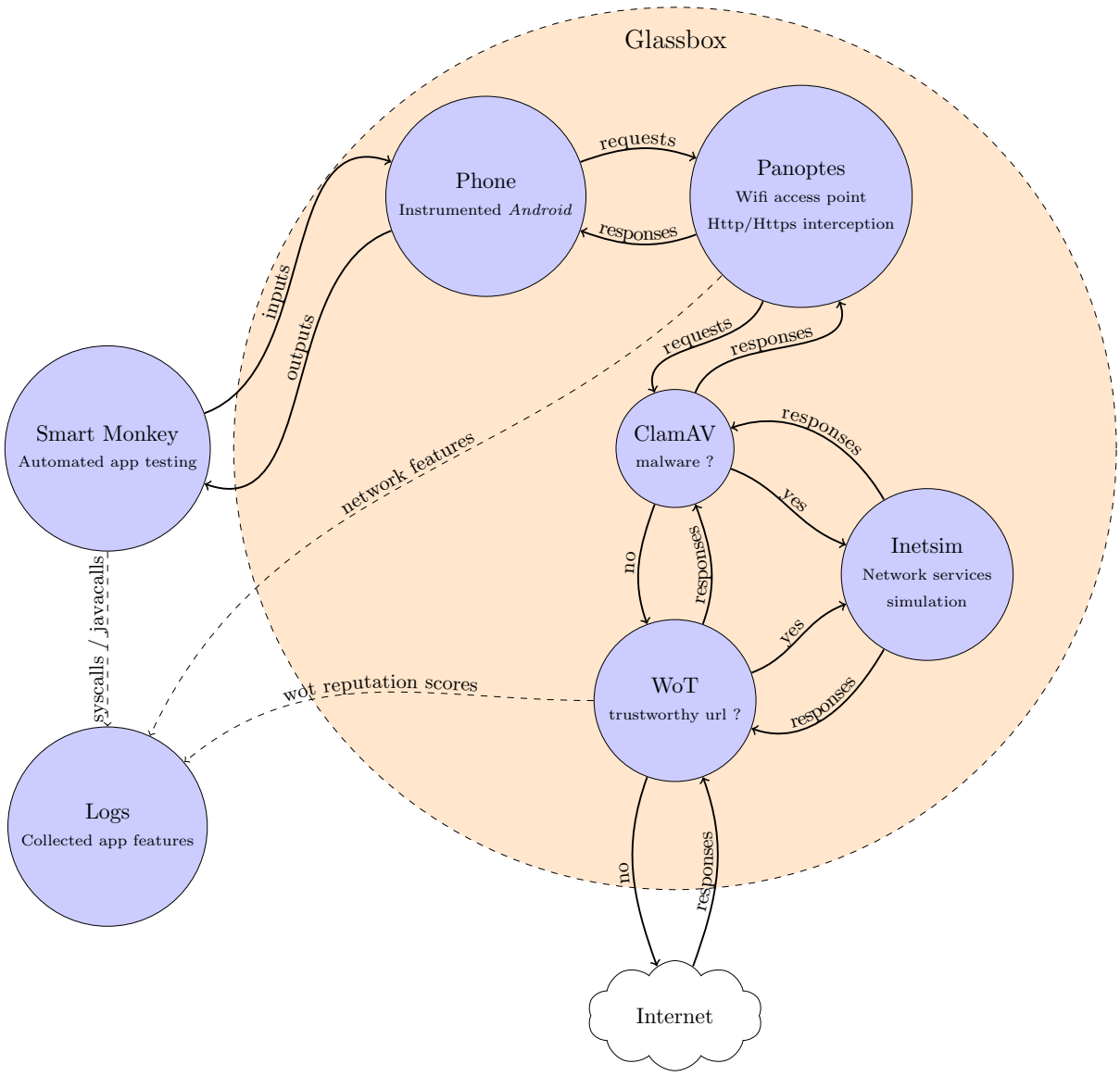


FIGURE 3.10 – Glassbox — Architecture overview.

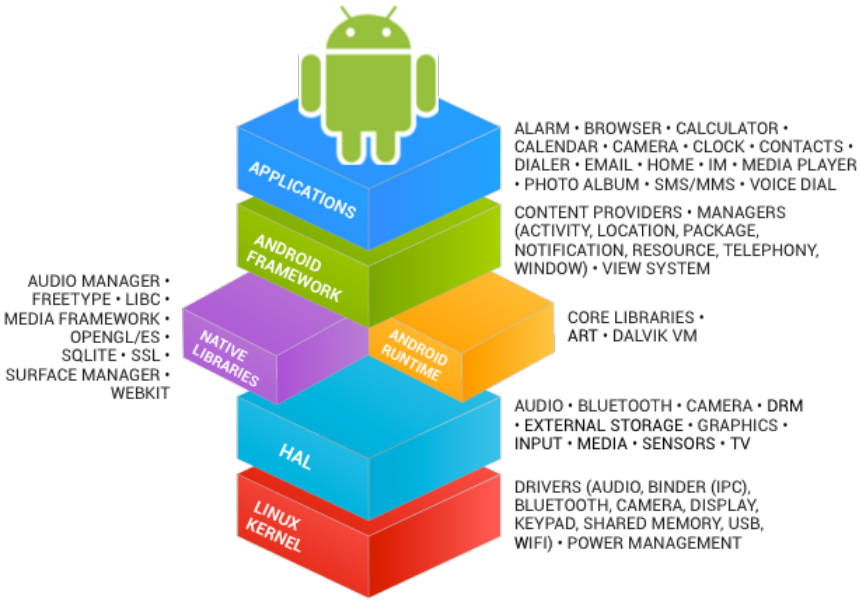


FIGURE 3.11 – Android architecture overview.

disabled.Then *inlining* is an optimization that replaces short and frequently used methods with their actual code. Although, it slightly increases the application size in memory, runtime performance are increased. As there is no method any more, it cannot be hooked in the *ART* interpreter. That optimization is also disabled.

A monitoring routine is added to *ART* interpreter that logs any method call from a targeted application *pid*. A sample of a capture of Java calls is shown in annexes. All these modifications overload the global execution of *Android*. Whereas it is not noticeable for most of the applications, on gaming applications are visibly slow down by this approach. Lastly, the phone is shipped with a real *SIM* card for luring malware payloads with SMS/MMS/Calls. Many malware applications may use it for stealing money with premium numbers, and because we use a real *SIM* card it would actually cost us money. We modified the telephony framework of the *Android* API to reject all outgoing communications except for our own phone number. When a forbidden call is made, the calling UI pops and closes after one second around. This way does not crash applications that rely on calls and SMS.

Network Control & Monitoring

All communications of the instrumented phone pass through a transparent *SSL/TLS* interception proxy behind a wifi access point. This is set by *Panoptes* [47], a wifi man in the middle tool we have developed prior to this thesis. To understand how it works we need to describe a part of the *TLS* handshake. Here is the regular behavior of an *https* request on *Android*:

Android have a keystore of all root certificates the system trusts. When a *SSL/TLS* request is initialized, the requested server sends its certificate. It contains identifying information — like the domain name that must verify the contacted domain name — and a signature that can only be decrypted with the right root CA. The server certificate is tested with each trusted root CA, and if one matches the communication is accepted. Extended information on the *TLS* handshake can be found with the RFC 2246 memo [53].

For our interception system to work, an *SSL/TLS* root certificate from a custom certification authority (CA) is implanted in the keystore of *Android*. When the device requests an *https* web page, the request goes through the proxy. It is parsed and a new one is initialized to be sent to the original recipient. The response is encapsulated in a new *SSL/TLS* response signed by our custom certificate. This custom certificate is dynamically generated with the recipient identifying information

and our custom root CA private key. As the communication is signed by an authority of certification that is known by the client, *Android* accepts it without triggering any warning. Finally, all *HTTP/HTTPS* communications are logged and a report can be generated which is convenient for manual analysis if needed.

This system has been extended to support the manipulation of requests. The objective is to restrict the proliferation of malware and the damage that it may produce. As *Glassbox* runs malware, it may have a negative impact on its environment. An extreme mean could be to disconnect the system from internet but we would see less or no malicious behavior at all for numerous applications. Our design is a trade-off between safety and behavior detection:

- *ClamAV* [54] is used to detect known malware sent through the network. If a malware is detected, the payload is removed from the request and is redirected to *Inetsim* [55], a server that simulates network services. It replies consistently to the requests. That way forbids the communication between the application and internet without crashing it.
- For all other requests, we assess the reputation of the domain name or *IP* address with the *Web of Trust (WoT)*[48] API. *WoT* is a browser extension that filters urls based on different reputation rating. These ratings come mainly from the users. If the request contacts a known address with a good reputation, we forbid the application under test to reach it, then it is redirected to *Inetsim*. The advantages are twofold. The application cannot damage a respectable website, and it pre-filters behaviors for classification.

Finally, features from communications content are collected and the *WoT* reputation scores as well. A sample of a network capture is shown in the annexes.

Automated Application Testing

Smart Monkey is an automated testing program, we have developed, based on *Grey Box* strategies. The context of the application is determined at runtime for the automatic exploration. We use *UIautomator*[49], a tool that can dump the hierarchy tree of the current UI elements present on the screen. It enables us to monitor variables of each UI element at runtime. For a smart exploration, we need to know if we have already processed an element. Unfortunately, elements do not carry such a unique identifier. Nonetheless, we found that elements can be identified to some degree:

- **Strong identification** — Elements can have an associated *ID string* that developers set. Concatenated to the current activity name we have robust identification, but for most of the elements this field is empty. With the same method, a *content description* is sometimes associated with the element. We can also use this for strong identification.
- **Partial identification** — If we do not have access to the previous values, which happens most of the time, we can use lesser discriminative values. Textfields can be set with an initial value, or a printed text can be associated with it. With no better available options we use the element dimensions to identify it. Obviously, when an element is partially identified, a risk of false positive is possible.

Moreover each element carries a list of actions it can trigger. Our automatic exploration consists of systematically triggering all actions of all elements for all activities. We do not try each combination of actions as it would not scale and be mostly redundant. To this basic general process we add targeted actions to trigger more sophisticated behaviors:

- Some textfields of interest are detected like phone number, first or last name, email address, IBAN, country, city, street addresses, password or pin code. These textfields are filled with consistent values accordingly. For this task, we use databases of realistic data (samples can be found in the annexes). Uncategorised textfields are filled with a pseudo-random string.

48. https://www.mywot.com/wiki/API
49. https://stuff.mit.edu/afs/sipb/project/android/docs/tools/help/uiautomator/index.html

- The order of actions done matters. For example, login and password textfields must be filled before validating. In the exploration, filling textfields and check-boxes take precedence over the rest.
- An application can register a receiver for a broadcast *Android* event like the change of phone state or wifi state. It can be done statically in the application manifest or dynamically. Those dynamic receivers could be hidden from static analysis with obfuscation. To trigger the receivers code, we test applications with a list of broadcast events that are often used by malware (a partial list is given in annexes). Moreover, real SMS and phone calls are sent to the real device own number.

We finally use the *Monkey* program during the analysis. It can help to trigger behaviors requiring complex input combination that *Smart Monkey* could miss. At the end comes the cleaning phase. For our real device we keep a white list of regular processes and installed applications — regular, system and device administrator applications. Non-authorized processes are killed and applications uninstalled. Important phone configurations like wifi, data network and sounds are reset to a predefined value.

Experimentation

We use the *average coverage of basic blocks* for quantifying the performance of the application code coverage. It is a measure of the performance of the *Smart Monkey* component of *Glassbox*. Here are definitions of the vocabulary used in the experimentation:

- A *basic block* is an uninterrupted section of instructions. A *basic block* begins at the start of the program or at the target of a control transfert instruction (JUMP/CALL/RETURN). It ends at the next control transfert instruction.
- The *basic block coverage* for an application is the number of unique basic blocks executed at runtime divided by the number of unique basic blocks present in the source code.
- The *average coverage of basic blocks* is the sum of the *basic block coverage* of all applications divided by the number of applications.

Dataset

We use the *AndroCoverage Dataset* [56] for our experimentation. It contains 100 applications from *F-Droid*[50], which is a repository of free and open source (*FOSS*) applications. We have manually selected them with the following criteria for each application:

- It does not depend on a third party library or application as an automatic tool would be unable to install it.
- It does not depend on root privilege. To meet the requirement of a maximum of testing tools configuration, we stick with regular privilege.
- It does not depend on local or temporary remote data. We want the application to be usable worldwide and in the long-term. This category excludes applications for a temporary event or a specific country.

Our goal is to use applications which show a large variety of different and steady behaviors. It is why we predict that performance on the *AndroCoverage Dataset* will be overestimated compared to the average of real applications. This dataset is to be used to compare the performance of different automated testing tools on the same ground.

The *AndroCoverage Dataset* is supplied with tools which instruments the application, adding monitoring routines for code coverage. These tools are partially founded on *BBoxTester* [57], a tool for measuring the code coverage for *Black Box* testing of *Android* applications.

50. https://f-droid.org/

Methodology

Research community used different strategies for automated application testing, with different evaluation methods and different datasets. To promote the successful strategies for future researches on the domain, we need a standard for the experimentation. Otherwise, we cannot compare the results objectively. The research titled *Automated Test Input Generation for Android: Are We There Yet?* [58] shows the re-evaluation on the same ground of 5 published automated testing tools for *Android*. The experimental results found is far from what have been claimed in the published papers. Moreover, according to this study, the *Monkey* program has the best performances above all at around 53% *average coverage of statements* on 68 selected applications. Either all researches on *Android* automated testing are not better than a random event generator or the evaluation methodology lacks pertinence. Our opinion is that a better methodology can highlight the contribution of main researches to the field. To summarize, these observations reveal several problems on the experimental results:

- (1) They are currently not reproducible.
- (2) They cannot be compared to each other.
- (3) They do not highlight the contribution of the evaluated testing method compared to *Monkey* program.

To answer those problems, we propose the following rules:

- (2) [51] A **common performance measure**. We propose the *average coverage of basic blocks*. *Statement coverage* (also called *line coverage*) is considered as the weakest code coverage measure by specialists in software testing. This metric should not be used when another one is available. For an argued reflection about coverage metrics, we refer to the paper *What is Wrong with Statement Coverage* [59].
- (1)(2) A **common dataset** and **common tools** for instrumenting the applications. We propose the *AndroCoverage Dataset* [56].
- (1)(2) A **common configuration** — *Monkey* arguments, a fixed seed for every random number generator used and application versions. This information is either present in annexes of this document or on the *AndroCoverage* Github web page.
- (3) To assess the performance of the combination of both *Monkey* and the evaluated testing method (in our case it is *Smart Monkey*). Compared separately, the performance of *Monkey* and the evaluated method does not highlight new code paths that have been triggered by the evaluated method. A complex method would not seem successful whereas it would have triggered complex conditions that *Monkey* could never find. Moreover, the *Monkey* program is embedded in every *Android* device (real and virtual), it interacts in a very fast pace with the application and produces good results. Then, on an operational situation, it makes sense to use it in addition to any research tool.

Results

Smart Monkey usually runs *Monkey* at the beginning of the analysis. For a fair trial, we tested the performance of its code coverage with and without *Monkey*. The configuration of the *Monkey* tool has been described in annexes. The results are presented on Table 3.6.

The *Monkey* program tends to generate bugs with the instrumentation process. For a significant number of applications (16%) we are unable to get the coverage rate. We note that the same applications crash between *Monkey* and *Smart Monkey* so the crash rate has no effect on the performance comparison between both programs.

The raw results do not give enough insight of the contribution of *Smart Monkey*. We are interested in the new paths that have been triggered compared to the *Monkey* program. Therefore we calculate the increase of *average coverage of basic blocks* of *Smart Monkey* compared to *Monkey*:

51. This number and the following ones refer the problem number it solves

TABLE 3.6 – Code coverage results

Method	Classes average coverage	Methods average coverage	Blocks average coverage	Crash rate
Monkey	32.93%	35.05%	36.32%	16%
Smart Monkey (w/o Monkey)	34.84%	36.68%	37.73%	0%
Smart Monkey (with Monkey)	37.12%	41.6%	41.23%	16%

$$\frac{smartmonkey_{ac}}{monkey_{ac}} = 1.1352$$

Where:

- $smartmonkey_{ac}$ is the average coverage of basic blocks of Smart Monkey.
- $monkey_{ac}$ is the average coverage of basic blocks of Monkey.

It means that the testing strategy we have set up in *Smart Monkey* leads to average increase of 13.52% of *basic blocks coverage*.

Dynamic analysis systems that allow internet communications are vulnerable to fingerprinting. Our platform is not an exception. For example, *Bouncer* [49] has been the target of remote shell attacks [60] that enabled the fingerprinting of the system. The malware gets some information on the system and sends it to a command and control server. Hence, the malware author can reshape the trigger conditions of the logic bomb. We accepted this risk for now. A solution halfway between shutting down all communications and no filtering at all could be to strip all outgoing information — POST request contents/GET url variables/Cookies/Metadata fields. This could lead to a loss of behaviors and the negative impact of such solution needs to be measured. Anyway a smart malware author will eventually find a way to leak remote shell outputs.

The network monitoring has limitations. First, it cannot currently handle all protocols like *POP*, *IMAP* and *FTP* so these protocols are simply blocked. In fact the communications are parsed, to get its content, the destination and the metadata. So this parsing needs to be changed for each protocol. It is an impossible task of adding one by one all protocols, so we would need to measure the protocol usage and implement the most used ones. At last, there is a countermeasure to our *SSL/TLS* interception, namely *certificate pinning*. The requirement of the interception is the implantation of a custom root certificate in the *Android* keystore of trusted certificates. An application can choose to discard the *Android* keystore and to embed its own. Therefore when a communication, encrypted with our custom certificate, is checked, the communication is rejected. This technique is used in many banking applications, as we have shown at Black Hat [47]. In fact the point of view of the bank is: the user *OS* cannot be trusted. Although we have no evidence that it happens for malware, it may be used by an avant-gardist malware and others would follow the trail. It is inconvenient for malware authors to buy a certificate signed by an authority of certification, as a payment trace could identify them. Despite of that, it is possible to get a valid certificate from *Let's Encrypt*[52], or to control a legitimate server via hacking and use it as a relay for the *C&C* server. In these cases, *certificate pinning* could be used for hiding communications from analysts or interception systems. A counter to this technique is instrumentation. By monitoring arguments of the *SSL/TLS* encryption method, one can get the plaintext communications. We have done it manually for some banking applications [47] with *APImonitor* [45], but doing it automatically is another issue. Applications that use *certificate pinning* generally embed their own library for *SSL/TLS* encryption, so detecting dynamically which call is the *SSL/TLS* encryption method can be challenging.

Last, the cleaning phase of *Glassbox* fits the security needed for a prototype. However, to move to an operational situation with malware that could execute 0-day root exploits, we need a real factory-reset of the phone. This is why we plan to integrate the open source project *BareDroid* as a part of *Smart Monkey*, for its factory-reset capability on real device.

52. https://letsencrypt.org/

This study contributes to the domain of dynamic analysis system for *Android* in three ways. First, we presented *Glassbox* a functional prototype of a platform that uses real devices, controls network and GSM communications to some extends and monitors Java calls, systems calls and network communication content. Second, we experimented *Smart Monkey*, an automatic testing tool with a *Grey Box* testing strategy. We showed that it enhances the application code coverage compared to the common *Black Box* testing tool called *Monkey*. Last, we presented a method of evaluation of automated testing tools to research community. This method covers the problems of reproducibility, the comparison with other works and of the contribution measurement of the tool. We made the dataset available on Github under the name *AndroCoverage*.

The next step is to use *Glassbox* on malware/benign applications and to use the features found on a machine learning algorithm. We are working on the classification of these data with a neural network.

BIBLIOGRAPHY

[1] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis–1,000,000 apps later : A view on current android malware behaviors. In *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014 Third International Workshop on*, pages 3–17. IEEE, 2014. 47, 66, 67, 68, 69

[2] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin : Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014. 33, 48, 49, 59, 117, 137, 152, 154

[3] Hyunjae Kang, Jae-wook Jang, Aziz Mohaisen, and Huy Kang Kim. Detecting and classifying android malware using static analysis along with creator information. *International Journal of Distributed Sensor Networks*, 11(6) :479174, 2015. 48

[4] Mohsen Damshenas, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Ramlan Mahmud. M0droid : An android behavioural-based malware detection model. *Journal of Information Privacy and Security*, 11, 2015. 48

[5] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*, pages 252–276, Bonn, Germany, 2017. Springer. 48

[6] Paul Irolla and Alexandre Dey. The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques*, pages 1–5, 2018. 49, 117, 137, 152

[7] Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. Droidkin : Lightweight detection of android apps similarity. In *International Conference on Security and Privacy in Communication Systems*, pages 436–453. Springer, 2014. 50

[8] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on pattern analysis and machine intelligence*, 27(8) :1226–1238, 2005. 51, 153

[9] Andrew W Moore. Cross-validation for detecting and preventing overfitting. *School of Computer Science Carneigie Mellon University*, 2001. 52

[10] Diederik Kingma and Jimmy Ba. Adam : A method for stochastic optimization. *arXiv preprint arXiv :1412.6980*, 2014. 15, 53, 104, 118

[11] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions : User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, page 3. ACM, 2012. 56

[12] Matthew Leeds, Miclain Keffeler, and Travis Atkison. A comparison of features for android malware detection. In *Proceedings of the SouthEast Conference*, pages 63–68. ACM, 2017. 58

[13] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell. Androdialysis : Analysis of android intent effectiveness in malware detection. *computers & security*, 65 :121–134, 2017. 58

[14] Gunjan Kapse et al. Detection of malware on android based on application features. *(IJCSIT) International Journal of Computer Science and Information Technologies*, 6, 2015. 58

[15] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. Puma : Permission usage to detect malware in android. In *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*, pages 289–298. Springer, 2013. 58

[16] Hyun Jae Kang, Jae-wook Jang, Aziz Mohaisen, and Huy Kang Kim. Androtracker : Creator information based android malware classification system. In *Information Security Applications-15th International Workshop, WISA*, volume 8909, 2014. 59

[17] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Automated static code analysis for classifying android applications using machine learning. In *2010 International Conference on Computational Intelligence and Security*, pages 329–333. IEEE, 2010. 59

[18] Yajin Zhou and Xuxian Jiang. Dissecting android malware : Characterization and evolution. In *security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012. 24, 59

[19] BooJoong Kang, Suleiman Y Yerima, Kieran McLaughlin, and Sakir Sezer. N-opcode analysis for android malware classification and categorization. In *Cyber Security And Protection Of Digital Services (Cyber Security), 2016 International Conference On*, pages 1–7. IEEE, 2016. 59

[20] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupe, et al. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 301–308. ACM, 2017. 59, 60

[21] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer : Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013. 59, 143

[22] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6(1) :37–66, 1991. 59

[23] Feng Shen, Justin Del Vecchio, Aziz Mohaisen, Steve Ko, and Lukasz Ziarek. Android malware detection using complex-flows. *IEEE Transactions on Mobile Computing*, 2018. 59

[24] Ludovic Apvrille and Axelle Apvrille. Pre-filtering mobile malware with heuristic techniques. *Proceedings of GreHack*, 2013. 60

[25] Ludovic Apvrille and Axelle Apvrille. Identifying unknown android malware with feature extractions and classification techniques. In *TrustCom/BigDataSE/ISPA (1)*, pages 182–189, 2015. 60

[26] Axelle Apvrille and Ludovic Apvrille. Sherlockdroid, an inspector for android marketplaces. *Hack. lu, Luxembourg*, 2014. 31, 60

[27] Karim O Elish, Xiaokui Shu, Danfeng Daphne Yao, Barbara G Ryder, and Xuxian Jiang. Profiling user-trigger dependence for android malware detection. *Computers & Security*, 49 :255–273, 2015. 61

[28] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013. 61

[29] Éric Filiol and Paul Irolla. (in)security of mobile banking...and of other mobile apps, 2015. 62

[30] Alexandre Dey, Loic Beheshti, and Marie-Kerguelen Sido. Health state of google's playstore. 2018. 30, 63

[31] Paul Sabanal. Hiding behind art. *Online : https ://www. blackhat. com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp. pdf*, 2015. 66

[32] Thorsten Schreiber. Android binder–android interprocess communication. In *Seminar thesis, Ruhr-Universität Bochum*, 2011. 66

[33] T. Bläsing, L. Batyuk, A. D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 5th International Conference on*, pages 55–62, Oct 2010. 66, 67, 69

[34] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid : behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011. 67, 68, 69

[35] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid : an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012. 67, 69

[36] Lok Kwong Yan and Heng Yin. Droidscope : seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, 2012. 66, 67, 68

[37] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground : automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013. 67, 69

[38] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 899–914. IEEE, 2015. 67, 69

[39] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glauco Barroso Junquera, and Paulo Lício de Geus. Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1) :9–17, 2015. 67, 68, 69

[40] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid : Automatic reconstruction of android malware behaviors. In *NDSS*, 2015. 66, 67

[41] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20. ACM, 2015. 67, 68, 69

[42] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8. ACM, 2016. 67, 68, 69

[43] Michelle Y Wong and David Lie. Intellidroid : A targeted input generator for the dynamic analysis of android malware. 2016. 67, 69

[44] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Acquiring and analyzing app metrics for effective mobile malware detection. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 50–57. ACM, 2016. 67, 68, 69

[45] pjlantz. Droidbox - apimonitor.wiki, 2012. [Online] https://code.google.com/archive/p/droidbox/wikis/APIMonitor.wiki. 68, 76

[46] Github - smali readme, 2009. [Online] https://github.com/JesusFreke/smali. 68

[47] Eric Filiol and Paul Irolla. (in)security of mobile banking... and of other mobile apps. Black Hat Asia 2015. 68, 72, 76

[48] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. Baredroid : Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 71–80. ACM, 2015. 69

[49] Hiroshi Lockheimer. Android and security, 2012. [Online] http://googlemobile.blogspot.fr/2012/02/android-and-security.html. 69, 76

[50] Hitesh Dharmdasani. Android.hehe : Malware now disconnects phone calls, 2014. [Online] https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html. 70

[51] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus : automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225. ACM, 2014. 70

[52] Paul Irolla and Eric Filiol. Glassbox : Dynamic analysis platform for malware android applications on real devices. *arXiv preprint arXiv :1609.04718*, 2016. 70, 138, 154

[53] T. Dierks and C. Allen. The tls protocol version 1.0, 1999. [Online] http://www.ietf.org/rfc/rfc2246.txt. 72

[54] Tomasz Kojm. Clamav, 2004. 73

[55] Thomas Hungenberg and Matthias Eckert. Inetsim : Internet services simulation suite, 2013. 73

[56] Androcoverage dataset, 2016. [Online] https://github.com/androcoverage/androcoverage. 74, 75

[57] Yury Zhauniarovich, Anton Philippov, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Towards black box testing of android apps. In *2015 Tenth International Conference on Availability, Reliability and Security (ARES)*, pages 501–510, August 2015. 74

[58] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android : Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440. IEEE, 2015. 75

[59] Steve Cornett. What is wrong with statement coverage, 1999. [Online] http://www.bullseye.com/statementCoverage.html. 75

[60] Percoco and J. Nicholas. Adventures in bouncerland. 2012. 76

CHAPTER 4

Neural Networks

Contents

Neural networks is a vast and old field of study, in this part we have selected the elements of the neural network theory that is useful for this thesis and more precisely for the writing of *libmla* algorithms. We do not claim to cover the whole of the theory and practical problems, which elsewhere has already been formulated in depth in works such as *Neural networks: a systematic introduction* [1] and *Deep Learning* [2]. In a first section we describe the Theory of Neural Networks. Then, in the second section, we cover its problems of implementation. Last, we discuss Deep Learning and its applicability to the Android malware detection problem.

4.1 Neural Network Theory

In this part we introduce Machine Learning and Neural Networks. We extend the theory up to the training of a *Multi-Layer Perceptron* (*MPL*) . For various reasons such as speed of training and execution, and relatively low needs for training data — reasons that will be discussed later — it is towards this category of neural network that we went.

4.1.1 Introduction

The human mind has always been a subject of study, and more recently the intelligence. How does it work? Can we replicate it? Can we imitate it? Can we build an intelligent machine? During the 40s, a movement named *Cybernetics* came into being, composed of scientists from a wide range of domains. It has given rise to Artificial Intelligence, neuroscience and current social sciences. The *Macy conferences* were the source of Cybernetics as a study domain. They have

been founded by Warren Sturgis McCulloh (1899-1969), researchers in neurology. He envisions, like Allan Turing, that functions of the mind could be described as mathematical functions. He published in 1943 with Walter Pitts, *A logical calculus of the ideas immanent in nervous activity* [3]. It is the first model of artificial neuron. The Macy conferences are a cross-domain group of mathematicians, logicians, anthropologists, psychologists and economists. Its objective is to promote a general science of the human mind operations. Arturo Rosenblueth presents, upon these conferences, *Behaviors, purpose, and Teleology* [4] published in 1943 with Norbert Wiener and Julian Bigelow. This article becomes the milestone of Cybernetics. In particular, it defines that an object is controlled by the margin of the error which separates, at any given time, from the objective the object tries to reach. It is the description of the mechanism of error feedback, which is nowadays the principle of neural network training algorithm.

Cybernetics is a unified vision of the domains of automatic, electronic and theory of information, applied to the control of systems. The work of Norbert Wiener in 1948, *Cybernetics or Control and Communication in the Animal and the Machine* [5] is the founding text of Cybernetics. One of the lesson of this book is *any organism maintains its cohesion by means of acquisition, use, retention and transmission of information*. It describes mechanisms of feedback and auto-regulation. It is the science of control of systems and information aiming at managing a system, by controlling its environment. Cybernetics bred the robotization of production, the control and regulation of global financial networks, new methods of management, neuromarketing, etc. In 1950, in a book published by the name *Cybernetics and Society* [6], Norbert Wiener predicts the end of human work, replaced by intelligent machines and warns political leaders against of the usages of Cybernetics, which would lead to mass unemployment, social exclusion and in the long run to the obliteration of democracy.

Currently, Artificial Intelligence is a vast scientific domain. The most advanced discipline of Artificial Intelligence is Machine Learning (*ML*), i.e. algorithms which are able to generalize the solution of a problem from samples of resolutions of this problem by men and women. Machine Learning is currently in use when the problem to solve is:

- Much too complex, meaning there is too much variables to be taken into account for a human being. For instance: the regulation of traffic lights to reduce traffic jam. It involves too many variables, because every action impacts the whole traffic of a city. This subject has been extensively studied and solutions with neural networks have been provided [7] [8] [9].
- Uninteresting and repetitive. For instance, transcription of audio recordings. Doctors currently use speech to text software with artificial intelligence (ex: Dragon [1]). Before that, this task was done — and is still done in some places — by medical secretaries.
- Not profitable regarding the cost. Artificial intelligence can do this task faster than a human, making it profitable.

More generally, Machine Learning is used to optimize processes and to discover relations between objects and categories. In particular neural networks are used in a wide range of application domains:

- *IBM Watson*: is able to understand and answer questions expressed in natural language. It won *Jeopardy!*, a general knowledge TV game; it has been trained to diagnose diseases regarding symptom descriptions from patients.
- *Google Deep Dream*: is able to apply the style of an image over another one. Figure 4.1.
- Recognition and generation of manuscript characters. In the example (Figure 4.2), a neural network are trained on the writing style of several authors. It is then able to produce an image from a digital text that mimics handwritten text.

1. https://www.nuance.com/dragon/transcription-solutions.html

FIGURE 4.1 – Google Deep Dream

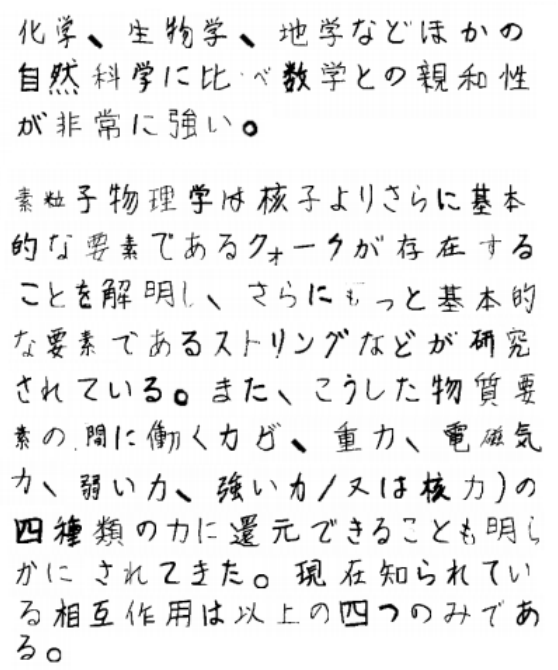


FIGURE 4.2 – Manuscript text generation by neural network (Fujitsu — Loïc Behesti from ESIEA)

Definitions & Vocabulary

Any machine learning algorithm works with a dataset. We start by formalizing the dataset and its constituting elements. The dataset is composed of samples. For each sample, there is an *input vector* associated with an *expected output vector*.

Input variable, input vector, observation:

$$\mathrm{X} = (x_1, x_2, \ldots, x_d)$$

Where:

- x_i: descriptive variable, feature
- d: dimension of the input space

Expected output, signification, desired value:

$$\mathrm{Y} = (y_1, y_2, \ldots, y_s)$$

Where:

- y_i is a real: quantitative variable, variable to predict. When y_i is a real, the problem to solve is classified as *regression*, *approximation* or *optimization*.
- y_i is an integer: qualitative variable, class, truth (0/1). When y_i is an integer, the problem to solve is classified as *discrimination*.
- s: dimension of the output space.

A dataset composed of n samples, is represented by the following matrices:

Dataset - Input

$$\begin{bmatrix} \mathrm{X}_1 \\ \mathrm{X}_2 \\ \vdots \\ \mathrm{X}_n \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1d} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & x_{n3} & \dots & x_{nd} \end{bmatrix}$$

Dataset - Output

$$\begin{bmatrix} \mathrm{Y}_1 \\ \mathrm{Y}_2 \\ \vdots \\ \mathrm{Y}_n \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} & y_{13} & \dots & y_{1s} \\ y_{21} & y_{22} & y_{23} & \dots & y_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ y_{n1} & y_{n2} & y_{n3} & \dots & y_{ns} \end{bmatrix}$$

Categories of Learning

There is two general categories of machine learning. The first one, *supervised learning*, is the training of an artificial intelligence for a task where we have a dataset containing examples of this problem, solved by humans. In the second one, *unsupervised learning*, we have data that have not been labeled. An artificial intelligence groups similar data together, producing clusters.

Supervised Learning

The learning rule is external to the system. To each *input data*, there is an *expected output value*, the learning algorithm processes the *error* between the *expected output value* and the *actual output value* and modifies the internal parameters accordingly. Some example of *(corrective) supervised learning* problems:

- Regarding some input vectors, (x_1, x_2), and expected output values, $f(x_1, x_2) = e^{-x_1} + 2.e^{-x_2}$, find an approximation of $f(x_1, x_2)$.
- Recognize the handwritten character *'c'* in an image. Input vectors are the values of image pixels and the expected value is 1 when the character is *'c'* and 0 otherwise.

The *supervised learning* category can be subdivided into two kinds of subcategories: *corrective learning* (previous examples) and *reinforcement learning*. In *reinforcement learning*, there is no *expected output*. An external system gives a cost value to the *actual output*, and the goal of the learning algorithm is to minimize the cost value.
Here is a non-exhaustive list of supervised learning algorithms:

- *Neural network*
- *SVM* (Support Vector Machine)
- *Decision Tree* (or *Random Forest* — an aggregation of *Decision Tree*)
- *HMM* (Hidden Markov Model)
- *Naive Bayes classifier*

Unsupervised Learning — Clustering

The learning rule is internal to the system. It could be the extraction of recurrent patterns from input data or clustering similar samples. Then it is up to someone to give a meaning to these patterns/clusters. In this category of problems there is no expected output. The Figure 4.3 (public image) shows an example of a clustering algorithm (in this case, the *OPTICS* algorithm) on Gaussian noise.

In this problem, input vectors are coordinates of the points $\mathrm{X}_i = (x_i, y_i)$. The learning rule can use a variety of criteria to cluster data but it involves the distance between samples.
Here is a non-exhaustive list of different algorithms:

- *K-mean clustering*
- *Hierarchical clustering*
- *Self-organizing map*

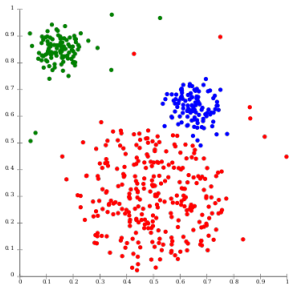


FIGURE 4.3 – Finding three clusters in gaussian noise input data

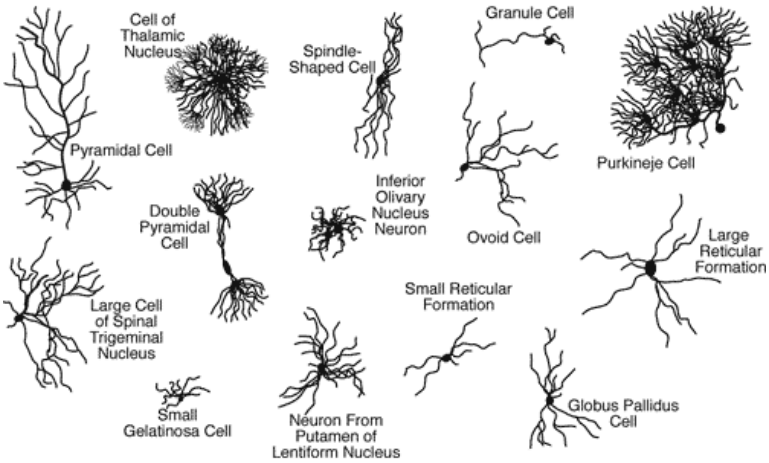


FIGURE 4.4 – Variety of neurons

Unsupervised learning algorithms are a research direction less active than supervised learning. The way we currently use unsupervised learning is to find clusters on data. These clusters are then labeled by humans and finally the trained AI is often used to classify new instances. It is used as a trained supervised learning AI. Since we have at disposal huge quantities of labeled data for lots of different problems, we prefer using directly supervised learning as it is more efficient. Research in machine learning is quasi-exclusively oriented toward supervised learning algorithm, especially *neural networks* and *deep learning*. Nevertheless, unsupervised learning is the way AI will evolve from a dedicated task solver to a general intelligence that is able to solve problems for which it has received no training.

4.1.2 Neural Networks — From a Biological Paradigm

In biology, information in neural network is stored on contact points between neurons. Other shapes of storage are known. Neural networks are complex systems that organize themselves.

Architectures of Neurons

Nervous system has different architectures regarding the regions of the cerebral cortex. Each of them is composed of similar blocks of neurons. When analyzing a human brain under a microscope, we can see different shapes of neurons in Figure 4.4 (source [10]).

There are around 10,000 specific types of neurons. Each of them have different characteristics and functions. In Figure 4.5 (credit : Thomas Deerinck and Mark Ellisaman, 2004) and 4.6 (source [11]) we show that neurons are organized in layers of similar neurons.

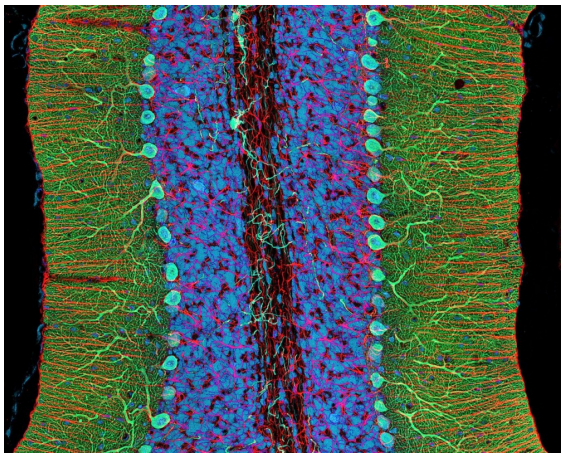


FIGURE 4.5 – A close view on a layer of neurons

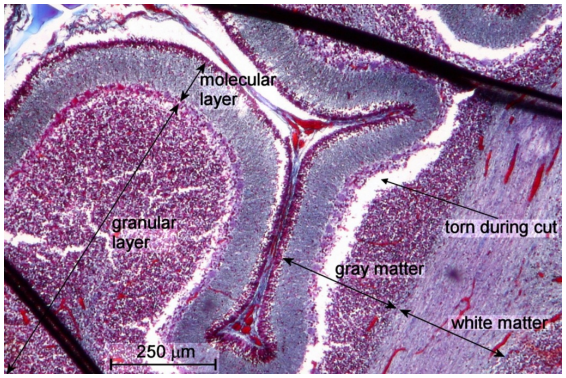


FIGURE 4.6 – Organization of neuron layers

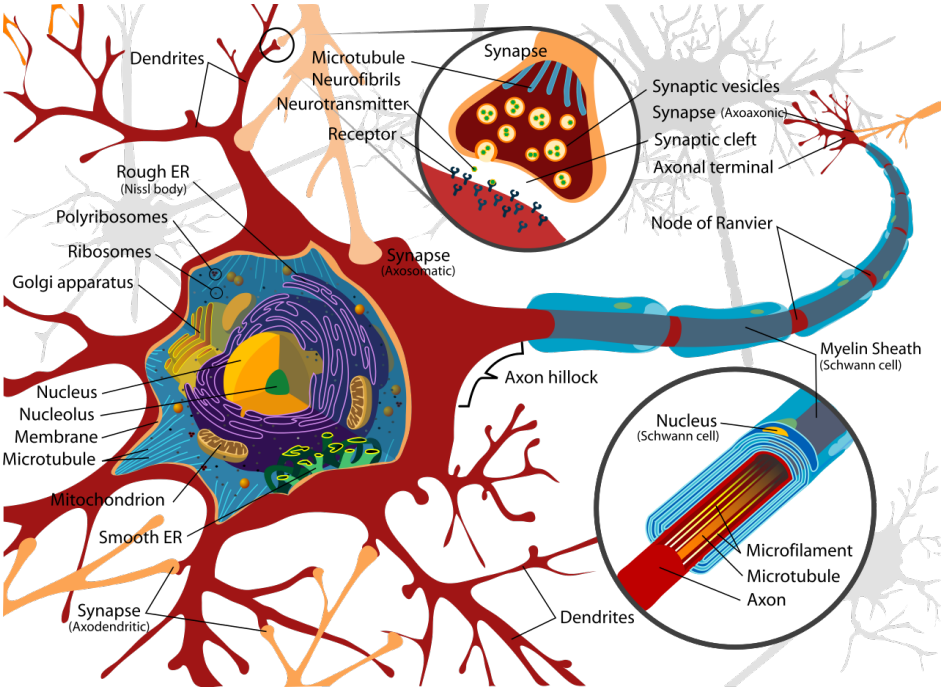


FIGURE 4.7 – Neuron internal structure

In Figure 4.7 (public image, author: Mariana Ruiz Villarreal) here is a detailed scheme of a neuron internal structure.

Neurons emit and receive signals. Left branches are input transmission channels for information and is called *dendrite*. The dendrites receive information at the contact point of the *synapse* of other neurons. Output signal is transmitted by the *axon*. The size of the axon greatly vary regarding zones to connect. While most axons are microscopic, neurons of the spine have axons that go over a meter (the size of the spine). The *core* of the cell produces the required energy for its own operation.

We use these four elements of the neurons — dendrite, synapse, core and axon — to build a mathematical model of the neuron. These elements are regarded as the minimal element shaping the fundamental structure of a biological neuron. Formal neurons, for numerical processing, have by analogy input/output channels and a core. The synapes are simulated by a numerical weight on each input connection (that is also an output connection from a preceding neuron).

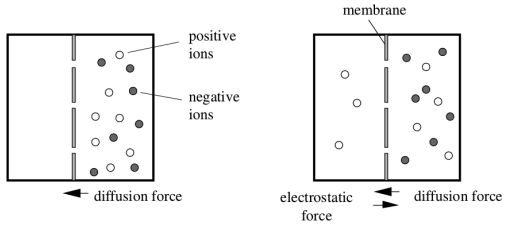


FIGURE 4.8 – Ions movement around the neuron membrane

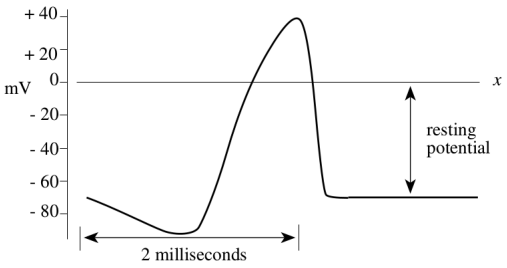


FIGURE 4.9 – Potential for action

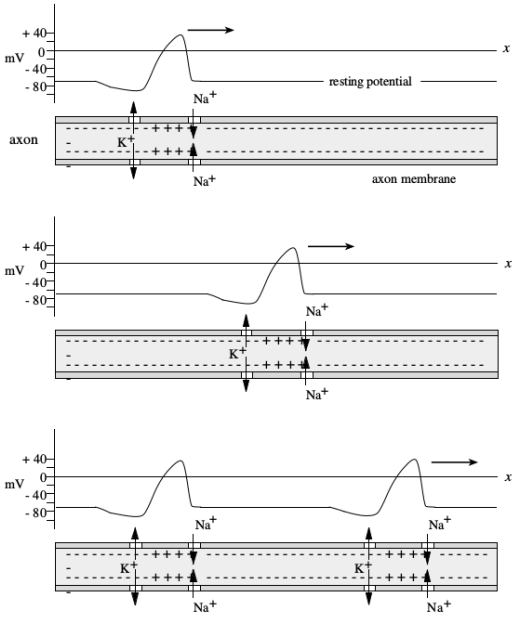


FIGURE 4.10 – Transmission of the potential for action

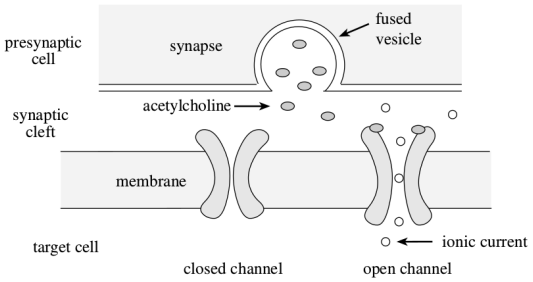


FIGURE 4.11 – Chemistry of the synapse

Information Transmission Inside a Neuron

Information transmission between neurons is made by electrical discharges. It is not a simple electronic transport such that in a copper wire. Evolution came up to another solution involving positive and negative ions that move through semi-permeable membranes.

Salts, present in our body, dissolve into intra and extra cellular fluids producing ions. Sodium chloride ($NaCl$) dissolves into positive sodium (Na^+) and in negative chlorine (Cl^-). Other ions are present like potassium (K^+) and calcium ($Ca^{2}+$). Permeability of membranes is determined by the number and the size of pores. Membrane channels are selectively permeable at sodium, potassium or calcium. In Figure 4.8 (source [1]), to the left it is the initial state of information transmission.

Positive ions go through the membrane, leading to a repulsion of negative ions. A potential difference arises, also known as inverse potential, and the system starts to behave like a little battery. Eventually, there is no more positive ion inside the membrane and the system become stable. The inverse potential is then of $-80mV$ at equilibrium. The cell also owns sodium ions that have an equilibrium potential of $58mV$. A phenomenon appears as soon as the cell comes to $-80mV$. It will not be detailed here, but pores start opening to let sodium ions get out and let potassium sodium come in. When all sodium ions are outside the cell, every pores open and the equilibrium potential of the cell becomes $-70mV$. Overall, this phenomenon is a change of polarity also known as *potential for action* (Figure 4.9, source [1]). Every perturbations, namely the opening and closing of channels are transmitted through the axon like falling dominos (Figure 4.10, source [1]).

Each neuron is a system that transmits an all or nothing kind of information. At this level, we can talk about digital transmission of information. Neuron can vary the frequency of the apparition of the *potential for action*. Actually, the message which is transmitted depends on the impulsion frequency. When the *potential for action* reach synapses, it frees neurotransmitters (Figure 4.11, source [1]).

Some neurotransmitters attach to the ionic channels of the following neuron. There are different kinds of neurotransmitters. Some of them lead to the opening of potassium pores. In this case, we say the neuron is *excited*, and the mechanism of *potential for action*, that we described previously, is started. Other kinds of neurotransmitters lead to the closing of the ionic channels for a while. In that case, we say the neuron is *inhibited*, it is unable to start a *potential for action* for a short delay.

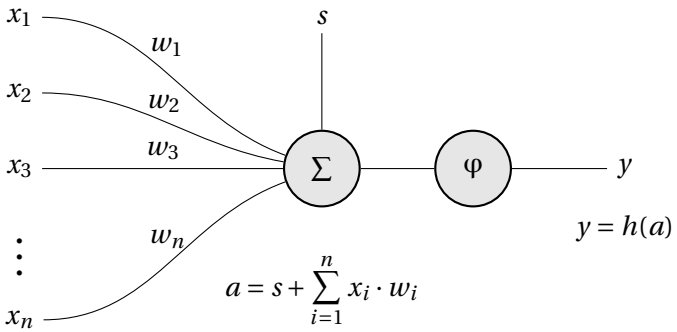
In neurons, the information is stored in the synapses. Extended repetitions of the excitation change the minimum level to excite a neuron, also known as the threshold of excitation. A similar mechanism happens for inhibiting neurotransmitters. It is this mechanism that enables the brain to *learn*.

From this biological paradigm comes the artificial neural network where the formal neuron is an elementary processor unit. On the first hand, we see how this unit works, and what are its capabilities and limitations. On the other hand, we build a network of formal neuron and we present a particular configuration, namely the *multi-layer perceptron* that is inspired by the neural architecture of the visual cortex. It is a general configuration that is able to solve efficiently a wide range of problems.

4.1.3 Formal Neuron

Description of a Formal Neuron

The neuron is the elementary processor of the neural network. The following figure represents the model of the formal neuron:



Where:

- $x_1 \dots x_n$ are the input data variables.
- $w_1 \dots x_w$ are the (synaptic) weights of the network. Each weight is associated with an input value.
- s is the *activation threshold* or the *bias*. Its purpose will be explained later.
- y is the neuron output.
- a is a weighted sum of inputs, this value is called the *activation level* of the neuron.
- h is called transfer function or activation function. There are several activation functions for different purposes.
- All the neuron variables are often bounded by the intervals $[-1,1]$ or $[0,1]$, but there is no formal obligation.

Common activation functions:

- *Copy function (1).* This function is often used when the neuron purpose is the approximation of a linear function.

$$(1): h(x) = x$$

- *Step function (2).* This function is used when the neuron learns to approximate a boolean function.

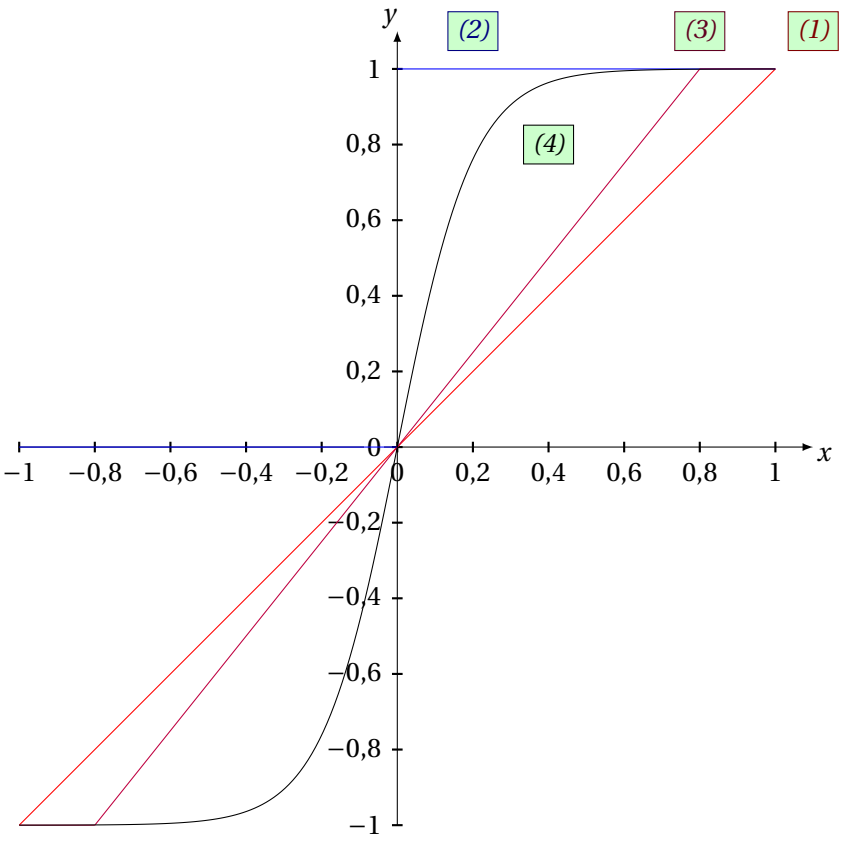
$$(2): h(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$$

- *Picewise linear function (3).* This function is sometimes used for classification.

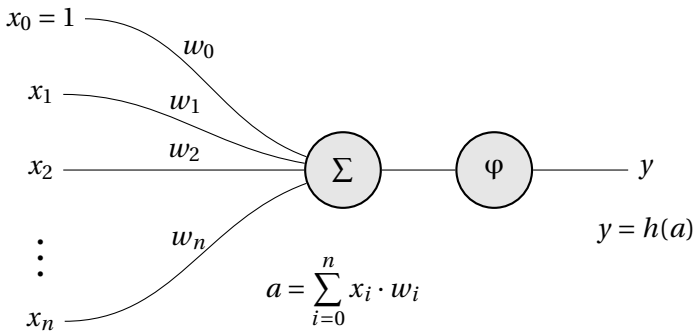
$$(3): h(x) = \begin{cases} 1 & \text{if } x > p \\ x/p & \text{if } -p < x < p \text{ , with } 0 < p < 1 \\ -1 & \text{else} \end{cases}$$

- *sigmoid function (4).* This function is preferred for classification.

$$(4): h(x) = \frac{2}{1 + e^{-k \cdot x}} - 1$$



The *threshold* is a parameter independent from the input which allows the activation curve to be shifted to the right or to the left. In fact, it gives the capacity to the neuron to correct biased data. Actually the *threshold* is also called *bias*. The *threshold* is a variable to be trained, as the weights. To simplify the representation of the neuron, we integrate the bias in the weights. It is equivalent to represent it as a *threshold* or a weight with a constant associated input at 1:



At the beginning all weights are randomly set, then they are optimized step by step with a learning algorithm using a pool of known data. When this learning phase is over, the neuron is ready to process new data. For a single neuron, we can use the following simple learning algorithm of weight optimization:

```
Until the learning is over, do
  For each couple (x⃗, y_ex), in the learning dataset, do
    a = Σ_{i=0}^{n} x_i · w_i
    y = h(a)
    e(t) = y_ex − y

    For each i ∈ [0, n], do
      w_i(t+1) = w_i(t) + μ · x_i · e(t)
    End for

    learning is over if a stopping condition has been reached
  End for
End until
```

Where:

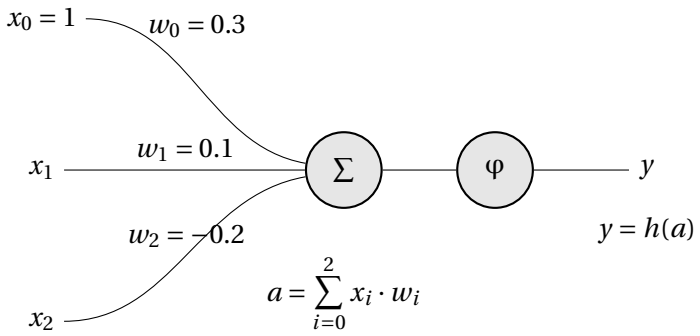
- t is the current iteration of the learning process.
- $e(t)$ is the current *error* between the *expected output* and the *actual output.*
- μ is the *learning coefficient*, a value between 0 and 1 which controls the speed of the learning. With a greater μ, the learning will be faster, but less precise and vice versa.

Several stopping conditions can be used:

- The weights do not change anymore because they have reached their optimal value.
- The current *error* is below an arbitrary threshold.
- The learning algorithm has reached an arbitrary maximal iteration.

Learning with One Neuron

In this section we are going to see what we can learn with one neuron and its limitations. For both examples, the following neuron will be used:



Where:

- $w_0 = 0.3$, $w_1 = 0.1$, $w_2 = -0.2$
- $h(\mathrm{X}) = \begin{cases} 1 & \text{if X} > 0 \\ 0 & \text{else} \end{cases}$
- $\mu = 0.5$
- The learning stops when the weight values converge, meaning that a pass on the whole dataset does not change any weight.

In the following examples, we train one neuron to learn the truth tables of two logic gates. For these examples, we train the neurons until the weight values converge. An iteration correspond to the training on one sample (a couple of x_1, x_2 input values and their corresponding expected output y_{ex}). We train each samples from the dataset, and when we ran out of samples, we start again from the first sample of the dataset. We assume that the weights have converged when their values do not change when we train then with every sample of the dataset.

OR Logic Gate

We train the neuron to learn the *OR* logic gate truth table:

x_1	x_2	y_{ex}
0	0	0
1	0	1
0	1	1
1	1	1

At $t = 0$:

$$a = 0.3$$

$$y = 1$$

$$e(t) = y_{ex} - y = -1$$

As x_1 and x_2 are null, only w_0 is updated.

$$w_0(t+1) = w_0(t) + \mu.x_0.e(t) = 0.3 - 0.5 * 1 * 1 = -0.2$$

At $t = 1$:

$$a = 0.1 - 0.2 = -0.1$$

$$y = 0$$

$$e(t) = y_{ex} - y = 1$$

$$w_0(t+1) = w_0(t) + \mu.x_0.e(t) = -0.2 + 0.5 * 1 * 1 = 0.3$$

$$w_1(t+1) = w_1(t) + \mu.x_1.e(t) = 0.1 + 0.5 * 1 * 1 = 0.6$$

At $t = 2$, no weight update since $e(t) = 0$.
At $t = 3$, no weight update since $e(t) = 0$.
At $t = 4$, we start to learn the truth table from the beginning and w_0 is updated with value -0.2.
At $t = 5$, no weight update since $e(t) = 0$.
At $t = 6$:

$$y = 0, e(t) = 1, w_0(t+1) = 0.3, w_2(t+1) = 0.3$$

At $t = 7$, no weight update since $e(t) = 0$.
At $t = 8$, we start to learn the truth table from the beginning and $w_0(t+1) = -0.2$.
At $t = 9$, no weight update since $e(t) = 0$.
At $t = 10$, no weight update since $e(t) = 0$.
At $t = 11$, no weight update since $e(t) = 0$.
At $t = 12$, no weight update since $e(t) = 0$.

We consider that weights have converged since their values do not change when we train them over each input of the learning dataset. The neuron is trained and is able the reproduce the exact outputs of the *OR* logic gate.

XOR Logic Gate

Train the neuron on the *XOR* logic gate truth table:

x_1	x_2	y_{ex}
0	0	0
1	0	1
0	1	1
1	1	0

By applying the same method used on the *OR* logic gate, we find that the algorithm keep updating the weights endlessly. The weights do not converge. Hence our current neuron is unable to learn the *XOR* logic gate. In fact, a single neuron is able to find a linear separation within the inputs values in which each subset corresponds to an output value. In the first example, it was able to learn the OR logic because it is a problem which can be linearly separated. To the contrary, *XOR* cannot be linearly separated (Figure 4.12).

Intuitively, to bypass this limitation, a naive solution would be to use two neurons to learn *XOR* because the space is separated by two bounds. However, three neurons are mandatory to solve the *XOR* problem. A neuron is only able to learn a basic logic gate as *OR*, *AND*, *NOR* or *NAND*. But *XOR* is the composition of three basic logic gates:

$$x_1 \text{XOR} x_2 = (x_1 \text{OR} x_2) \text{AND} (x_1 \text{NAND} x_2)$$

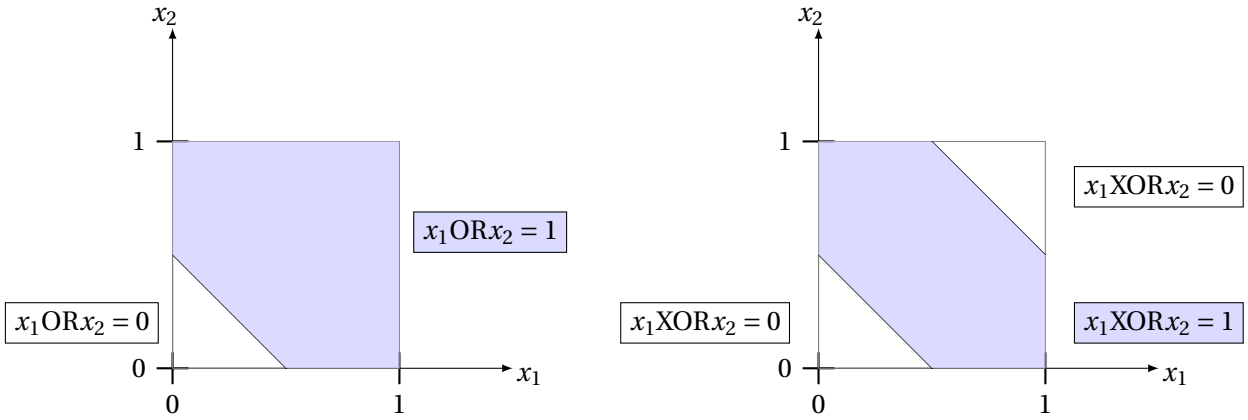
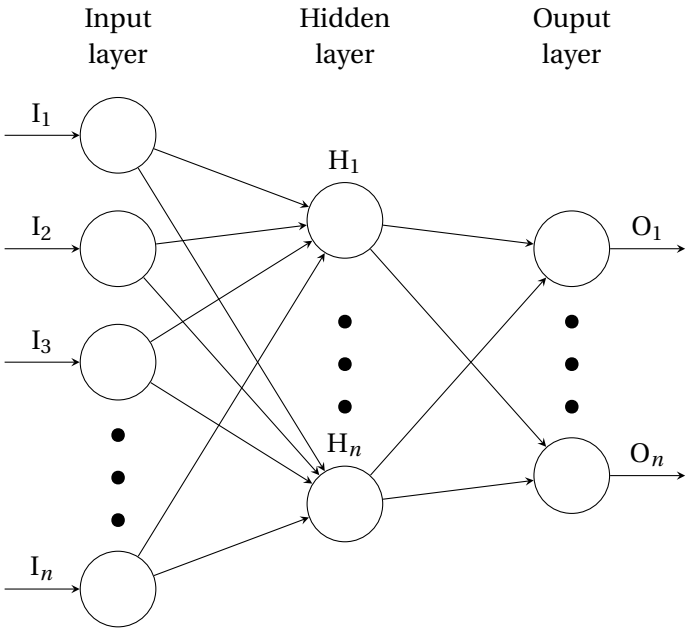


FIGURE 4.12 – Separation of the space of the OR and XOR logic gates

We can foresee that we can assemble neurons in a network for learning any kind of electronic system. A question still remains about more complicated problems, especially when we cannot list all possible inputs and outputs? We will see in the next part how to build and use neural networks as a heuristic system to solve approximation and classification problems.

4.1.4 (Artificial) Neural Network

The combination of formal neurons in network enables the resolution of problems of classification and approximation.



Overview of Neural Networks

Neurons can be arranged in many network architectures, for different tasks such as data compression or image recognition. One of the most common kind of *artificial neural network* (*ANN*), the *Multi-Layer Perceptron (MLP)*, is described with the above scheme. In such network, each layer is composed by several neurons. A *MLP* is composed by: one input layer, one or more hidden layers and each neuron of the input and hidden layer is connected to each neuron of the next layer. i.e. each neuron output value of a layer becomes the input value of each neuron of the following layer.

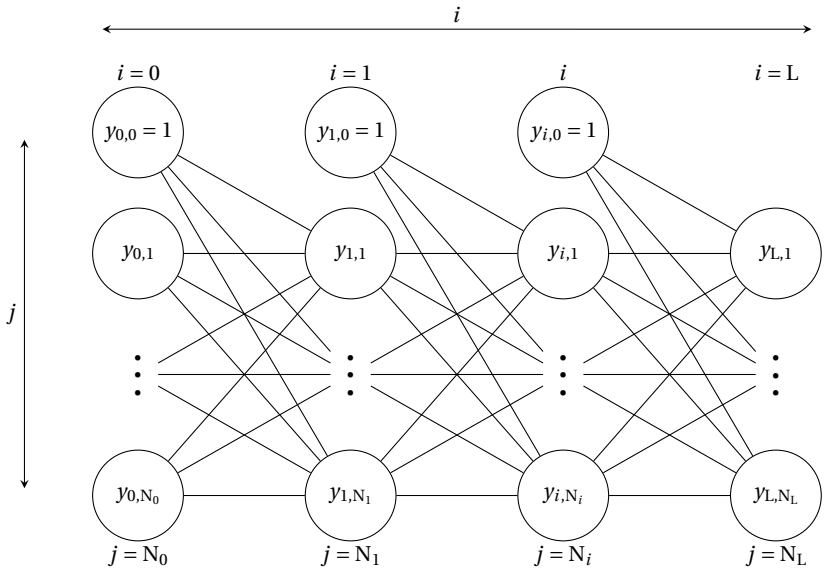
In the ANN paradigm, everything is a neuron, but the input neurons do not process any data, its role is to feed the first hidden layer the input values from the samples. There can be several hidden layers, but one is sufficient to solve any problem. The number of neurons in the hidden layer is chosen but it corresponds to the complexity of the problem. There is no general method that helps us to guess this number in advance, we have to find it empirically.

The output layer has as many neurons as the problem needs outputs. In a classification problem, there are as much neurons as the number of classes. For instance, in a text classification problem with 6 different languages, we extract the 'a-z' character frequency to learn from labeled text the language frequency. In this case we would need a neural network with 26 input neurons, each holding the value of a character, and 6 output neurons. In the learning dataset, for each input text there would be a table of 26 input variables, and a table of 6 expected output values.

We have seen in the previous section how to train a single neuron, but it is not directly practicable for a neural network since the error $e(t)$ can only be calculated for the last layer. The learning algorithms for ANN try to transmit a part of current error to each neuron of the layer regarding their contribution to the error. The most common one is the *Gradient Descent Backpropagation* algorithm [12].

Gradient Descent Backpropagation

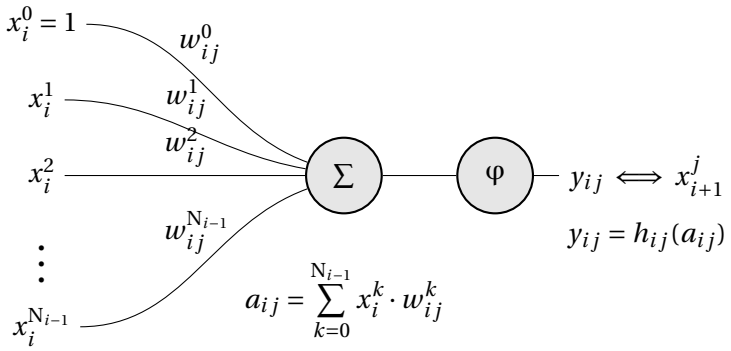
A general neural network representation:



Where:

- $0 \leq i \leq \mathrm{L}$ is the iterator of the current layer, and $\mathrm{L}+1$ is the total number of layers.
- $0 \leq j \leq \mathrm{N}_i$ is the neuron iterator for the current i^{th} layer, and $\mathrm{N}_i + 1$ is the total of neurons in this layer.

In this network, an arbitrary neuron $(i, j), \forall\, i \neq 0, \forall\, j \neq 0$ is as follows:



Where $0 \leq k \leq N_{i-1}$ is the iterator for the weight. It starts at 0 (the index of the bias weight) and stops at N_{i-1} (the neurons number for the previous layer $i-1$). $h(x)$ is a known differentiable transfer function.

Let $E(t) = \frac{1}{2}\sum_{j=1}^{N_L}(e_j - y_{Lj})^2$ be the quadratic error between the expected output $\vec{e}$ and the actual output $\vec{y_L}$. The goal of the training is to minimize this error. The error only depends on synaptic weights since all other variables are known. We can express the error as a function of weights of the entire network, so we can express it as:

$$E(t) = E(\vec{w(t)})$$

With $\vec{w(t)} = (w_{00}^0, w_{00}^1, \ldots, w_{LN_L}^{N_{L-1}})$. At the next step, $t+1$, of the learning algorithm the error must be lower $E(t+1) < E(t) \iff E(\vec{w(t+1)}) - E(\vec{w(t)}) < 0$. Weights at the next iteration $(t+1)$ can be expressed as the weights of the current iteration (t) with a small deviation $\vec{\Delta w(t)}$. We call this deviation, the *direction* of the weights. We set down $\vec{w(t+1)} = \vec{w(t)} + \vec{\Delta w(t)}$. We are looking for a direction $\vec{\Delta w(t)}$ that lowers the error.

$$E(t+1) = E(\vec{w(t)} + \vec{\Delta w(t)})$$

The *Taylor-Young theorem* [13] at order 1 gives us an approximation of $E(t+1)$ around $\vec{w(t)}$:

$$E(t+1) = E(\vec{w(t)}) + \vec{\nabla E(w(t))}^T.\vec{\Delta w(t)} + o(||\vec{\Delta w(t)}||)$$

As a reminder, The *Taylor-Young theorem* for multivariate functions is expressed as follows:

Theorem 4.1.1: Taylor-Young

Let $f:\mathbb{R}^n \mapsto \mathbb{R}$ be a k times differentiable function at the point $a \in \mathbb{R}^n$. f can be expressed such as

$$f(x) = \sum_{i=0}^{k} \frac{d^i f_{\vec{a}}(\vec{(x-a)^i})}{i!} + o(||\vec{(x-a)^k}||)$$

With $\vec{x^k} = (x_1, \ldots, x_n)^k = (x_1, \ldots, x_n, x_1, \ldots, x_n, \ldots, x_1, \ldots, x_n)$. $\vec{x^k}$ has $n.k$ components.

The differential of $E(\vec{\Delta w(t)})$ at the first order is:

$$d^1 E_{\vec{w(t)}}(\vec{\Delta w(t)}) = \vec{\nabla E(w(t))}^T.\vec{\Delta w(t)}$$

With the gradient $\vec{\nabla E(w(t))} = (\frac{\partial E(t)}{\partial w_{00}^0}, \frac{\partial E(t)}{\partial w_{00}^1}, \ldots, \frac{\partial E(t)}{\partial w_{LN_L}^{N_{L-1}}})$

By choosing the direction $\vec{\Delta w(t)} = -\mu \cdot \vec{\nabla E(w(t))}, \mu > 0$, our goal is to construct $E(t+1)$ such as $E(t+1) < E(t)$. We get:

$$E(t+1) \approx E(\vec{w(t)}) - \mu.||\vec{\nabla E(w(t))}||^2$$

As $\mu.||\vec{\nabla E(w(t))}||^2 > 0$, we found that with the chosen direction we get $E(t+1) < E(t)$. Now we need to evaluate each:

$$\Delta w_{ij}^k = -\mu \cdot \frac{\partial E(t)}{\partial w_{ij}^k}$$

This requires to derive the neural network functions in reverse layer order (starting by the last layer). For each Δw_{ij}^k, we need to derive until we get an expression that contains only known variables. First, we calculate the weight deltas for the last layer L:

$$\frac{\partial \mathrm{E}(t)}{\partial w_{\mathrm{L}j}^k} = \frac{\frac{1}{2}\sum_{l=1}^{\mathrm{N_L}}(e_l - y_{\mathrm{L}l})^2}{\partial w_{\mathrm{L}j}^k} = \frac{\partial \frac{1}{2}\cdot(e_j - y_{\mathrm{L}j})^2}{\partial w_{\mathrm{L}j}^k} = -\frac{\partial y_{\mathrm{L}j}}{\partial w_{\mathrm{L}j}^k}\cdot(e_j - y_{\mathrm{L}j})$$

$$\frac{\partial \mathrm{E}(t)}{\partial w_{\mathrm{L}j}^k} = -\frac{\partial h(a_{\mathrm{L}j})}{\partial w_{\mathrm{L}j}^k}\cdot(e_j - y_{\mathrm{L}j}) = -\frac{\partial h(\sum_{l=0}^{\mathrm{N_{L-1}}} x_{\mathrm{L}}^l \cdot w_{\mathrm{L}j}^k)}{\partial w_{\mathrm{L}j}^k}\cdot(e_j - y_{\mathrm{L}j})$$

$$\frac{\partial \mathrm{E}(t)}{\partial w_{\mathrm{L}j}^k} = -x_{\mathrm{L}}^k \cdot h'(a_{\mathrm{L}j})\cdot(e_j - y_{\mathrm{L}j})$$

As x_{L}^k, $h'(a_{\mathrm{L}j})$ and $(e_j - y_{\mathrm{L}j})$ are all known variables, we get:

$$\Delta w_{\mathrm{L}j}^k = \mu \cdot x_{\mathrm{L}}^k . h'(a_{\mathrm{L}j})\cdot(e_j - y_{\mathrm{L}j})$$

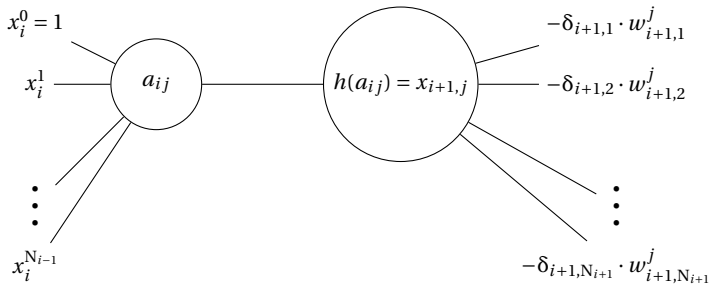
We introduce δ_{ij}, a variable that represents the partial error of the neuron ij. For the last layer ($i = \mathrm{L}$) we set down:

$$\boxed{\delta_{\mathrm{L}j} = h'(a_{\mathrm{L}j})\cdot(e_j - y_{\mathrm{L}j})}$$

The rule of weight modification, for the last layer, becomes:

$$\boxed{w_{\mathrm{L}j}^k(t+1) = w_{\mathrm{L}j}^k(t) + \mu \cdot x_{\mathrm{L}}^k \cdot \delta_{\mathrm{L}j}}$$

For the other layers $0 < i < \mathrm{L}$, we observe, when we derive, that $\frac{\partial \mathrm{E}(t)}{\partial w_{ij}^k}$ is dependent on all $\delta_{i+1,j}$ because the term $y_{ij} \iff x_{i+1,j}$ appears in the derivation of all neurons from the next layer.



$$\frac{\partial \mathrm{E}(t)}{\partial w_{ij}^k} = -\frac{\partial x_{i+1,j}}{\partial w_{ij}^k}\cdot\sum_{l=1}^{\mathrm{N}_{i+1}}\delta_{i+1,l}\cdot w_{i+1,l}^j$$

$$\frac{\partial \mathrm{E}(t)}{\partial w_{ij}^k} = -x_i^k \cdot h'(a_{ij}).\sum_{l=1}^{\mathrm{N}_{i+1}}\delta_{i+1,l}\cdot w_{i+1,l}^j$$

As the terms x_i^k, $h'(a_{ij})$ and $\sum_{l=1}^{\mathrm{N}_{i+1}}\delta_{i+1,l}\cdot w_{i+1,l}^j$ contains only known variables, we write down the final expression:

$$\Delta w_{ij}^k = \mu \cdot x_i^k . h'(a_{ij})\cdot\sum_{l=1}^{\mathrm{N}_{i+1}}\delta_{i+1,l}\cdot w_{i+1,l}^j$$

We define $\delta_{i,j}$ by recurrence for $0 < i < \mathrm{L}$ such as:

$$\delta_{ij} = h'(a_{ij}) \cdot \sum_{l=1}^{N_{i+1}} \delta_{i+1,l} \cdot w_{i+1,l}^{j}$$

Which leads to:

$$w_{ij}^{k}(t+1) = w_{ij}^{k}(t) + \mu \cdot x_i^k \cdot \delta_{ij}(t)$$

Implementation of the Backpropagation Algorithm

The learning algorithm we described is known as *stochastic gradient descent* [12]. For each sample in the dataset, we propagate its input values in the network to obtain the error $E(t)$ between the expected output for this sample and the network output. Then we backpropagate the error and update the weights.

Input Propagation

1. Take a sample from the dataset, with input vector X and expected output Y. We start the processing at the first hidden layer.
2. Process the output value of each neuron of the current layer. The vector that contains all these output values becomes the new X.
3. If there is a next layer, go to it then go to step 2.
4. Process the error vector $E(t) = Y - X$.

Error Backpropagation

1. Calculate the derivative of the transfer function $h(x)$. For the simple sigmoid function, $h(x) = \frac{1}{1+e^{-x}}$, we have:

$$h'(x) = \frac{e^{-x}}{1+e^{-x}} = h(x) \cdot (1 - h(x))$$

2. Calculate the partial errors from the last layer:

$$\delta_{Lj} = h'(a_{Lj}) \cdot (e_j - y_{Lj})$$

3. Calculate recursively all partial errors for the preceding layers:

$$\delta_{ij} = h'(a_{ij}) \cdot \sum_{l=1}^{N_{i+1}} \delta_{i+1,l} \cdot w_{i+1,l}^{j}$$

4. Modify the weights with the formula:

$$w_{ij}^{k}(t+1) = w_{ij}^{k}(t) + \mu \cdot x_i^k \cdot \delta_{ij}$$

Stochastic Gradient Descent Pseudo-Code

```
Until the learning is over, do
  For each couple (x⃗, y_ex), in the learning dataset, do
    # Processing the ANN output
    For each layer i ∈ [0,L], do
      For each neuron j ∈ [0,N_i], do
```

$a_{ij} = \sum_{k=0}^{N_{i-1}} x_i^k \cdot w_{ij}^k$
$y_{ij} = h_{ij}(a_{ij})$
End for
End for

# Processing the current error
For each neuron $j \in [1, N_L]$ of the output layer, do
$e_j(t) = y_{ex,j} - y_j$
End for

# Processing of the partial errors
For each layer $i \in [1, L]$ in *reverse order*, do
If $i == L$
For each neuron $j \in [1, N_L]$, do
$\delta_{Lj}(t) = h'_{Lj}(X_{Lj}) \cdot e_j(t)$
End for
Else
For each neuron $j \in [0, N_i]$, do
$\delta_{ij}(t) = h'_{ij}(X_{ij}) \cdot \sum_{l=0}^{N_{i+1}} \delta_{i+1l}(t) \cdot w_{il}^j(t)$
End for
End if
End for

# Update of the weights
For each $i \in [0, L]$, $j \in [0, N_i]$, $k \in [0, N_{i-1}]$, do
$w_{ij}^k(t+1) = w_{ij}^k(t) + \mu \cdot x_i^k \cdot \delta_{ij}(t)$
End for

learning is *over* if a *stopping condition* has been *reached*
End for
End until

4.2 Problematics of Neural Network Application

When setting up a neural network, one realizes that the algorithms presented in the previous section are not sufficient to obtain a functional artificial intelligence. Here we will see the problems of implementation of neural networks that we have solved and implemented in *libmla*. In the first part we see the problem of the saturation of the weights of the network, which prevents the network to learn or leads to overfitting. In a second part we deal with the problem of overfitting & overtraining, where the neural network is perfectly capable of classifying training data but has very low performance on unknown data. Next, we discuss the initialization of the neural network that plays an important role in the convergence of learning. In this part we contribute to the state-of-the-art by proposing a simple method of initialization which gives good results. Finally, we expose implementation tricks, necessary to obtain a competitive speed of execution.

4.2.1 Saturation

The saturation effect appears very quickly on *Multi-Layer Perceptrons* (*MLPs*). During training, the weights take extreme values. This has several consequences. With a sigmoid transfer function, there is no intermediate value between the extreme output values (-1 and 1). There is therefore a loss of information, and a lower capacity to classify or approximate unknown entries because the network lacks flexibility. In addition, saturation makes it impossible to go back intermediate values. If it turns out that it is more optimal for the weight to go from positive to negative (or vice versa), this is no longer possible. If a majority of the weights of the neural network is in a saturated state, the network no longer learns and is stuck in that state. To solve this problem, two strategies are mainly used.

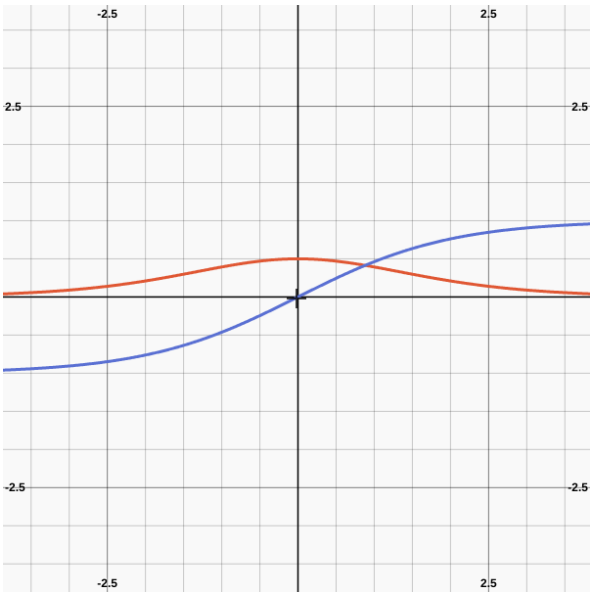


FIGURE 4.13 – The sigmoid (in blue) and its derivative (red)

Clipping the Derivatives

The first strategy, that is easy to set up, is the clipping of the derivative transfer function. In Figure 4.13, we show the sigmoid function and its derivative.

The derivative tends to zero for hightly negative or positive values. If a neuron is saturated, it exhibits extreme activation values, thus the derivative is near-zero. In fact, the more saturated the neuron, the less likely it is to return to a flexible state. By clipping the derivative for high values, the system keeps the flexibility to correct saturated units. In this case, the derivative does not correspond to the mathematical derivative. However the derivative has the correct sign, so the gradient direction is approximately the same. We implemented this strategy in *libmla*:

```
float neuralNetwork::sigmoidDerivative(float x) {
  static float constexpr clampLimitPos = 4.0f, clampLimitNeg = -4.0f;
  static float const clampedValue = ((-2.0f) * SIGMOID_FACTOR * mla::exp(clampLimitPos)) /
                                    (std::pow(mla::exp(clampLimitPos) + 1.0f, 2.0f));

  if (x < clampLimitNeg || x > clampLimitPos) {
    return clampedValue;
  } else {
    x = mla::exp(SIGMOID_FACTOR * x);
    return ((-2.0f) * SIGMOID_FACTOR * x) / (std::pow(x + 1.0f, 2));
  }
}
```

L1 and L2 regularization

Another strategy is the penalization of the high weights of the network. *L1* and *L2* regularization [2] are commonly used to penalize high weights. The principle is to add the norm (*L1* or *L2*) of the weights to the error function.

$$\text{(L1):E}(t) = \text{E}(\overrightarrow{w(t)}) + \beta \cdot \sum_i |w_i| \qquad \text{(L2):E}(t) = \text{E}(\overrightarrow{w(t)}) + \beta \cdot \frac{1}{2} \sum_i w_i^2$$

With β, a known real positive. To know the new formula of the weight modification, we derive again the functions of the neural network. It is quite simple finally, since from the point of view of a specific weight w_i, it is about deriving the terms:

$$\text{(L1):}\beta \cdot |w_i| + c \qquad \text{(L2):}\beta \cdot \frac{1}{2} w_i^2 + c$$

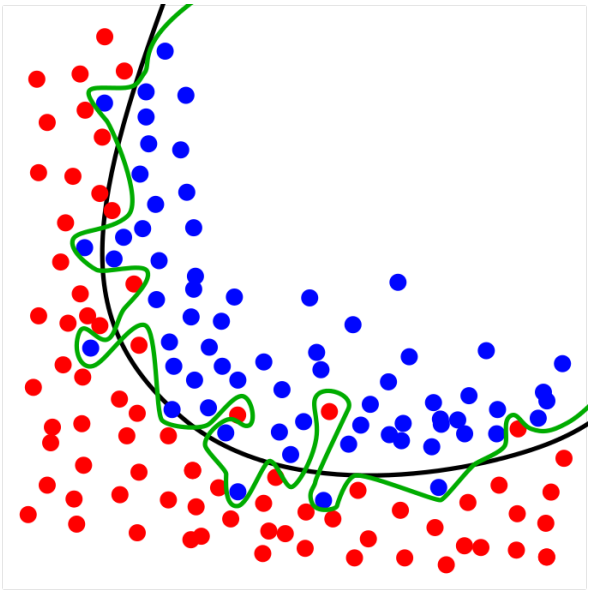


FIGURE 4.14 – Overfitting of training data. In green an overfitting classifier, in black a balanced classifier

With c, a constant. It holds since the weights are interdependent variables. The new weight modification formula becomes:

$$w_{ij}^k(t+1) = w_{ij}^k(t) - \mu \cdot \frac{\partial E(t)}{\partial w_{ij}^k}$$

$$\boxed{(\text{L1}): w_{ij}^k(t+1) = w_{ij}^k(t) - \mu \cdot \beta . sign(w_{ij}^k(t)) + \mu \cdot x_i^k \cdot \delta_{ij}(t)}$$

$$(\text{L2}): w_{ij}^k(t+1) = w_{ij}^k(t) - \mu \cdot \beta \cdot w_{ij}^k(t) + \mu \cdot x_i^k \cdot \delta_{ij}(t)$$

$$\boxed{(\text{L2}): w_{ij}^k(t+1) = (1 - \mu \cdot \beta) w_{ij}^k(t) + \mu \cdot x_i^k \cdot \delta_{ij}(t)}$$

Both L1 and L2 regularization drag the weight to zero — and in fact they are also called L1 and L2 *weight decay*. L1 does it by adding a constant term with the opposite sign of the weight and L2 does it by multiplying the weight by a constant between 0 and 1. Thus, at each iteration of the training algorithm, the absolute value of the weights are reduced, preventing saturation. β becomes a new hyperparameter of the network, that needs to be adjusted for the problem to solve. Since the *weight decay* of L2 is proportional to the weight, a fixed β value fit a broader range of problems. It is why we implemented L2 *weight decay* in *libmla*, instead of L1.

4.2.2 Overfitting

Overfitting is the empirical observation that an approximation or classification algorithm uses a model that is too complex to predict the objective function [1]. Overfitting appears when the gap between the error on the training data and the error on new samples is high. Figure 4.14 (public image, author Ignacio Icke) presents the difference of an overfitting classifier that has a model too complex and a balanced classifier that gives a simpler model that is more likely to be able to generalize on new data.

Overfitting in the neural networks comes mainly from 3 causes: the saturation of the neurons, an architecture too complex for the problem to be treated and the overtraining. The first case has been treated in the previous part. To the contrary, *underfitting* appears when the error on the

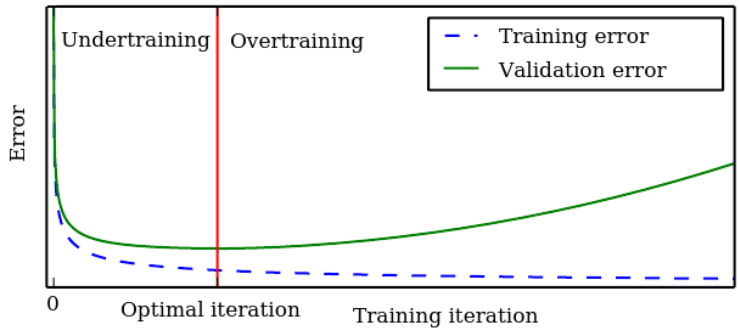


FIGURE 4.15 – The optimal iteration of the training process

training set is too high. The causes are the perfect opposites of the overfitting: near-zero weights, an architecture too simple and undertraining. Underfitting is easy to detect since the error on the training set stagnates, the model is not able to learn. We do not study in depth underfitting here as the cases we encountered it were rare and the causes, obvious — an hyperparameter with extreme value, or problems of feature set normalization.

The complexity of the architecture on *MLPs* is adjusted with the amount of hidden units. It is unreal to give a definite formula to get perfect number of hidden neuron, as the perfect one is very dependent on the feature size, the sample number, the problem complexity and other hyperparameters. A rule of thumb is that a good number of hidden units is comprised between the number of output units and the number of input units. In *libmla*, when this hyperparameter is not specified we set it to $\sqrt{n}$ (and $2\sqrt{n}$ when their is *Dropout*, see later) where n is the number of input units. We found, empirically, that it gives good results for most problems. However, it is ultimately another hyperparameter of the network to optimize.

Cross Validation

Overtraining is the observation that at one point, iterating again on the training algorithm increases the error on new data. To have a better understanding of this phenomenon, we introduce two concepts of a neural network:

- **Knowledge**. It is the ability of a neural network to remember (or to fit to) training data. In other words, the more knowledge the system has, the more likely it gives output values that fit to training data. In our case, knowledge is deduced from the error on the training data. The lower the error, the more the system has knowledge.
- **Ability of Generalization**. It is the ability of the system to give good results for an unknown input vector. It is measured by the error on samples that are not part of the training set.

We want a neural network with the best *ability of generalization*. At a point in the learning process, these two concepts are in conflict. In order to measure the ability of generalization, we test the performances of the network regularly on new data during the learning process. This technique is called *Cross Validation*. A fraction of the dataset (usually 20%) is reserved for the validation. At each iteration, the network error on the validation set is measured. Theoretically we obtain the type of curves presented in Figure 4.15 (source [2]).

In order to obtain the best network surely, simply save the best network state (i.e. the network that has the lowest error on the validation set) during the training, and load this state at the end of the training. The end of the training being given in this case by a maximum iteration of the algorithm.

Another approach, called *early stopping*, takes advantage of the existence of an optimal iteration to stop the training when it is reached. From the theoretical curves, it seems easy to implement *early stopping*. When the error on the validation set is above the one from the previous iteration, stop the training. Unfortunately, actual error curves from real problems look like the ones pictured

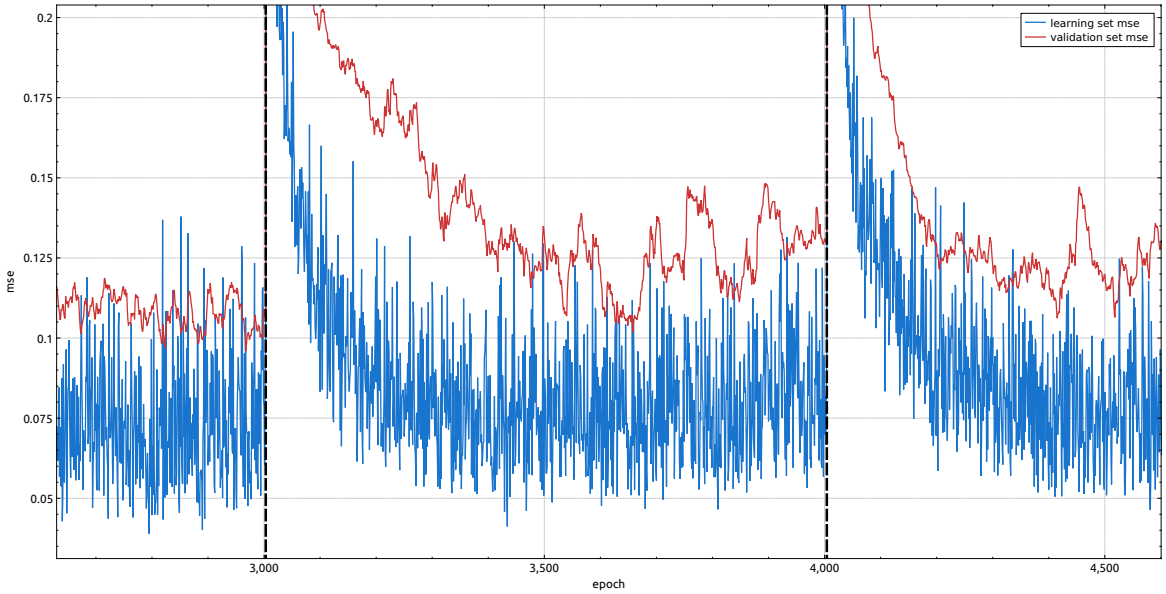


FIGURE 4.16 – Actual training and validation error curves (very noisy)

in Figure 4.16. It shows the training of the third fold of a neural network with the *ADAM* [14] learning algorithm, *stochastic weight update*, *dropout* [15] and *L2 regularization*. These settings tend to produce noisy curves.

In these cases, *early stopping* is not possible to apply, as it is extremely hard to set up a rule that stops at the optimal operation. On cases with settings that produce less noisy curves (Figure 4.17) — same configurations without *dropout* and *L2 regularization*. By averaging the previous values of the validation curve, and comparing the previous averaged value with the current one, it is possible to implement effectively the *early stopping*. Several strategies can be adopted to cope with noise — like waiting that the curve is n time increasing —, but ultimately no strategy can determine the optimal iteration for the first training in Figure 4.17. *Early stopping* is a great theoretical tool but a dangerous one for actual problems. It is implemented in *libmla*, but we realized it does more harm than it helps. Hence, we do not use it.

Dropout

Dropout [15] is a technique that addresses the problem of overfitting and that speeds up the training process. This works by randomly disabling input layer and hidden layer neurons at each iteration of the training algorithm (Figure 4.18, source [15]). It prevents neurons from co-adapting too much. The initial paper [15] shows that dropout is equivalent to training 2^n thinned network from a neural network of n units. By averaging the prediction of these 2^n thinned networks, dropout achieve the regularization of the network, reducing overfitting. As a byproduct of the neuron disabling, the network is quicker to train.

The paper study the optimal proportion of neurons to be disabled. We followed the author recommendations and implemented in *libmla* 80% random disabling for the input layer and 50% random disabling for the hidden layer. It tends to make the training and validation curves very noisy (see Figure 4.16 where there is dropout compared to Figure 4.17 without it). However the resulting accuracy is really enhanced for most of problems. The network from Figure 4.16 achieve 98% accuracy on a spam detection problem with *10-fold cross validation* [16], whereas the network from Figure 4.16 achieved 81.6% accuracy on the same problem. Dropout implementation has interdependencies with the implementation of other learning algorithms (ex: *ADAM*), that are not taken into account in Machine Learning libraries. Because some units are disabled during the training, the decay of momentums from the *ADAM* algorithm should be different for each neuron.

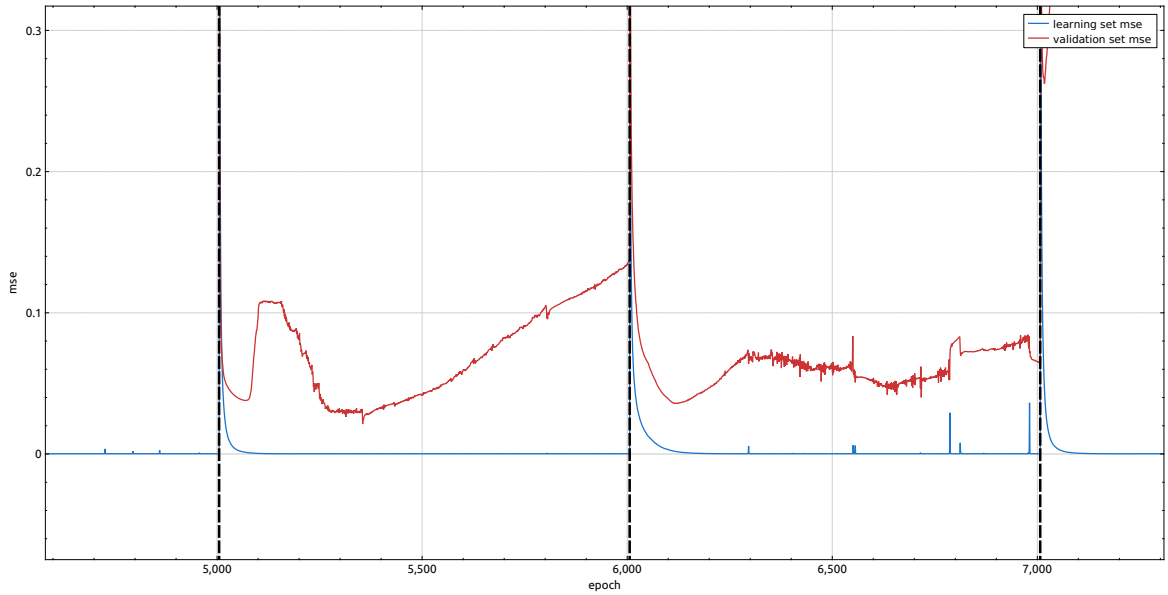


FIGURE 4.17 – Actual training and validation error curves (somewhat noisy)

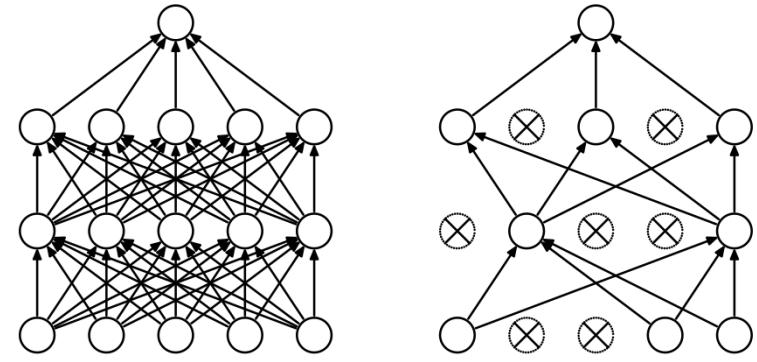


FIGURE 4.18 – Neural network with and without dropout

4.2.3 Initialization

The initialization of neural networks has received little attentions in the recent years, and yet it is a necessary step in establishing effective training. The network weight matrix is randomly initialized. If by pure chance, this weight matrix is close to the result of a trained network of neurons obtaining good results, there is no training to be done. If the weight matrix is totally distant from the one of a trained neural network, in the best case the training will take an additional time for this matrix to find acceptable values, in the worst case the network will not converge. The initialization is that much of importance. The question now is: what control can we have on initialization? It is obvious that if there was a method to initialize the weights in such a way that the network had good results from the first iteration; there would be no need for a training algorithm. On the other hand, it is possible to control the randomness in such a way that the weights are in a disposition which allows a fast converge of the training.

Revisiting the Nguyen-Widow Method: the Distributed Bias Technique

The first compelling work on neural network initialization is the *Nguyen-Widow* method [17]. The key idea is that the first hidden layer must start to cover the input space evenly. The first step is to randomize uniformly the w_{i1}^k weights between -1 and 1. The second step is adjusting the magnitude of the $\overrightarrow{\mathrm{W}_{i1}}$ vectors — i.e. the weight vector of each hidden neuron. The authors consider that the magnitude of weight vector should be relative to the size of the input space N. Let assume there are H hidden units. The goal is to make each hidden unit responsible for a slice of I/H size from the interval I of each input variable. The weight vector norms are then adjusted according to the formula:

$$|\overrightarrow{\mathrm{W}_{i1}}| = 0.7 \cdot \mathrm{H}^{\frac{1}{\mathrm{N}}}$$

The 0.7 coefficient makes a slight interval interleaving between slices. Last, the bias of each neuron serves the purpose of moving the center of the weight distribution in order to cover the input space:

$$s_i = \text{ uniform random number between } -|\overrightarrow{\mathrm{W}_{i1}}| \text{ and } |\overrightarrow{\mathrm{W}_{i1}}|$$

This method definitely enhanced the results of neural networks for all kinds of problems, as the authors shown in their paper and as we found in our experiments (see last chapter). However, it is unclear why this method works because the justification for the weight vector magnitude formula seems very vague. In the article, the neural networks used and the input space have low dimensionality. These numbers are very distant from the problem we are trying to solve and from most of problems solved by Machine Learning. By studying with a closer look their formula, we observe that for most of real problems the magnitude is 1. In Figure 4.19 we show 3 cases:

- Increasing of both H and N (same value).
- Setting the number of hidden neurons H to 100 and increasing N.
- Setting N to 100 and increasing H.

Whatever be the case, the magnitude quickly and strongly tends to 1.

In other words, what the *Nguyen-Widow* is actually doing is normalizing the weight vectors — which happens to hold nice mathematical properties for the neural network, see next part — and distributing the bias evenly along the $[-0.7, 0.7]$ interval. We can thus summarize this method by renaming it the *Distributed Bias Technique* that is really responsible for the performance enhancement. In fact, we see in the next part that there is an option far better than the normalization of the weight vectors.

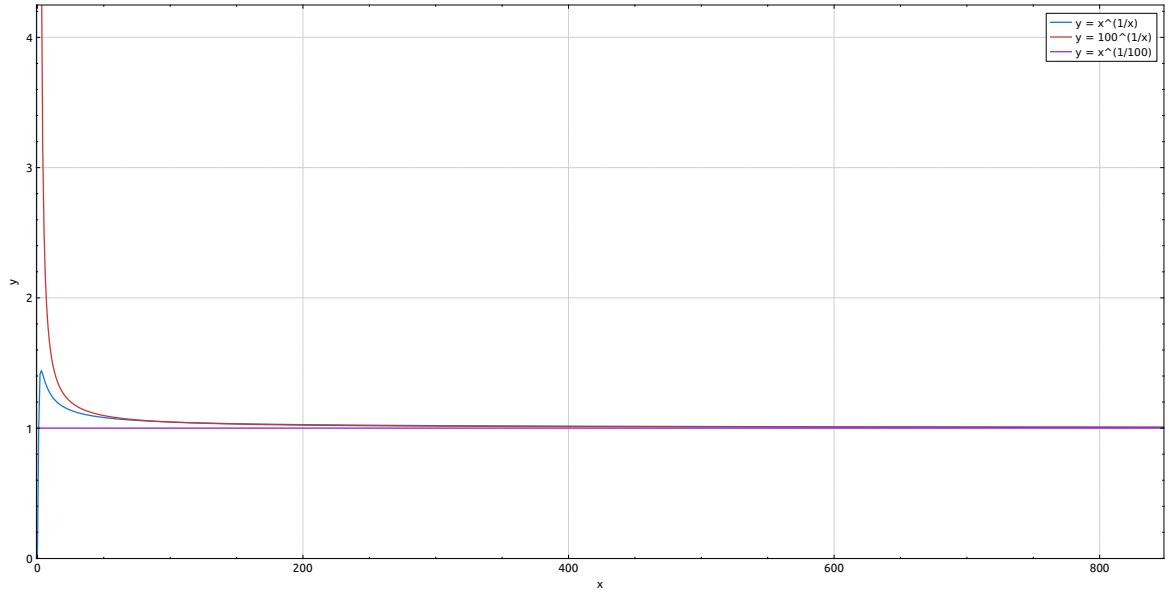


FIGURE 4.19 – Weight magnitude of the Nguyen-Widow method with realistic values

Controlled Variance of Initial Neuron Activations

Normalizing the initial weight vectors gives the guarantee that the initial activation levels are bounded. Indeed:

$$a_{i1} = \overrightarrow{W_{i1}} \cdot \overrightarrow{X} = \sum_{k}^{n} w_{i1}^{k} \cdot x_k$$

$$-|\overrightarrow{W_{i1}}| \cdot |\overrightarrow{X}| \leq a_{i1} \leq |\overrightarrow{W_{i1}}| \cdot |\overrightarrow{X}|$$

With a_{i1} the activation level of the neuron i of the first hidden layer, n the size of the input layer, $\overrightarrow{W_{i1}} = (w_{i1}^0, \ldots, w_{i1}^n)$ the weight vector of neuron $(i,1)$, $\overrightarrow{X} = (x_0, \ldots, x_n)$ an input vector. If we normalize $\overrightarrow{W_{i1}}$, a_{i1} is bounded by $|\overrightarrow{X}|$.

The *sigmoid* transfer function has a range of intermediate values in the range $[-1,1]$. This range is dependent on the constant used for the sigmoid, but for neural networks it is usually the range of intermediate values. Outside of this range, the function saturates and the neuron is likely to be in a saturated state. It is important that the initialization procedure let most of the neuron in a flexible state, it is as well important that this procedure produces initial activation levels that cover a wide range of the sigmoid transfer function — as shown in the previous part.

Boundaries on a_{i1} are not enough, as we know nothing about its distribution. We want to control its variance as well. We start by modeling each variable of the input vector X with $U(0,1)$ the uniform random distribution with minimal and maximal values 0 and 1. As a recall:

$$E(U(0,1)) = \frac{1}{2}$$

$$V(U(0,1)) = \frac{1}{12}$$

With $E(X)$ and $V(X)$ respectively the mean and variance of the random variable X. We want to control the underlying distribution $U(-b,b)$ of W_{i1}^k such that the a_{i1} variance is defined.

$$E(U(-b,b)) = 0$$

$$V(U(-b,b)) = \frac{b^2}{3}$$

We note that all variables involved are independent. As a recall, for two independent random variables ([18], p26):

$$\mathrm{E}(\mathrm{X}+\mathrm{Y}) = \mathrm{E}(\mathrm{X}) + \mathrm{E}(\mathrm{Y})$$

$$\mathrm{E}(\mathrm{X}\cdot\mathrm{Y}) = \mathrm{E}(\mathrm{X})\cdot\mathrm{E}(\mathrm{Y})$$

$$\mathrm{V}(\mathrm{X}+\mathrm{Y}) = \mathrm{V}(\mathrm{X}) + \mathrm{V}(\mathrm{Y})$$

$$\mathrm{V}(\mathrm{X}\cdot\mathrm{Y}) = \mathrm{V}(\mathrm{X})\cdot\mathrm{V}(\mathrm{Y}) + \mathrm{V}(\mathrm{X})\cdot\mathrm{E}(\mathrm{Y})^2 + \mathrm{V}(\mathrm{Y})\cdot\mathrm{E}(\mathrm{X})^2$$

Clearly, $\mathrm{E}(a_{i1}) = 0$, as each of the weight has a mean of 0. For $\mathrm{V}(a_{i1})$, we start by calculating all $\mathrm{V}(x_k \cdot w_{i1}^k)$.

$$\mathrm{V}(x_k \cdot w_{i1}^k) = \mathrm{V}(x_k)\cdot\mathrm{V}(w_{i1}^k) + \mathrm{V}(x_k)\cdot\mathrm{E}(w_{i1}^k)^2 + \mathrm{V}(w_{i1}^k)\cdot\mathrm{E}(x_k)^2$$

$$\mathrm{V}(x_k \cdot w_{i1}^k) = \frac{1}{12}\cdot\frac{b^2}{3} + \frac{1}{12}\cdot 0 + \frac{b^2}{3}\cdot\frac{1}{4} = \frac{b^2}{9}$$

Hence,

$$\mathrm{V}(a_{i1}) = \sum_{k=1}^{n} \mathrm{V}(x_k \cdot w_{i1}^k)$$

$$\mathrm{V}(a_{i1}) = \frac{n\cdot b^2}{9}$$

By the *Law of Large Numbers*, the activation levels a_{i1} are distributed by a normal law $\mathrm{N}\left(0, \frac{n\cdot b^2}{9}\right)$. Meaning the more the input size increase, the higher the variance. Hence, neurons will likely to start in a saturated state. We want to control the variance of the activation, so we set the normal law of the activation levels to $\mathrm{N}(0,1)$. Thus:

$$\boxed{b = \frac{3}{\sqrt{n}}}$$

We started from the assumption that the variables of the input follow a uniform law $\mathrm{U}(0,1)$. This assumption is likely to be false. Instead of analyzing the distribution of each of the variables — which at least a tedious task, and it is probably not possible to draw firm conclusions —, we take the hypothesis that, each feature follows its own distribution law. At the start of this study, input vectors had variables that follow the $\mathrm{U}(0,1)$ law, so their norm would be on average:

$$|\vec{\mathrm{X}}| = \sqrt{\sum_{k=1}^{n} \mathrm{E}(\mathrm{U}(0,1)^2)}$$

The mean of the squared uniform law is given by:

$$\mathrm{E}(\mathrm{U}(0,1)^2) = \int_0^1 u^2 \cdot f_{\mathrm{U}}(u)\cdot du = \int_0^1 u^2\cdot 1\cdot du = \frac{1}{3}$$

Hence,

$$|\vec{\mathrm{X}}| = \sqrt{\sum_{k=1}^{n} \frac{1}{3}}$$

$$|\vec{\mathrm{X}}| = \sqrt{\frac{n}{3}}$$

We can then normalize the feature vectors in order they have a $\sqrt{\frac{n}{3}}$ norm, averagely. At this point, knowing nothing about the feature distributions, the only control we have is over the norm

of the feature vectors. Nevertheless, for convenience it is better to scale the weights of the network and keep the features pristine. Let d be the average norm of the input vectors.

$$\mathrm{V}\left(\sqrt{\frac{n}{3.d^2}}\cdot\overrightarrow{\mathrm{W}_{i1}}\cdot\overrightarrow{\mathrm{X}}\right)=\frac{1}{27}\cdot(\frac{n.b}{d})^2$$

hence,

$$\boxed{b=\sqrt{27}\cdot\frac{d}{n}}$$

So in the case where we can compute the average norm of feature vectors, the weights of the first hidden layer should be initialized by $\mathrm{U}\left(-\sqrt{27}\cdot\frac{d}{n},\sqrt{27}\cdot\frac{d}{n}\right)$. If we cannot compute the average norm of feature vectors, the first hidden layer should be initialized with $\mathrm{U}\left(-\frac{3}{\sqrt{n}},\frac{3}{\sqrt{n}}\right)$.

Let t be the number of hidden units. We are looking at a similar method to initialize the weights of the output neurons. We have t normal laws $\mathrm{N}(0,1)$ that passe through symmetric sigmoid tranfert functions:

$$h(x)=\frac{2}{1+e^x}-1$$

Determining $\mathrm{V}(a_{i\mathrm{L}})$, the variances of the activation levels for the last layer L is a challenging task, since we need to integrate $h(\mathrm{N}(0,1))$ and $h(\mathrm{N}(0,1))^2$. We opted for running a *Monte-Carlo* simulation with 100,000,000 samples. It could be interesting to get the theoretical variance, but simulation is far enough for our purpose. We found empirically that:

$$\mathrm{E}(h(\mathrm{N}(0,1)))=3.5675656302539516e-05$$

$$\mathrm{V}(h(\mathrm{N}(0,1)))=0.1736046409702093$$

The normal law $\mathrm{N}(0,1)$, produces on average the same amount of positive and negative numbers and these numbers have on average the same absolute value. The simoid we use, $h(x)=\frac{2}{1+e^{-x}}-1$, has a central symmetry around 0, $h(-x)=-h(x)$. With the normal law, for each $h(a)$, there will be an $h(-a)$. As $h(a)+h(-a)=0$, the mean of this sigmoid-normal law is 0. The mean we empirically found is very close to 0, so we have a good confidence about the value of $\mathrm{V}(h(\mathrm{N}(0,1)))$. Let $\mathrm{U}(-z,z)$ be the uniform distribution that initializes the weights of the output layer. As there is t hidden units, the variance of the activation levels is approximately:

$$\mathrm{V}(a_{i\mathrm{L}})=\sum_{k=1}^{t}\mathrm{V}(w_{i\mathrm{L}}^k\cdot x_k)$$

$$\mathrm{V}(a_{i\mathrm{L}})=\sum_{k=1}^{t}(0.1736\cdot\mathrm{V}(w_{i\mathrm{L}}^k)+0.1736\cdot 0+\mathrm{V}(w_{i\mathrm{L}}^k)\cdot 0)$$

As $\mathrm{V}(\mathrm{U}(-z,z))=\frac{z^2}{3}$,

$$\mathrm{V}(a_{i\mathrm{L}})=0.0578\cdot z^2\cdot t$$

$$\boxed{z=\sqrt{\frac{1}{0.0578\cdot t}}}$$

Finally, we should initialize the output layer weights with the $\mathrm{U}\left(-\sqrt{\frac{1}{0.0578\cdot t}},\sqrt{\frac{1}{0.0578\cdot t}}\right)$ distribution, where t is the number of hidden units.

Last, it is common to use dropout. It sligtly modifies the formulas as a fraction only of the input neurons and hidden neurons are active at any time. Let r_i and r_h be respectively the dropout rate

TABLE 4.1 – Uniform laws to use for a neural network initialization

Case	no dropout no average norm	no dropout average norm
Hidden layer	$\mathrm{U}\left(-\frac{3}{\sqrt{n}}, \frac{3}{\sqrt{n}}\right)$	$\mathrm{U}\left(-\sqrt{27}\cdot\frac{d}{n}, \sqrt{27}\cdot\frac{d}{n}\right)$
Outpout layer	$\mathrm{U}\left(-\sqrt{\frac{1}{0.0578\cdot t}}, \sqrt{\frac{1}{0.0578\cdot t}}\right)$	$\mathrm{U}\left(-\sqrt{\frac{1}{0.0578\cdot t}}, \sqrt{\frac{1}{0.0578\cdot t}}\right)$

Case	dropout no average norm	dropout average norm
Hidden layer	$\mathrm{U}\left(-\frac{3}{\sqrt{n\cdot(1-r_i)}}, \frac{3}{\sqrt{n\cdot(1-r_i)}}\right)$	$\mathrm{U}\left(-\sqrt{\frac{27}{1-r_i}}\cdot\frac{d}{n}, \sqrt{\frac{27}{1-r_i}}\cdot\frac{d}{n}\right)$ [2]
Outpout layer	$\mathrm{U}\left(-\sqrt{\frac{1}{0.0578\cdot t\cdot(1-r_h)}}, \sqrt{\frac{1}{0.0578\cdot t\cdot(1-r_h)}}\right)$	$\mathrm{U}\left(-\sqrt{\frac{1}{0.0578\cdot t\cdot(1-r_h)}}, \sqrt{\frac{1}{0.0578\cdot t\cdot(1-r_h)}}\right)$

at the input and hidden layer — then, $(1-r_i)$ and $(1-r_h)$ are the rates of active neurons. Table 4.1 summarize the uniform laws we should use to initialize the neural network in each cases.

4.2.4 Implementation Tricks

In order to achieve a competitive implementation, it is necessary to implement low level optimizations. In this part we will discuss the detail of the floating point calculation, the mechanisms of the processor to make the most of its computing power as well as the approximation of the mathematical functions that are at the basis of neural network processing.

Floating point operations

Between two integers, there is an infinity of floating numbers, yet we represent them on 32 or 64 bits. There is therefore inevitably a trade-off between an abstract real number and its machine representation. The floating representation defined by the IEEE 754 [19] provides a number of mechanisms to overcome the technical limitations of this representation. These mechanisms cause a slowdown of the performance of the floating point processing. To name just a few of these mechanisms:

- **Infinities & signed zero**. Infinities can be positive or negative and are the result an overflow or a division by zero. Moreover the zero value is signed. When a small number is rounded and become zero, it keeps its sign. If a number is divided by $-\infty$ it becomes -0. They are the usual cases when a number becomes -0. Except for some operation involving -0 itself or cases like $-n\cdot 0$, zero value are positive. Processing these special cases require checks and particular procedures.
- **NaN**. NaN (Not a Number) is an invalid state of the floating point. They are trigered by invalid operation like $\infty-\infty$, $ln(-1)$, $\sqrt{-1}$. Again, this special case requires a special processing.
- **Denormal number**. Also known as subnormal number. Regular floating points are normalized, meaning the exponent field is non-zero. The mantissa, plus an exponent at 1 serve to represent all floats from 2 to the last valid float before zero, *FLT_MIN*. When the exponent becomes zero, by the result of a division for instance, the mantissa serves to represent the numbers between *FLT_MIN* and zero. The processing of denormal number is usually slower than normal number.

With neural networks, it is common to have vanishing gradients or weights that tend to zero. Denormals will appear, it is an expected behavior. It is not a problem to remove the handling of denormals in the case where the network is well parametrized for the problem. Instead of being denormal, the number will be zeroed. However, if most weights or gradients are denormal numbers it means that the parameters of the network have a wrong order of magnitude. Either the

2. Considering that we do not change d value according with the ratio of activated units

initialization is bad, the learning coefficient or the regularization parameters are badly calibrated. Infinities and NaN should also not appear, if they do, it is mostly due to saturation. Disabling them speed up processing, but in an initial phase, all float checks should be enabled, to monitor signals of underlying problems. In *libmla*, we enable the following *GCC* optimizations regarding floating point calculation:

- *-fno-signaling-nans*. Do not trigger exceptions when a NaN is encountered.
- *-fno-trapping-math*. Compile code without math trapping routines (division by zero, underflow, overflow, inexact result and invalid operation).
- *-ffinite-math-only*. Assume that the arguments of operations are not NaN of infinities, allowing room for optimizations.
- *-fno-math-errno*. Do not set errno when executing single instruction functions (like sqrt).
- *-fcx-limited-range*. Do not reduce the range when performing complex division.
- *-fno-signed-zeros*. Ignore the sign of zeroes. In some cases, signed zeroes prevent simplification of expressions.
- *-freciprocal-math*. Allow the replacement the reciprocal of a value to be used instead of dividing. It can lead to optimizations with subexpression elimination.
- *-fassociative-math*. Allow the reordering of math operations for further optimization. Reordering can change the result of an operation because of the rounding errors.

All these options are enabled with the *-ffast-math* and *-funsafe-math-optimizations* flags. Denormals are disabled by setting the *FTZ* (Flush To Zero) or *DAZ* (Denormal Are Zero) CPU flags. *-ffast-math* may enable these flags, but there is no guarantee. In the initialization of the neural network, we use:

```
#ifdef NDEBUB
  _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
  _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
  unsigned csr = __builtin_ia32_stmxcsr();
  csr |= (1 << 15);
  __builtin_ia32_ldmxcsr(csr);
#endif
```

To make sure *FTZ* and *DAZ* flags are enabled.

Vectorization with Template Metaprogramming

Machine Learning, and particularly neural networks, often involves operations on vectors and matrices. For ease of use, clarity and code maintainability it is preferable to work with dedicated classes that handle vectors and matrices, along with their usual functions — norm, normalization, scalar product, matrices product, etc. It avoids writing *for* loops, endlessly. Unfortunately, working with objects breaks the performance in quite a tragic way for two reasons. The first one is the creation of intermediate objects during operations. Ex:

$$M = M_1 + \sqrt{2.M_2}$$

Where M, M_1 and M_2 are matrices. This operation produces 3 intermediate matrices that store the results of the different operations before assigning them to M. The second reason is linked with the first one. The chain of operations is broken by these intermediate objects, disabling a large number of possible optimizations from the compiler and the CPU.

Template Metaprogramming is the development of full recursive classes with *C++ templates*. *Templates* is the evolution of preprocessor macros. Templates are also preprocessor directives, but they are specialized in the creation of classes and functions. The main difference between a macro

and a template, is that a template can call another template. It opens the field of possibilities, enabling the creation for instance of a class that can deal with any primitive type. It is possible with Template Metaprogramming to make the compiler transforming:

```
m = m1 + sqrt(2*m2);
```

Into:

```
for (i = 0; i < m.size(); ++i)
  for (j = 0; j < m.size(); ++j)
    m[i][j] = m1[i][j] + sqrt(2*m2[i][j]);
```

No intermediate objects are used, the loops are kept very simple enabling compiler and CPU optimizations to take place. We implemented this technique in *libmla*.

Two strategies can be taken when writing efficient and effective code. We can make micro optimizations everywhere, retailed for the specific CPU architecture or we can write code that is in a good disposition to take advantage of compiler and CPU optimizations. Nowadays, it is very hard to beat compiler optimizations (that targets our specific architecture) with manual code optimizations — beyond the scope of algorithmic and structural optimizations, of course. Hence, we chose the second strategy for *libmla*. This way, we profit from the continuous improvement of compilers. In *libmla*, we use the *-O3* and *-march=native* flags, enabling aggressive loop and vectorization optimizations targeting the current CPU architecture, especially the use of *SIMD*[3].

Fast Approximation of Mathematical Primitives

The work in this part has been inspired by the paper *A Fast, Compact Approximation of the Exponential Function* [20]. The author manipulates the floating point representation of *double* precision (IEEE-754) to model the exponential function. This approximation competes in terms of error with table lookup approximation, but it is faster and does not need to store any data. We reused its method to compute an approximation of the exponential function for 32-bits floating point precision. In fact, the *libmla* neural network implementation is entirely in 32-bits floating point precision to take advantage of *SIMD*. Last, we deduced from the exponential approximation, the logarithm / square root / inverse square root / power functions.

32-bits floating point numbers are represented in the $(-1)^s(1+m)2^{x-x_0}$ form, where s is the sign bit (1 bit), m the mantissa (a binary fraction in the range $[0,1)$, represented on 23 bits) and x the exponent (on 8 bits), shifted by a constant bias x_0 (of value 127). The format can be manipulated by accessing the memory as a 32-bits integer. In particular, operations done on the exponent part (x) are exponentiated.

Let assume that we want to exponentiate y. Let u be a union of types {integer (noted i), float (noted f)}. y must be left-shifted by 23 bits after the bias x_0 has been added. $u.i = 2^{23}(y+127)$ computes 2^y for an y integer. If y is a floating point, its fractional part resides in the fractional part of $u.f$. This behavior is highly desirable because the fractional part (m) amounts to a linear interpolation between neighboring integer exponents. Therefore, this technique exponentiates floating points as well as a lookup table of 2^8 entries with linear interpolation. To compute e^y:

$$e^y = 2^{y'}$$
$$y = ln(2^{y'})$$
$$y' = \frac{y}{ln(2)}$$

Hence, y must be divided by $ln(2)$ first. The exponential function approximation is given by the formula:

3. https://www.moreno.marzolla.name/teaching/high-performance-computing/2017-2018/L08-SIMD.pdf

```
float pow(float x, float y) {
 union { float f; int i; } u = { .f = x };
 u.i = (u.i - 1064878240.0f) * y + 1064878240.0f;
 return u.f;
}

float ln(float x) {
 union { float f; int i; } u = { .f = x };
 return u.i * 8.262958294867817e-08f - 87.99044486232242f;
}

float exp(float x) {
 #pragma GCC diagnostic push
 #pragma GCC diagnostic ignored "-Wnarrowing"

 union { float f; int i; } u = { .i = 12102203.161561485f * x + 1064878240.0f };
 return u.f;

 #pragma GCC diagnostic pop
}

float sqrt(float x) {
 union { float f; int i; } u = { .f = x };
 u.i = (u.i >> 1) + 532439120.0f;
 return u.f;
}

float rsqrt(float x) {
 union { float f; int i; } u = { .f = x };
 u.i = 1597317360.0f - (u.i >> 1);
 return u.f;
}
```

FIGURE 4.20 – Fast approximation of mathematical primitives

$$u.i := a.y + b - c$$

Where $a = \frac{2^{23}}{ln(2)}$, $b = 127.2^{23}$ and c is an adjustment parameter that serves to adjust the overall error between the approximation and the native function. In our case, values at the input of the sigmoid function (and therefore the exponential function) usually are in the range $[-2,-2]$, so we optimized c in order to minimize the squared error between *std::exp(x)* and its approximation *mla::exp(x)*. After testing values from 0 to 1,000,000 we found that $c = 474976$ is the best constant to minimize the error in that interval. We pushed further the method to find approximations for $ln(x)$, $\sqrt{(x)}$ and $\frac{1}{\sqrt{(x)}}$. We also found an approximation for x^y but it holds only for small x values, this approximation needs further study. All approximations are given in Figure 4.20.

For $ln(y)$:

$$u.f = e^y \iff u.i = a.y + b - c$$
$$u.i = \frac{u.i - b + c}{a} \iff u.f = y$$

Hence, to apply the ln function to y:

1. set $u.f = y$
2. return $u.i * \frac{1}{a} - \frac{b-c}{a}$

By observing that $\sqrt{x} = x^{\frac{1}{2}}$ and $1/\sqrt{x} = x^{-\frac{1}{2}}$, with the same type of reasoning we get an approximation of $\sqrt{x}$ and $rsqrt(x) = \frac{1}{\sqrt{x}}$.

We benchmarked these approximations against their native implementations from the *C++ standard library*. Benchmarking is not as obvious at it may seem, as there is *GCC* optimizations, CPU internal optimizations, concurrent processes, cache management, etc. So, it is hard to be sure what we are benchmarking. We applied the following benchmark procedure to exclude most of the possible biases:

TABLE 4.2 – Benchmark of the mathematical primitive approximations

	Exp	Log	Sqrt
Benchmark overhead	18s:303ms:165µs:883ns	18s:335ms:487µs:613ns	18s:226ms:485µs:376ns
Fast approximation	**22s:421ms:997µs:819ns**	**22s:350ms:916µs:250ns**	22s:197ms:376µs:928ns
C++ STD function	28s:889ms:791µs:718ns	30s:426ms:509µs:496ns	**18s:842ms:544µs:72ns**

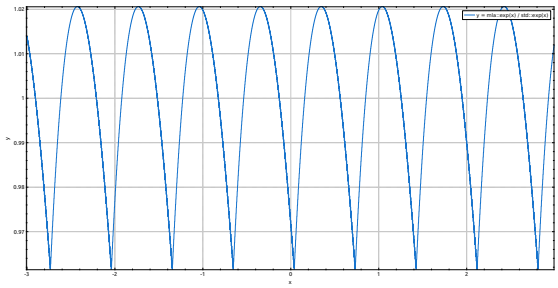


FIGURE 4.21 – Fast exp error

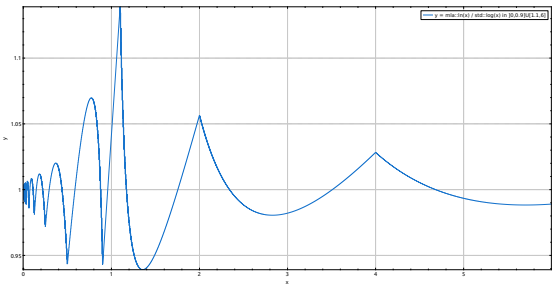


FIGURE 4.22 – Fast log error

- For testing a function, we accumulate the return value of the function with a random input between 0 and 10 over 1,000,000 iterations. That way, neither the CPU or GCC can predict the return value of the functions and optimize them. The high number of iterations copes with concurrent processes that may stop the processing and pollute the cache.
- The accumulator is made *volative*, so *GCC* does not optimize it away.
- Several operations (dependent on previous results) are made on the accumulator in order it does not becomes 0, inf or nan. The CPU processes operations differently with variables in those states.
- Before every benchmark we flush the *L1*, *L2* and *L3* caches by writing random numbers in an allocated table larger than the sum of the caches.
- The benchmark overhead is measured as a reference measure — i.e. the same code without the function calls. So for a benchmark we have 3 measures: the benchmark overhead, the approximation function and the native function.
- Benchmarks are run 5 times and for each measure, the best of 5 is chosen.
- GCC optimization options are set to *O2*.
- All tests have been made on a laptop with a *Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz* CPU.

The results are presented in Table 4.2. We took all precautions with benchmarking, but it is illusory to draw an exact performance ratio between the native and approximation version. From a computer to another, from a operation context to another, numbers will differ. What we can say with certainty is that the fast exponential and the fast logarithm are significantly quicker than their native counterparts. The square root function is not faster because the CPU of our computer has a square root instruction, and no program will beat a hardware instruction. We did not benchmark the $rsqrt(x)$ as there is no native version in the *C++ standard library*; it would be an unfair comparison.

In terms of error rate of these approximations, we calculated the approximation / native version ratio for small input ranges (corresponding to our use in neural networks). The error curves are presented in Figure 4.21, 4.22, 4.23, 4.24.

We observed the following maximal errors:

$$0.96 < \frac{mla::exp(x)}{std::exp(x)} < 1.02$$

$$0.97 < \frac{mla::ln(x)}{std::log(x)}, x \in]0, 0.9] \cup [1.2, +\infty[< 1.06$$

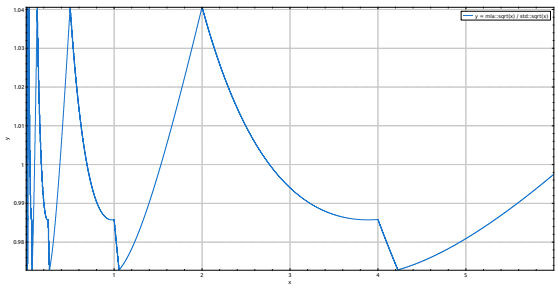


FIGURE 4.23 – Fast sqrt error

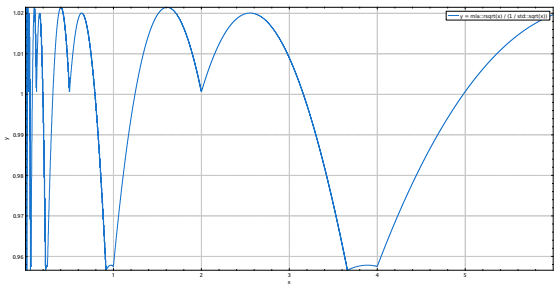


FIGURE 4.24 – Fast rsqrt error

$$0.97 < \frac{mla::sqrt(x)}{std::sqrt(x)} < 1.04$$

$$0.955 < \frac{mla::rsqrt(x)}{std::exp(x)^{-1}} < 1.022$$

Note that for $ln(x)$ there is a high error around 1 as the function tends to 0. For the rest of the range the error is quite close to 0.

We use the exponential approximation in *libmla* currently. The others are not useful yet. As a final word for this part, we notice that error curves for the square root and the inverse square root have a fractal shape. Whatever the zoom (centered on zero) we use, the curve shape stay the same. Not that it means something in particular, but it is interesting to notice.

4.3 Deep Learning

Deep neural networks, which have shown remarkable successes in recent years in a multitude of fields [12], are essentially neural networks, distinguished primarily by their depth. The data passes through multiple layers of neurons. Each layer trains on a distinct set of features based on the output of the previous layer. As we go deeper into the network, the layers are able to recombine features from previous layers, leading to increased complexity of features learned. There is no feature selection process in Deep Learning, compared to classical Machine Learning. The network is fed with raw input data and the successive apposition of layers learns what features are important to the model. As a drawback, they require a large amount of computing power and a large dataset size to be effective.

4.3.1 Main Models

Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of neural network that has shown excellent results in various classification tasks and image recognition [12]. CNNs are based on the assumption that there are spatially-local correlations. This is enforced by maintaining a local connectivity between neurons of the adjacent layers. Stacking many such layers leads to filters, which can learn more global features. In CNNs there is also the idea of shared weights. Using the same weights across multiple hidden units allows for features to be detected regardless of their position in the sample — it is why these kinds of networks fit image processing. It is also an effective way of increasing learning efficiency by reducing the number of free parameters to be learned. CNNs can be described in terms of 4 primary functions:

- **Convolution layer**. This step involves the convolution of the input vectors with a linear filter and adding a bias term. Let h^k_{ij} be the k^{th} feature map at a hidden layer. Given weights w^k and bias b_k, h^k_{ij} can be computed as follows:

$$h^k_{ij} = (w^k \cdot x)_{ij} + b_k$$

The size of the feature map is affected by three parameters:

- *Depth*. This refers to the number of filters used for the convolution.
- *Stride*. This refers to the number of steps the filter slides over the input matrix. Larger strides produce smaller feature maps.
- *Zero-padding*. Zero-padding the input matrix, or wide convolution [21] can allow us to apply the filters to border values in the input matrix and control the size of the feature map.

- **Non-linearity layer** The purpose of the non-linearity is to account for the non-linearity in most real-world data. Different types of non-linear functions are used for various models including *tanh*, *sigmoid*, *ReLU*, etc. *Rectified Linear Units* (*ReLU*, $h(x) = max(0,x)$) is an element-wise operation that replaces all negative values in the feature map by zeros. In Deep Learning, ReLU has been shown to outperform sigmoid or tanh (which is essentially a symmetric sigmoid) in most cases [22]. Half of the neurons are activated, and sparse activation is preferred on deep architectures for feature selection (as a recall, it is the network that learns the feature selection in Deep Learning) and for processing time. Moreover, when stacking neuron layers it creates a *vanishing gradient* effect with sigmoid-like transfer function. Gradient values are multiplied together and as the derivative of sigmoid-like function tends to zero when their input tends to their extremum values, the gradient magnitude diminishes from a layer to one other. ReLU, having a constant or null derivative, solves the vanishing gradient problem.
  Applying the non-linear function f_{N} to h^k_{ij} we get the rectified feature map.

$$\mathrm{F} = f_{\mathrm{N}}(h^k_{ij}) = f_{\mathrm{N}}((\mathrm{W}^k * x)_{ij} + b_k)$$

- **Max-pooling layer**. Max-pooling splits the input matrix into a set of non-overlapping spaces and, for each such sub-region, outputs the maximum value. It provides the following advantages:
  - It reduces the computational load by eliminating non-maximal values.
  - It gives us a sort of translational invariance [23]. In other words, by providing robustness to feature position it allows us to reduce the dimensionality of intermediate representations.

- **Classification layer**. Here, we employ a fully connected layer, which is a multilayer perceptron with a *softmax* transfer function (which is a form of generalized sigmoid, adapted for multiclass problems). The convolutional and pooling layers give us high-level features of the input matrix. The purpose of the fully connected layer is to classify the input based on these features. The layer is also a cheap way to learn non-linear combinations of these features. The output of the softmax activation is equivalent to a categorical probability distribution and makes sure that the sum of output probabilities of the fully connected layer is always 1.

Long Short-Term Memory Networks

Long-short term memory (LSTM) [24] is a variant of recurrent neural networks (RNNs). RNNs use recurrent connections within the hidden layer to create an internal state representing the previous input values. This allows RNNs to capture temporal context. This is why LSTM and RNN networks are adapted for learning sequential data. However, as the time interval expands, the updated gradient from backpropagation can decay or explode exponentially, referred to as the vanishing and exploding gradient problems respectively. This makes it difficult for RNNs to learn long-term dependencies. LSTM uses constant error carousel (CEC) to propagate a constant error signal through time, using a gate structure to prevent backpropagated errors from vanishing or exploding. The gate structure controls information flow and memory by tuning the value of CEC

according to current input and previous context. A gate is essentially a pointwise multiplication operation and a nonlinear transformation that allows errors to flow backwards through a longer time range.

4.3.2 Study of Deep Learning Applied to Android Malware Detection

Deep Learning has received little attention in the security domain, so we wanted to test its effectiveness and the ease of setting it up compared to MLP. This study has been achieved with the help of Abhilash Hota, M2 engineer from ESIEA in France. A paper relating this study is also in submission, at the time I am writing this PhD manuscript.

Input Data & Dataset

We tested different input data and dataset size:

- **Dataset 1**. This was the first dataset that our models were trained on. It consists of the results of static analysis performed on 3,500 malware samples from the Drebin Dataset [25] and 2,700 benign samples from *Fdroid*[4]. We took only unique samples from Drebin Dataset, in order to remove statistical bias in the experiment [26]. The neural networks were trained on opcode sequence of application source code, which is the sequence of Java bytecode operators. It was clear early on that deeper architectures were overtraining on such a limited dataset very quickly. This led us to create the second dataset.
- **Dataset 2**. We downloaded applications massively from *Androzoo* [27]. It is a growing collection of Android applications collected from several sources, including the official GooglePlay app market. It currently contains 7,412,461 different APKs, each of which has been (or will soon be) analyzed by tens of different antivirus products to know which applications are detected as malware. The APKs downloaded from Androzoo were verified against VirusTotal[5] to build a dataset of 150,000 malware samples and 150,000 benign samples. Keeping computational constraints in mind, we restricted ourselves to application samples of size less than 4MB.

Preprocessing steps involved in machine learning applied to the security domain are very similar to those involved in machine learning applied to the *Natural Language Processing* (NLP) domain. In fact, the source code of a program is a language — even if it is not natural. It has words that assemble themselves into syntax to form a meaning. In the NLP domain, it is common to pretrain Deep Learning with a dedicated neural network that learns a numerical representation of words.

It is achieved with *autoencoders* [28]. It is a multilayer perceptron which learns to produce its inputs. It has an odd number of hidden layers and the middle one is thinner than the input layer. As the network learns to reproduce its inputs, the middle hidden layer acts as a compression algorithm, trained to compress the input samples. Autoencoders have many applications, but their capacity to compress data can be hijacked to learn an efficient representation of the input space. This kind of use to pretrain machine learning is currently made in two ways:

- The autoencoder is trained, then the input dataset is passed through the network until the middle layer. Then, the output of the middle hidden layer is used to produce a new sample. The whole dataset is transformed with this method.
- After the autoencoder training, the first layers until the middle hidden layer are used for building a new neural network, whose first layers are the ones from the trained autoencoder.

Modern neural networks that learn representation are based on autoencoders. *Doc2vec* [29] is an unsupervised algorithm that generates vectors for sentences, paragraphs or documents. Many

4. https://f-droid.org/en/
5. https://www.virustotal.com/en

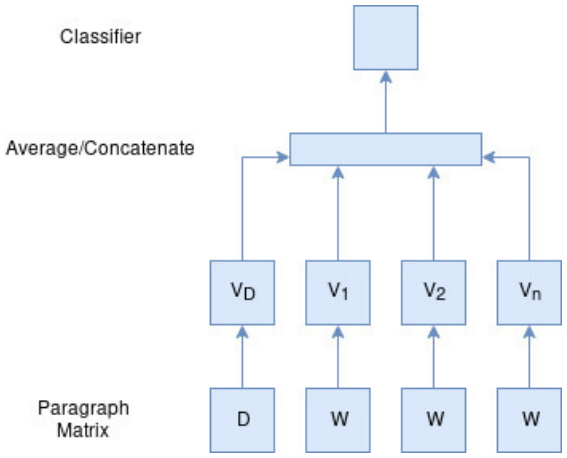


FIGURE 4.25 – Doc2Vec neural network architecture

neural network architectures require inputs to be fixed-length vectors. The *bag-of-words* [30] technique is one approach to this end, but has some drawbacks. It loses the ordering of the words and ignores semantics. Paragraph Vector, or doc2vec, was proposed as an unsupervised approach to learning fixed-length feature representations from variable-length texts, such as sentences, paragraphs or documents.

Every paragraph and every word are mapped to unique vectors which are averaged or concatenated to predict the next word in a given context. The paragraph label acts as a context memory. The paragraph vector stays consistent across all contexts from the same paragraph but not across paragraphs. The word vectors, however, are shared across paragraphs. In Figure 4.25 we present the architecture of doc2vec.

This approach was designed to resolve certain weaknesses in traditional bag-of-words models. It retains the semantics of the text and does not lose information about word order. The algorithm itself has two key stages:

- Training: compute word vectors W, softmax weights U, b and paragraph vectors D on already seen paragraphs
- Inference: compute paragraph vectors D for new paragraphs by adding more columns in D and gradient descending on D while W, U, b are fixed.

The input data are the sequences of raw bytes from the *DEX* file of the applications. These sequences are used to generate document vectors of length 300 representing each sample. We use two models of Deep Learning in our study: a CNN and a LSTM. The document vectors served as inputs for the convolutional neural networks. For the LSTM, the network is fed with the raw bytes.

Classification Models

For the experiment, we use 4 Deep Learning models, 3 CNNs and 1 LSTM with different internal and input configurations.

- **CNN 1**. This network, shown in fig.4.26, consists of three 2D convolutional layers, each followed by ReLU units and maxpooling. An initial dropout layer is added to avoid overfitting. The final output is obtained through a fully connected layer and softmax activation. The document vectors of length 300 were reshaped into matrices of dimensions (30,10) and used as the input. Essentially the network would be treating it as an input image. Thus the use of 2D convolutions. The model is trained using NADAM [31]. This is a variation of ADAM [14] that employs Nesterov momentum instead of classical momentum. Hyperparameters was optimized over 50 trials. A 9:1 split was maintained for training and validation.
- **CNN 2** This network has the same architecture as CNN 1, shown in fig.4.26. However the difference is that larger document vectors of length 10,000 were generated and the input

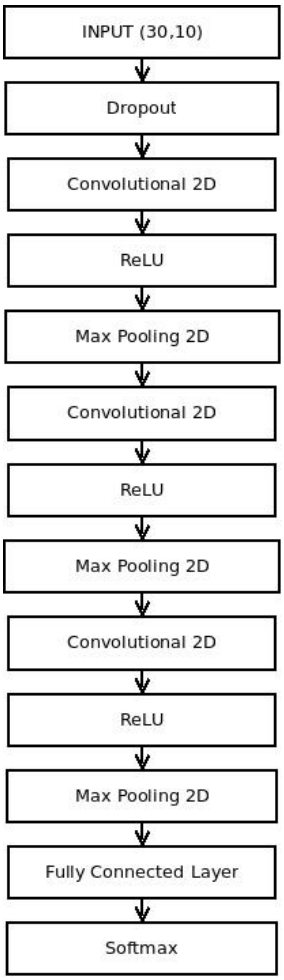


FIGURE 4.26 – CNN 1

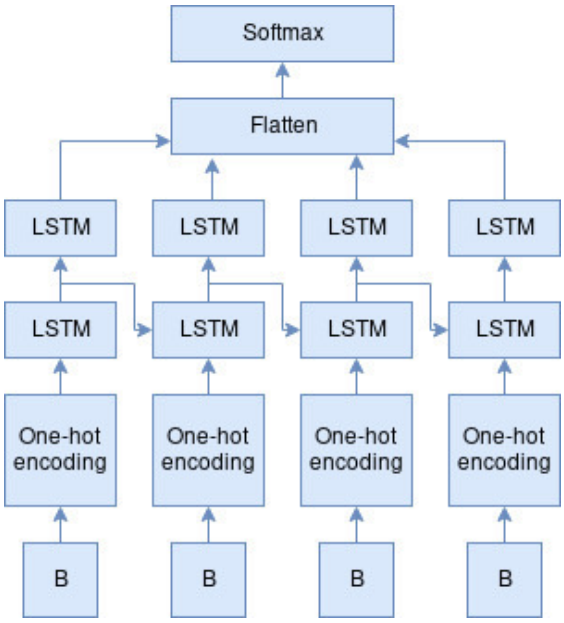


FIGURE 4.27 – LSTM Network

was a matrix of dimensions (100,100). The intention was to see if using larger document vectors would allow the network to learn better representations and thus be able to classify the samples better. The model is trained using ADAM. Hyperparameter optimization was performed by experimenting over 50 trials.

- **CNN 3** This model is based on the MalConv architecture [32]. This architecture was developed for malware detection by ingesting PE files. Here, it has been modified to accept raw bytes from the dex files extracted from the APKs as the input sequences. The architecture, shown in fig. 4.28 is based on CNNs. Here the problem is treated similarly to a natural language processing problem. The input bytes are tokenized and an embedding layer learns representation for the tokens. Since the input here is in the form of row vectors instead of 2D images, we deploy a 1D convolutional layer. The gating layer determines the fraction of information that is sent to subsequent layers. The model is trained using ADAM. Hyperparameter optimization was performed by experimenting over 60 trials.
- **LSTM 4** This model, shown in fig.4.27, is based on LSTM cells. The model is trained using ADAM over 1,000 steps, with validation checks every 20 steps.

Experimental Results

Training time over Dataset 1 was shorter. However, due to less data the models would easily overfit. Accuracy over the training set would quickly reach about 98-99% but accuracy over the test set was limited to about 60-80%.

Dataset 2 gives much better results considering the increased amount of data available (Table

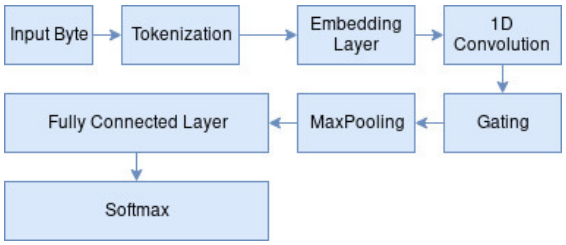


FIGURE 4.28 – CNN 3 - MalConv

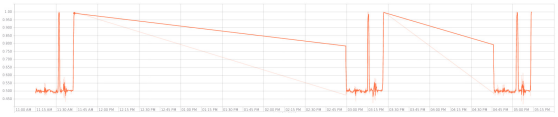


FIGURE 4.29 – Training Accuracy

FIGURE 4.30 – Training loss

4.3). Accuracy over the training set approaches 1 and over the test set approaches 95.3%. As expected training time is significantly increased to about 160-170 minutes (computer: *Intel Xeon e5-2686, 61GB RAM, Nvidia Tesla K80 GPU with 12 GB video memory*).

4.3.3 Conclusions

Deep neural networks have shown excellent results in recent years and they show promise in the field of malware detection. In their current state of development, however, they present certain challenges. They require immense amounts of data and processing power to learn the features that make them successful. When it comes to endpoint device security in Android systems, most smartphones running the OS have resource limitations. As such, building a real-time monitoring and malware detection system would currently not be feasible using deep neural networks on the endpoint devices. Moreover as Deep Learning takes the raw byte code as input, it is very sensitive to slight modifications of the binary. It requires a fixed length input, so any shift of the binary bytes (like adding bloating code) breaks the detection. As such, deep neural networks that learn to detect malware based on static analysis would be subject to the same limitations as traditional signature-based approaches. Malware authors have shown a considerable talent for avoiding signature detection systems. Adversarial neural networks have been very successful in fooling neural network based recognition systems [33] [34].

As such,it would be preferable to have deep neural networks running on the cloud and analyzing application behavior for applications available on the marketplaces, instead of running on endpoint devices for real-time detection. Potentially, unsupervised approaches to deep learning could be used to generate representations or signatures for malware. These signatures could then be used by more traditional signature-based detection systems on endpoint devices. There is a lot more scope for work in this field. The study presented here considers only static analysis or raw byte analysis. However the results do show promise and potential for dynamic analysis methods.

FIGURE 4.31 – Test Accuracy

FIGURE 4.32 – Test loss

TABLE 4.3 – Malware Detection Accuracy

	Dataset 1	Dataset 2
CNN 1	67%	94.4%
CNN 2	70%	95.1%
CNN 3	89%	93.7%
LSTM	70%	**95.3**%

BIBLIOGRAPHY

[1] Raúl Rojas. *Neural networks : a systematic introduction*. Springer Science & Business Media, 2013. 83, 89, 90, 102, 157

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*, volume 1. 2016. 83, 101, 103

[3] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4) :115–133, 1943. 84

[4] Arturo Rosenblueth, Norbert Wiener, and Julian Bigelow. Behavior, purpose and teleology. *Philosophy of science*, 10(1) :18–24, 1943. 84

[5] Norbert Wiener. *Cybernetics : Control and communication in the animal and the machine*. Wiley, 1948. 84

[6] AI Berg. Cybernetics and society. *Soviet Review*, 1(1) :43–55, 1960. 84

[7] I AREL. Reinforcement learning-based multi-multi-agent system for network traffic signal control. *Intelligent Transport Systems, IET*, 4(2) :128–135, 2010. 84

[8] Tahere Royani, Javad Haddadnia, and Mohammad Alipoor. Traffic signal control for isolated intersections based on fuzzy neural network and genetic algorithm. In *Proceedings of the 10th WSEAS international conference on signal processing, computational geometry and artificial vision*, pages 87–91, 2010. 84

[9] Manh Hung Nguyen, Tuong Vinh Ho, and Tan Hiep Nguyen. On the dynamic optimization of traffic lights. *Asian Simulation and Modeling, Mahidol University*, pages 35–43, 2013. 84

[10] Robert Stufflebeam. Neurons, synapses, action potentials, and neurotransmission. *Consortium on Cognitive Science Instruction*, 2008. 87

[11] RE Ucheya, UM Ucheya, and FE Amiegheme. Is a combine therapy of aqueous extract of azadirachta indica leaf (neem leaf) and chloroquine sulphate toxic to the histology of the rabbit cerebellum? *Annals of medical and health sciences research*, 1(2) :203–214, 2011. 87

[12] Jürgen Schmidhuber. Deep learning in neural networks : An overview. *Neural networks*, 61 :85–117, 2015. 96, 99, 115

[13] Brook Taylor. *Methodus incrementorum directa & inversa*. impensis Gulielmi Innys, 1717. 97

[14] Diederik P Kingma and Jimmy Ba. Adam : A method for stochastic optimization. *arXiv preprint arXiv :1412.6980*, 2014. 15, 53, 104, 118

[15] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1) :1929–1958, 2014. 15, 104

[16] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995. 104

[17] Derrick Nguyen and Bernard Widrow. Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 21–26. IEEE, 1990. 106

[18] Gilbert Saporta. *Probabilités, analyse des données et statistique*. Editions Technip, 2006. 108, 142

[19] Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008. 110

[20] Nicol N Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 11(4) :853–862, 1999. 112

[21] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv :1404.2188*, 2014. 116

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 116

[23] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity : The all convolutional net. *arXiv preprint arXiv :1412.6806*, 2014. 116

[24] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget : Continual prediction with lstm. 1999. 116

[25] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin : Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014. 33, 48, 49, 59, 117, 137, 152, 154

[26] Paul Irolla and Alexandre Dey. The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques*, pages 1–5, 2018. 49, 117, 137, 152

[27] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo : Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE, 2016. 117

[28] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 37–49, 2012. 117

[29] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196, 2014. 117

[30] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model : a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4) :43–52, 2010. 118

[31] Timothy Dozat. Incorporating nesterov momentum into adam. 2016. 118

[32] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole exe. *arXiv preprint arXiv :1710.09435*, 2017. 119

[33] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv :1312.6199*, 2013. 120

[34] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv :1606.04435*, 2016. 120

CHAPTER 5

Systematic Characterization of a Sequence Group

Contents

Finding similarities in a group of sequences often involves studying their common subsequences or their common substrings [1]. In our case, Android malware detection/classification, we study the event sequences coming from the dynamic analysis of applications. For several reasons, these sequences are mostly made of benign events. This specific set up makes classic sequence similarity criteria useless without any machine learning. The sequence membership to a group is characterized by subsequences of any length. Heuristic algorithms for extracting short subsequences already exist, but no attempt to solve the problem systematically has been proposed. We propose a new algorithm for building the *Embedding Antichain* from the set of common subsequences (noted A_Γ). We show that this mathematical representation is very compact and embeds all common subsequences of a sequence set. It is a tool for characterizing a group of sequences. The construction of this representation reveals several complex subproblems. A few of them are solved in this study, along with practical implementations. Moreover, we solved different reduced problems and provided suboptimal solutions for the others. This study opens a new path that has cross-domain applications. Specifically, in the malware detection/classification domain the *Systematic Characterization of Sequence Groups* is a tool that can be used for automatic generation of malware family signatures and detection heuristics. We experimented A_Γ for building an Android malware family detector, on the sequences of executed Android API calls and it yields the accuracy of 97.74%. The work in this chapter has been submitted at the ForSE 2019 conference.

5.1 Motivations

The classic method of detecting computer viruses is detection by pattern matching. When one or more samples of a malware family are found, analysts extract the characteristic features of the malware, called signature. It is often a sequence of opcodes, of library calls or strings in the executable that identify them. This process takes time and requires reverse engineering and malware analysis skills that only a few possess. That is why research currently focuses on Machine Learning to automate the malware detection process. However, the classical method is mostly used in industry and for individuals — in antivirus software. Requiring low computing power and offering a near-zero false positive rate, this is the method of choice still to date. We searched for methods and algorithms to automate the process of creating malware family signatures. Several samples of the same family of malware obviously have common features that differentiate them from other families of malware and benign software.

There are two ways of analyzing malware : static or dynamic analysis. Static analysis consists of reverse engineering the binary without execution. The source code can be rebuilt, as well as the different external resources used. The source code can be interpreted as a graph of calls of functions and instructions, when one knows the entry points — that is to say the functions that can be called from outside the program. On Android, for a given application, there may be dozens of entry points because an application must be able to react to lots of system events (the phone turns off, turns on, changes main application, etc.) in each activity of the application. The static analysis of the code in the form of Control Flow Graph can therefore generate dozens of graphs. Dynamic analysis consists of running the application in a controlled environment and retrieving traces of its execution. These traces are often system call sequences, Android API calls, I/O events, or network communications. The complexity of finding common characteristics between sequences is in essence much less than finding common characteristics between sets of graphs, so we moved to dynamic analysis.

We started from the assumption that malware applications from the same family run a specific sequence of events that characterize them. We do not know if this assumption is true or not, but this is the idea that founded the mathematical problem presented in this part, along with its resolution. The final results from the application of the *Embedding Antichain* to the malware classification problem shows that there some ground truth in this assumption, otherwise we would not been able to obtain a 97.74% accuracy.

The characteristic malicious sequence, however, can be scattered in a much larger mass of benign events. There are different reasons for that. An Android application manages many events related to the user interface, which are not *a priori* malicious. In addition, many malware families run their payload through repackaged applications, or through fake applications. The cases where an Android malware application contains only a payload being extremely rare, the malicious sequences of events are scattered in a stream of innocuous events.

Finally, the last problem is that Android applications are multithreaded. The reactivity of the user interface is dependent on it. Thus, on a time sequence of events, two consecutive events of a characteristic malicious sequence may be separated by dozens of events from another concurrent loop. We wanted a method that could handle these different problems, natively, and therefore a method that can be generalized to other problems. Another direction that can be taken, for the detection of viruses, is to pre-filter potentially malicious events in order to reduce the complexity of the problem.

Anyway we chose to design a deterministic algorithm able to handle these difficulties. The existing algorithms that study the similarity between sequences cannot handle these difficulties. *Longest Common Subsequence* (*LCS*) [2], *sequence alignment algorithm* [3], *Levenstein* [4], *Smith Waterman* [4], *N-Gram* based methods [4] (without any machine learning), are useless on this problem because the characteristic subsequence is a sparse data.

We started from the hypothesis that the set of common subsequences of a family of malicious sequences contains the information needed to establish a detection rule. Can we find a deterministic algorithm that is able to collect all common subsequences exactly between n sequences?

The naive approach has a $O(\sum_{i=1}^{n} 2^{|s_i|})$ complexity both in space and time where $\{|s|\}_i$ is the set of sequence sizes. That approach is highly impractical. We introduce the new problem of the *Embedding Antichain of Common Subsequences* and we show that this mathematical construction is a very compact representation of all common subsequences of a set of sequences. This mathematical object is new, to our humble knowledge, and the approach to determine the similarities within a group of sequence is also new — as it is not a statistical approach. As such, we found no state-of-the-art for building it, nor similar approaches.

We show how to build an approximation of this mathematical object, and that it is still an open problem. We explore the construction of a directed acyclic graph (DAG) from n sequences that can generate the *Embedding Antichain of Common Subsequences* (noted A_Γ). The complexity of the naïve approach shows us that it is a complex problem. We identified the hidden parameter of the complexity and we broke the problem into several subproblems. We first solve reduced versions of the subproblems to get a better understanding of the initial problem. Then we expose an algorithm that can build a suboptimal solution to this problem. At last we show how to use A_Γ to measure more accurately the similarity between a sequence and a group of sequences — i.e. the *Systematic Characterization of Sequence Groups*.

This chapter is organized as follows. In section 2 we introduce the notations and definitions we use in the chapter. In section 3, we present the *embedding antichain* and its characteristics i.e. the object of our study. Sections 4 & 5 present a solution for an easier problem, the *embedding antichain for sequences without intra-repetitions*. In sections 6 & 7 we provide a sub-optimal solution for the *embedding antichain for arbitrary sequences*. In section 8 we experiment our final algorithm as a classifier for Android malware and we conduct an empirical study of its time and space complexity. Last, in section 9, we bring a conclusion to this chapter.

5.2 Notation & Definitions

Let $\mathscr{A} = \{\lambda_1, \cdots, \lambda_n\}, n \in \mathbb{N}$ be a finite *alphabet* of n symbols. A *sequence* $s = (\lambda_1, \cdots, \lambda_m), m \in \mathbb{N}$ is a serie of symbols. Let Ω be the set of all sequences over $\mathscr{A}$. We define $\preceq$, a partial order over Ω, such that [5]:

$$(\{a\}, \{b\}) \in \Omega^2, \quad b \preceq a \iff \forall k, \quad b_k = a_{n_k}$$

Where $n_1 < n_2 < ... < n_k$ is an increasing sequence of indexes. When $b \preceq a$, b is said to be a *subsequence* of a. An *antichain* A over Ω is a *sequence* set such as:

$$\forall (a, b) \in A^2, \quad a \neq b \quad \Rightarrow \quad b \not\preceq a$$

Let S_n be a *sequence* set of size n. We note Γ the set of *common subsequences* of S_n. It is defined such that:

$$\Gamma = \{\gamma \in \Omega | \forall s \in S_n, \gamma \preceq s\}$$

A *Lower Set*, L, is a set such that if s_1 is in L and $s_2 \preceq s_1$, then s_2 is in L. Γ is, by definition, a *Lower Set*. A Γ-$embedding$ set is a set that generates Γ when we list all the unique subsequences of all its elements. For sake of simplicity, in the rest of the chapter, an *embedding* set is implicitly a Γ-$embedding$ set. A *maximal element* of Γ is a sequence that is not a subsequence of any other elements of Γ.

Last, we use two common operators on graphs, namely the *transitive reduction* [6] and the *transitive closure* [6]. The formal definition of the transitive reduction of a directed graph G is: *A directed graph G' is said to be a transitive reduction of the directed graph G provided that G' has a directed path from vertex u to vertex v, for any (u,v), if and only if G has a directed path from vertex u to vertex v, and there is no graph with fewer arcs than G' satisfying condition.*

The formal definition of the transitive closure of a directed graph G is: *A directed graph G' is said to be a transitive closure of the directed graph G provided that G' has an edge from vertex u to vertex v, for any (u,v), if and only if G has a directed path from vertex u to vertex v.* Figure 5.1 illustrates both operators, for *directed acyclic graphs* (*DAG*) in our context.

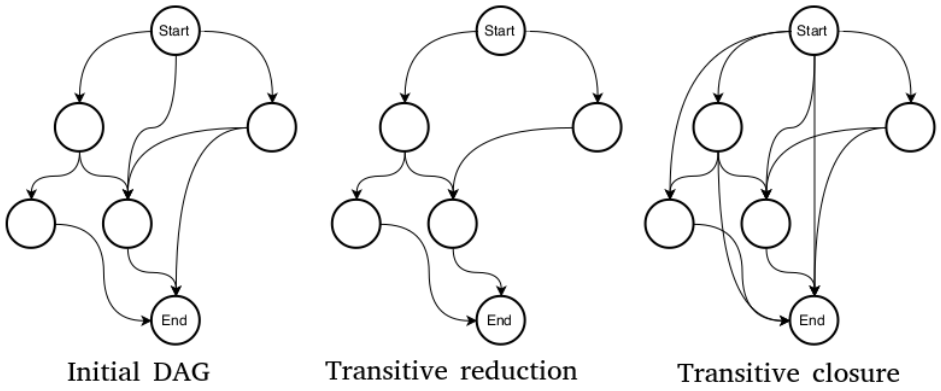


FIGURE 5.1 – Transitive reduction / closure

5.3 The Embedding Antichain

We define the *Embedding Antichain*, noted A_Γ, as the *Antichain* that generates Γ.

Proposition 5.3.1. *The Embedding Antichain is the minimal set that represents all common subsequences from n sequences.*

Proof. A property of an *Antichain* is that it can generate a *Lower Set* of the subsequences of its elements. As there is a one to one correspondence between an *Antichain* and a *Lower Set*, the *Embedding Antichain* is unique and is constituted by the *maximal elements* of Γ. Hence, the *Embedding Antichain* is the minimal set that represents all common subsequences from *n* sequences. ■

The representation of the *Embedding Antichain* can be even more compact by constructing the *minimal acyclic finite-state automata* (AFSA) [7] of this sequence set, because elements of A_Γ share symbols. It is a directed acyclic graph, that has one root node (named *START*) and one leaf node (named *END*). When enumerating all paths from *START* to *END*, it generates A_Γ. So our goal is to build a minimal DAG that generates exactly A_Γ. In Figure 5.2, we show how A_Γ represents Γ on a small example.

In Figure 5.3, we show an example of the construction of A_Γ DAG from 4 more complex sequences. In this example, enumerating all paths of the DAG from *START* to *END* ends up generating A_Γ. This DAG has been generated by an implementation of the algorithm we present in section 5.7.

The general idea in the study for A_Γ DAG construction is starting with two sequences to initialize the DAG, then processing each other sequence one after the other. Our objective is to design an algorithm that has sufficiently low asymptotic complexity for operational applications.

We have identified that the number of intra-sequence symbol repetitions — i.e. multiple occurrences of the same symbol in a sequence — is a factor that has a very high impact on the branching factor and the number of nodes of A_Γ DAG. To get a better understanding of the problem, we started solving a problem with a reduced difficulty by adding a constraint on sequences : the case where each sequence contains only unique symbols. We then expand the solution found for the subproblem to build a solution for the real problem. Throughout the article, we display instances of the algorithms on simple cases with words. The sequences that we use on our real application, and the output graph, cannot be displayed. They are simply too large.

For each of the following parts, the goal is to compute a DAG that generates A_Γ by enumerating all possible paths from a node noted START to a node noted END.

```
Sn = { "abcdd",
       "abddc" }

Gamma = { "a", "b", "c", "d",
          "ab", "ac", "bc",
          "bd", "dd", "ad",
          "abc", "bdd", "add",
          "abd", "abdd" }

A-Gamma = { "abc", "abdd" }
```

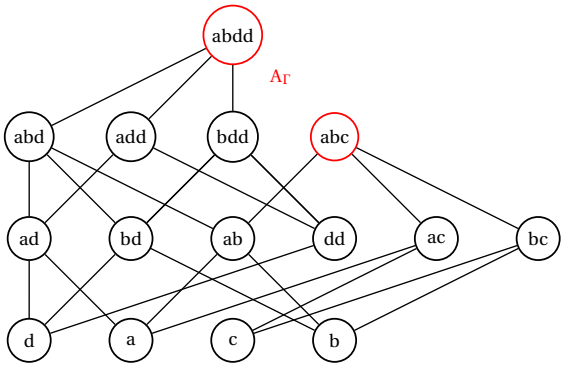


FIGURE 5.2 – A_Γ of Γ

```
Sn = { "absolumentsongeur",
       "mensongeabsolu",
       "solutionmensongere",
       "songeusementresolue" }

A-Gamma = { "solu",
            "sonso",
            "soeso",
            "mensoe" }
```

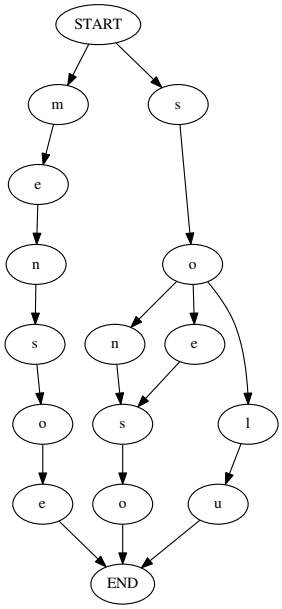


FIGURE 5.3 – Minimal A_Γ DAG

5.4 The Embedding Antichain Between Two Sequences Without Intra-Repetions

As we show in the next parts, repeated symbols in a sequence are an important contribution to the complexity of the problem. Adding constraints to our problem helps to compartment the different subproblems.

The following algorithm computes a DAG that exactly produces A_Γ when enumerating all possible paths from the START node to the END node. Each step is illustrated with an example, with the sequences ′ABCDEFGH′ and ′BDFAGHCE′ (Figures 5.4, 5.5 and 5.6):

Algorithm 5.4: A_Γ for two sequences with no repeat

```
input: s1 and s2, two sequences of size |s1| and |s2|
output: G(V, A), a DAG

begin
  create a matrix M of size (|s1|+2) * (|s2|+2)

  fill M[0][0] with a node of value START
```

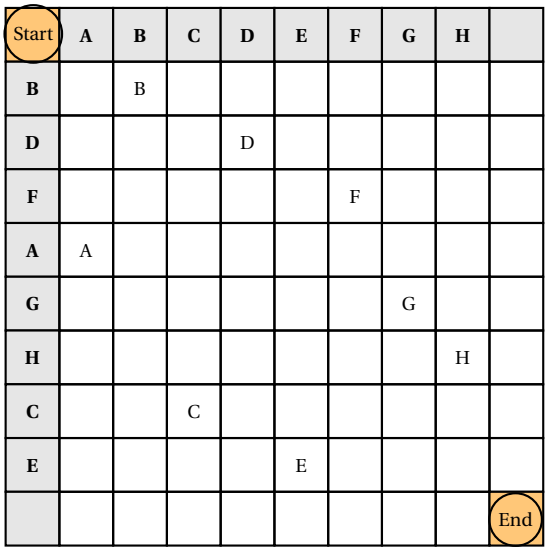


FIGURE 5.4 – Initilization

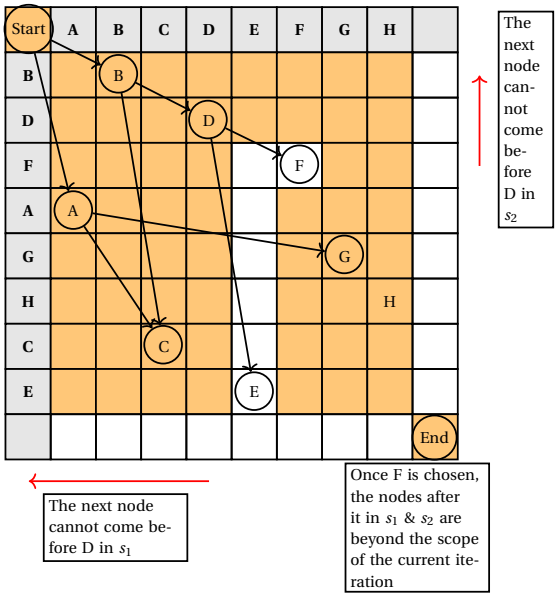


FIGURE 5.5 – Applying constraints on D for determining the next nodes of the DAG (E and F). Colored boxed cannot be chosen.

```
fill M[|s1|+1][|s2|+1] with a node of value END

for 0 < i < |s1|
  for 0 < j < |s2|
    if s1[i] = s2[j]
      fill M[i+1][j+1] with a node of value s1[i]
    else
      fill M[i+1][j+1] with a node of null value
    end if
  end for
end for
mark the node START as a node to process

while there are nodes to process
  node <- take the first one on the list
  mark all non null nodes in M as valid
  mark all nodes in M with
    i <= node.i and j <= node.j as invalid
  L <- list elements in M
  sort L in increasing order with the manhattan
    distance between node and L elements

  for element in L
    if element is valid
      add an edge linking node to element

      mark all elements in M
        with i >= element.i
        and j >= element.j as invalid
      add element as a node to process
```

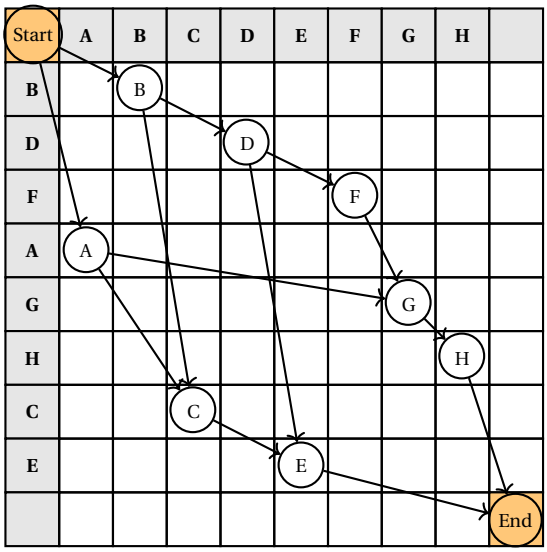


FIGURE 5.6 – Building A_Γ for two sequences without intra-repetition.

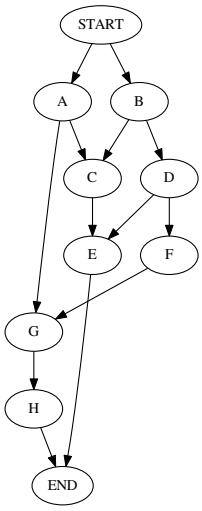


FIGURE 5.7 – Starting A_Γ DAG (result from previous section)

```
      end if
    end for
  end while
end
```

Proposition 5.4.1. *Algorithm 5.4 produces the minimal* A_Γ *DAG.*

Proof. The DAG nodes are common symbols and they are connected in increasing index order — increasing from both sequences. It produces Γ, the set of all common subsequences, meaning that the DAG produces a Γ-*embedding* set.

Moreover let us assume that the DAG does not produce any antichain. Let us consider all the possible paths from *START* to *END*. The sequences have no intra-repetition, so each DAG symbol. It means that at least one path of the graph is a subsequence of one other path. It implies the following situation, considering 3 arbitrary nodes A, B and C:

```
A -> B
B -> C
A -> C
```

If A is connected to B and C, it means that C is not reachable from B because the algorithm connects the current node to the closest one then constraints the next connections to nodes that are not reachable by the added child.

Consequently, this algorithm exactly produces the DAG of A_Γ for two sequences without repetition. As no arc or node can be removed without contradicting the previous properties, we also know that this is the minimal DAG that produce A_Γ. ■

Note that all algorithms presented in this study are designed to be human readable. The practical implementations we have designed are highly optimized versions and they cannot be presented as such in a research paper.

Order	0	1	2	3	4	5	6
Vertices	START	A,B	C,D	E,F	G	H	END

FIGURE 5.8 – Topological sort of G(V, A) ('depthMap')

5.5 The Embedding Antichain of n Sequences Without Intra-Repetition

The strategy we chose for finding the minimal A_Γ DAG of more than two sequences is incremental. We start with the previous algorithm for two sequences to get an initial A_Γ DAG. Then we process the sequences one after the other. We have found that this problem we are solving is very similar to the problem of building a *minimal acyclic finite-state automata* (AFSA) [7] of a word set. In this domain, researchers also chose an incremental strategy. Nonetheless, there are two major differences between this problem and ours. On the first hand we cannot represent or know in advance the elements of the word set. In our problem the word set is the common subsequence set, and as shown previously it grows exponentially with the sizes of the sequences. As such, we cannot represent it at any point of the solution. On the other hand, our DAG must be a *transitive reduction* [6], meaning we cannot remove an edge from the graph without removing a path from a vertex v to a vertex w. Hence, any solution for building the *AFSA* in the litterature are of no help here. Moreover, our graph yields more constraints than the classical *AFSA*. The new DAG exhibits the following properties:

- The DAG must be a *transitive reduction* otherwise there would be a path that is a subsequence of one other path.
- Each symbol is unique in the DAG because in each sequence, symbols are unique.
- As we process new sequences, the graph cannot grow in terms of vertices because symbols are unique.
- Let n be a node from the new DAG. Its children cannot come before in the *topological order* of the current DAG and in the indexes of the new sequence to process. These constraints hold because symbols are unique both in the current DAG and in the sequence to process.
- An edge cannot exist in the new DAG if this edge is not in the *transitive closure* of the current DAG. In other words, a node B is reachable from A in the new DAG if B is reachable from A in the current DAG.

The following algorithm computes the minimal A_Γ DAG and is illustrated with an example on three sequences $'$ABCDEFGH$'$, $'$BDFAGHCE$'$ and $'$EBGDFHAC$'$. The first and second sequences have been processed by the algorithm from the previous section to generate an initial DAG (Figure 5.7).

Algorithm 5.5: Minimal A_Γ DAG of $n+1$ sequences with no repetition, from the minimal A_Γ DAG of n sequences

---

```
input: G(V, A) a DAG and s a sequence of size |s|
output: G'(V', A'), a DAG

begin
  Let 'depthMap' be a map, that takes as value a list of
  nodes and as key their corresponding depth in the DAG

  depthMap <- topologicalSort( G(V, A) )
Let 'maxDepth' be the maximum key of depthMap

  Let 'TC' be the transitive closure matrix of G(V, A)
 (size |V|*|V|)
```

	Start	A	B	C	D	E	F	G	H	End
Start	0	1	1	1	1	1	1	1	**1**	1
A	0	0	0	1	0	1	0	1	**1**	1
B	0	0	0	1	1	1	1	1	**1**	1
C	0	0	0	0	0	1	0	0	**0**	1
D	0	0	0	0	0	1	1	1	**0**	1
E	0	0	0	0	0	0	0	0	**0**	1
F	0	0	0	0	0	0	0	1	**1**	1
G	0	0	0	0	0	0	0	0	**1**	1
H	0	0	0	0	0	0	0	0	**0**	1
End	0	0	0	0	0	0	0	0	**0**	1

FIGURE 5.9 – Transitive closure of G(V, A) ('TC')

	Start	E	B	G	D	F	H	A	C	End
0	Start									
1			B					A		
2					D				**C**	
3		E				F				
4				G						
5							H			
6										End

FIGURE 5.10 – New DAG initialization

```
  create a matrix M of size ((maxDepth+1) * (|s|+2))

  fill M[0][0] with a node of value START
  fill M[maxDepth+1][|s|+1] with a node of value END

  for 0 < i < maxDepth
    for 0 < j < |s|
      if at depthMap[i] there is a node of same value
 of s[j]
        fill M[i+1][j+1] with a node of value s[j]
      else
        fill M[i+1][j+1] with a node of null value
      end if
    end for
  end for
  mark the node START as a node to process

  while there are nodes to process
    node <- take the first one on the list

    for element in M
      if element.i > node.i
       and element.j > node.j
       and element is reachable from node in TC
        add an edge linking node to element
      end if

    end for
   end while

  G'(V', E') <- transitiveReduction(G(V, E))
end
```

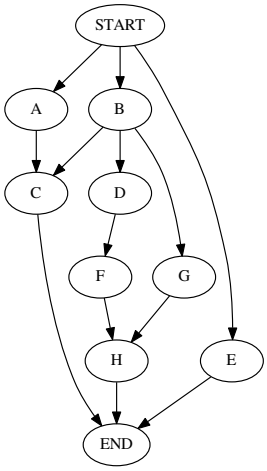


FIGURE 5.11 – Minimal A_Γ DAG for three sequences

The Figure 5.11 presents the result on the three sequences example.

Proposition 5.5.1. *Algorithm 5.11 produces the minimal* A_Γ *DAG.*

Proof. The algorithm starts essentially to connect all nodes to all others, excluding the impossible cases:

- A node cannot connect to a node that has a lower or equal topological order.
- A node cannot connect to a node that has a lower or equal sequence index.
- A node cannot connect to a node that is not reachable in the transitive closure.

These three conditions of node connections cannot remove a common subsequence. As a consequence, by enumerating all paths from *START* to *END*, it generates all possible common subsequences. At this point the DAG is already Γ-$embedding$. Then, by applying the *transitive reduction*, the graph become the minimal A_Γ DAG because symbols are unique so there cannot be two equal paths in the DAG. As nodes have unique symbols, no edge can be removed from the transitive reduction of the DAG without removing a subsequence. Therefore the graph is minimal and its paths form an antichain. It is the minimal A_Γ DAG. ■

5.6 The Embedding Antichain of Two Sequences

The intra-sequence symbol repetition invalidates the proof of the algorithm 5.4. To understand what symbol repetition induces, we applied Algorithm 5.4 on this new problem. We choose the sequences *"absolumentsongeur"* and *"mensongeabsolu"* (Figure 5.12) to illustrate the problematic.

The objective is to produce a minimal A_Γ DAG. We note that the DAG does not represent an antichain, we have highlighted in Figure 5.12 two edges that produces sequences that are subsequences of longer paths from *START* to *END*. The DAG is also not minimal because we can merge the two nodes *'u'* while being able to produce the exactly same unique paths. However the DAG produces a Γ-$embedding$ set. The DAG nodes are common symbols and they are connected in increasing index order — increasing from both sequences. It means that the DAG always produces a Γ-$embedding$ set.

We can easily make the DAG minimal by merging equivalent nodes at the end of the algorithm 5.4. Equivalent nodes are nodes with equal symbols that have exactly the same children or parents.

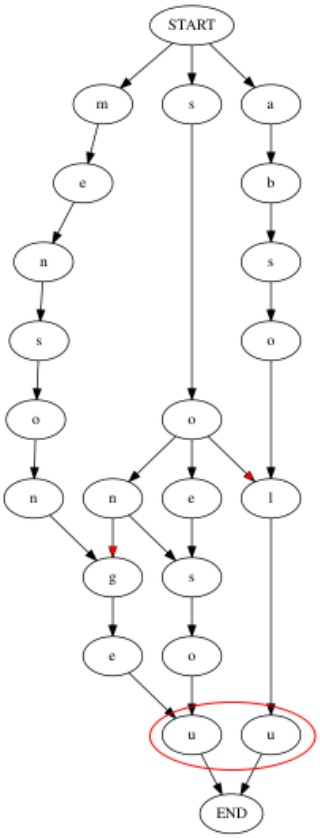


FIGURE 5.12 – Algorithm 5.4 on two sequences with intra-repetitions

When equivalent nodes are merged, a new node is created with children and parents from both equivalent nodes. However, designing an algorithm for finding the supernumerary edges (marked in red in Figure 5.12) is a challenging task regarding the fact that path enumeration in the graph has a polynomial complexity of high order.

Proposition 5.6.1. *The worst case complexity of path enumeration of a minimal* Γ*-embedding DAG is* $O(n^{k+1})$ *where* n *is the maximal number of nodes from all topological levels and* k *is the number of topological levels.*

Proof. We construct a DAG, denoted B, that contains all the sequences from a minimal Γ*-embedding* DAG, denoted A. This DAG has $n * k$, n nodes in each topological level. The nodes from A are in B in the same topological levels. All nodes are connected to all nodes from its next topological level. B has n^{k+1} paths, so the worst case complexity of A is $O(n^{k+1})$. ■

For the computation of the minimal A_Γ DAG, we tried different methods. None of them can compute the minimal A_Γ DAG with an acceptable algorithmic complexity. Our best attempt is founded on the fact that with reversed sequences for each DAG node, its children should be its parents and conversely. We note that applying 5.4 in forward and backward order leads to different DAG that are not equivalent when reversing children and parents. Relying on this observation, we reached an algorithm close to the solution:

Algorithm 5.6: Subsequence DAG for two sequences

---

1. Apply Algorithm 5.4 from Start to End (forward pass) and from End to Start (backward pass). The backward pass is the equivalent to a forward pass with sequences in reverse order.

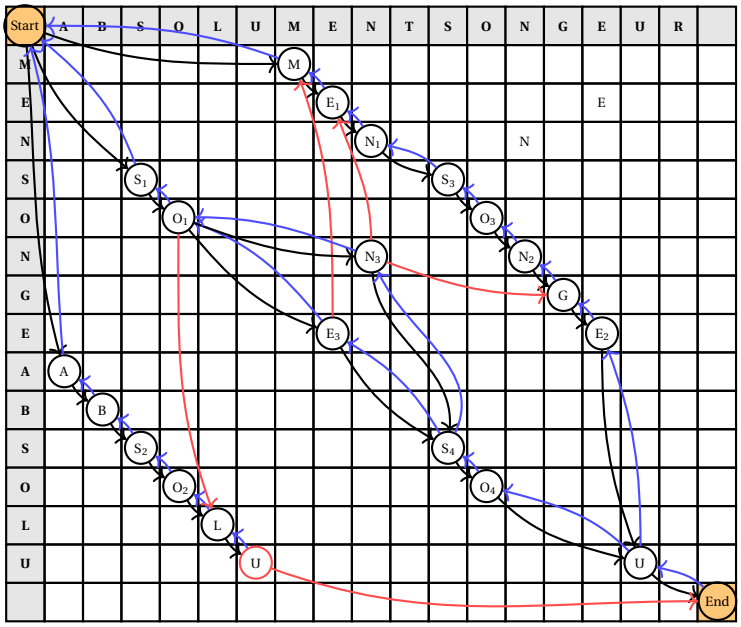


FIGURE 5.13 – Applying algorithm 5.4 forward and backward

−− > : Forward pass Vertex : Vertex to merge
< −− : Backward pass < − − / − − > : Path to delete

2. Merge equivalent vertices. Equivalent vertices are two vertices that have the same symbols and the same children or the same parents. Edges from the forward and backward passes are considered for the children/parent comparison. If no nodes have been merged, go to step 5.
3. Apply the transitive reduction on both forward and backward graphs. If no edge has been deleted, go to step 5.
4. Go to step 2.
5. Delete all edges that have no equivalent edge in the other step. An edge from vertex v to u from the forward step is considered equivalent to an edge from u to v from the backward step.
6. Some nodes can be left without parents, we call them *orphan nodes*. Link the orphan nodes to its parents in the forward and the backward DAG that have a common symbol and that still exists in the current DAG. If no such parent has been found, delete the orphan node.
7. Some nodes can be left without children, we call them *single nodes*. Link single nodes to its children in the forward and the backward DAG that have a common symbol and that still exists in the current DAG. If no such child has been found, delete the single node.
8. If an orphan or a single node has been deleted, go to step 6.

This algorithm works on the example with *"absolumentsongeur"* and *"mensongeabsolu"*, as shown in Figure 5.13.

However, when tested with longer sequences we noted that this additional constraint on the edges induces a problem and we found that this algorithm does not produce A_Γ or a Γ-*embedding* set, but a set that contains less or equal information. This problem arises when a forward path is equivalent to a backward pass but with different nodes.

5.7 The Embedding Antichain of n Sequences

Compared to the same problem without intra-repetition, processing the Γ-*embedding* DAG of $n+1$ sequences from the Γ-*embedding* DAG of n sequences brings two new difficulties. The first one is that each node from the previous DAG, that has several matches in the new sequence, must be duplicated in the {topological level / sequence index} matrix. It means, when using Algorithm 5.7 for an arbitrary node, that it can connect to children that are duplicated, actually creating duplicated paths or supernunary subsequences. To cope with this problem, a node must not be connected to more than one duplicated child. The second problem is the introduction of duplicated paths as two different nodes can have the same symbols. Like in the previous section, we apply a {merge / transitive} reduction phase to remove the most part of it. If the previous graph is a Γ-*embedding* DAG of $n+1$ sequences then the result is a Γ-*embedding* DAG, because none of the steps can remove a node or an edge that breeds a unique subsequence. Here is the full algorithm:

Algorithm 5.7: Γ-*embedding* DAG for n+1 sequences from a Γ-*embedding* DAG of n sequences

---

1. Process a topological sort and the transitive closure of the DAG.
2. For each node in the DAG create a new node for each corresponding symbol in the sequence. Every new node created at one iteration is called duplicated node.
3. Connect each new node, of topological level i and of sequence index j, to all new nodes that satisfy the following conditions:
   - It comes from a node reachable in the transitive closure of the DAG.
   - Its topological level is at minimum $i+1$.
   - Its sequence index is at minimum $j+1$.
   - The node has not been connected to another duplicated node.
4. Merge equivalent vertices. Equivalent vertices are two vertices that have the same symbol and the same children or the same parents. If it is not the first iteration of the merge phase and no vertex have been merged, stop the algorithm.
5. Process the transitive reduction of the current graph. If no edge has been deleted, stop the algorithm.
6. Go to step 4.

5.8 Experimental Results

In this part, we test our final algorithm on sequences coming from the dynamic analysis of benign and malware applications. The aim of the first experiment is to determine the time and space scalability of the algorithm. We identify parameters that influence the evolution of the processing time and the memory footprint of the program, then we show how the algorithm reacts under a large range of values. In the second experiment we explore a way of using the graph produced by our algorithm to detect if an application belongs to a particular malware family. For both our experiments, we explain in the first subsection exactly what are those sequences, how we collect them and how they are pre-processed.

5.8.1 Sequence Collection and Processing

We use a dataset of 719 Android malware and 521 benign samples. Malware samples come from the *Drebin Dataset* [8]. It is the most widely used malware dataset, in the domain of Android malware detection. In a previous study [9], we observed that a large part of malware samples

from this dataset share the same opcode sequences — i.e. the bytecode sequence without the operand. A large part of malware applications use repackaging, this is why for two malware samples that share the same opcode sequence, the only changes are often strings, class names and assets. 49.35% of applications have this characteristic. We showed that it artificially boosts the performance of machine learning algorithms because samples that have been learned from the training set are also found in the testing set. So, in our experiment we only use malware samples that exhibit different opcode sequences, that way we get an unbiased performance result of generalization capacity of the algorithm. The benign samples come from *F-Droid*[1], a repository of Free and Open Source Android applications. These applications are not a priori malicious.

We chose to represent a sample by the sequence of Java calls to the Android API it executes during a dynamic analysis process. In chapter 3 we came to the conclusion that Android API calls are features with a potent discriminative power when it comes to classification. We use *Glassbox* [10] to collect the sequences of API calls.

The number of samples of the dataset is not high enough for a large scale experiment. We were contraint by the lack of resources, the lack of time and several technical constraints. Indeed, we only used one phone and many applications could not be executed on it. We would have needed a system of industrial quality. Many phones, covering a wide range of Android versions. A multi-charge system to aliment the phones, since the dynamic analysis process consumes more battery than what the usb power supply produces when it connected to a computer. Another explanation is that most of the malware command and control servers are currently shut down. Academic dataset contains outdated malware applications. Hence, many applications are no longer working and therefore no longer produce dynamic data.

The application UI relies on polling loops to be reactive, and a large portion of the code logic is handled by loops. This results in repeated substrings within the sequence. These repeated substrings do not convey new information and hinder both the performance and efficiency of algorithms that process them. These artefacts, in the domain of biology are called *tandem repeat*[2]. Lin & al. [11] also identified this problem in their study of system call sequences from the dynamic analysis of applications. They solved it by removing consecutive call repetition, i.e. tandem repeats of size 1. We designed a tandem repeat removal algorithm to increase both performance and efficiency of downstream algorithms. Let be ttr, a function that keeps only the first instance of every tandem repeat. For example:

$$trr(\text{"}aezezezabcdabcd\text{"}) = \text{"}aezabcd\text{"}$$

The algorithm we use is inspired from a naïve approach. Figure 5.14 shows this algorithm running on the previous example.

On the diagonal is the number of the iteration. To detect a tandem repeat, we run the matrix, noted m, downwards, starting from the current iteration on the diagonal, until we find a position where $m(i,j) = 1$. The distance d, between the starting point and the first valid position is saved. Then, we run the matrix diagonally until a non-valid position is found, i.e. $m(i,j) = 0$. We save the traveled distance, d'. We have $d' = k.d + r$ with $(k,r) \in \mathrm{N}^2$. Last we erase the $k.d$ first positions from the diagonal. To erase a position (i,j), we remove all (i',j) and all (j,i') for all i' possible positions. Then the matrix is reassembled. In our example, removal appeared at iteration 2 and 4.

This naïve algorithm is obviously of $\mathrm{O}(n^2)$ time and space complexity where n is the sequence size. It is impractical. We noted that tandem repeat size in our data rarely exceeds 10. So we break the sequence in parts of random size — between 100 and 500 — and apply this algorithm on them. As the breaks can happen in the middle of a tandem repeat, we apply again this procedure until there is no change in the sequence or 3 times maximum. In this way, our tandem repeat removal algorithm has a time and space complexity of $\mathrm{O}(n)$.

1. https://f-droid.org
2. https://meshb.nlm.nih.gov/record/ui?ui=D020080

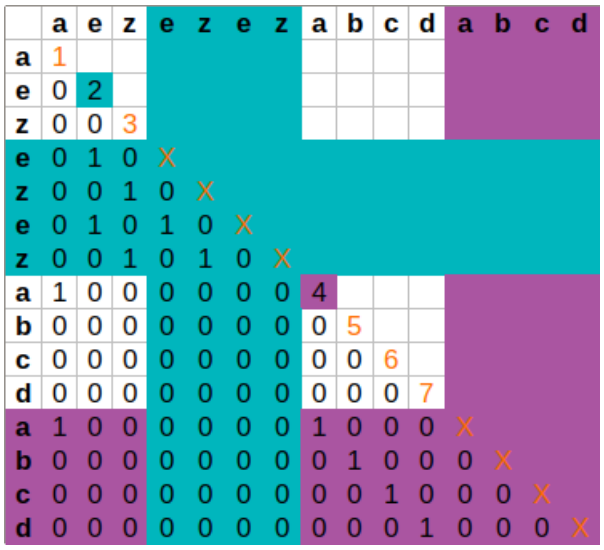


FIGURE 5.14 – Tandem repeat removal algorithm

5.8.2 Evolution of the Computation Time and the Memory Footprint

Theoretically analyzing the complexity of Γ-*embedding* DAG algorithm is a challenging task, as we implemented it with many optimizations. We assess empirically the evolution of the computation time and memory footprint of the algorithm for several parameters. The memory footprint is measured with $max(|V|+|E|)$, where $|V|$ and $|E|$ are the number of nodes and edges in the DAG, max takes the maximum of $|V|+|E|$ that have been reached during the DAG construction.

We identified three parameters that we can control and that influences the complexity: the number of sequences to process, the size of the sequences and the size of the alphabet. Then we use linear regression to find the best approximation of the complexity regarding each of these parameters. First, we define the statistical tools we use to qualify the following experiments. Last, we provide the experimental results along with the statistical evaluation. In this problem, we are unable to construct the worst cases for the algorithm to test its worst case complexity. Moreover we do not have enough real cases to have a fine grain control over the parameter we want to assess. To provide input data, we generate random sequences with control over the sequence size and the alphabet size. The following experiments give a general idea of the average asymptotic complexity for real cases albeit exceptions may occur for some particular patterns and configurations.

The coefficient of linear correlation, also known as *Bravais-Pearson coefficient* [12] is used exclusively to measure the linear relationship between X and Y:

$$r_{X,Y} = \frac{Cov(X,Y)}{\sigma_X \sigma_Y}$$

Where COV(X,Y) is the *covariance* between X and Y, σ_X is the standard deviation of X. If $Y = aX + b$ then $r = \pm 1$. *Linear regression* is the search of a couple $(\hat{a}, \hat{b})$ that minimize:

$$f(\hat{a}, \hat{b}) = \sum_{i=1}^{n} (y_i - (\hat{a}x_i + \hat{b}))^2$$

With the *least squared method*, we get:

$$\Delta_{Y/X}: \hat{Y} = \hat{a}X + \hat{b} \text{ with } \hat{a} = \frac{Cov(X,Y)}{V(X)} \text{ and } \hat{b} = \bar{y} - \hat{a}\bar{x}$$

Where V(X) is the *variance* of X. We can find other kind of relationship with the linear regression method. By transformation of X and Y to $g(X)$ and $g(Y)$, we can estimate the linear relationship between the transformed series. In this study, we assess the following relationships:

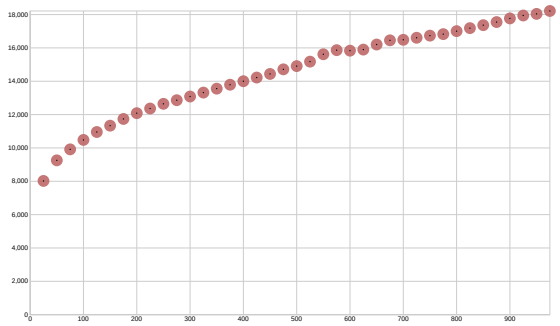


FIGURE 5.15 – Time (ms) / Number of sequence experiments, sequence size: 100, alphabet size: 5

TABLE 5.1 – Correlation between the processing time and the number of sequences

	$\hat{Y} = a\sqrt{X} + b$	$\hat{Y} = a\log X + b$
a	369.109	2996.071
b	6703.207	-3310.605
r	**0.998**	0.977
$s_{\hat{Y}}$	**135.736**	561.187

$$Y = a.log(X) + b \text{ (logarithmic relationship)}$$

$$Y = a\sqrt{X} + b \text{ (squared root relationship)}$$

$$log(Y) = aX + b \text{ (exponential relationship)}$$

$$\frac{1}{\sqrt{Y}} = aX + b \text{ (hyperbolic decline relationship)}$$

The *standard error* of the estimation is a measure of the dispersion between the real series X, Y and the regression curve:

$$s_{\hat{Y}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

In Figure 5.15 is presented the effect of the variation of the number of sequences over the processing time.

The input data is random sequences of 100 characters. The alphabet is of size 5 and each character is uniformly distributed. The algorithm is very dependent on the number of intra-repetitions but we have no control over this parameter. To measure effectively the effect of the variation of the number of sequences, i.e. not being affected by random noise, for each DAG construction we keep the previous random sequences and we add 25 new random sequences. The plot clearly indicates a logarithmic or squared root relationship between the number of sequences and processing time. In Table 5.1 we present the coefficients of linear correlation and the standard deviation for each of the regression curves we tested.

While both squared root and logarithmic relationships have very high linear correlation coefficients, the square root relationship best fits to the serie. the computation time and the memory footprint evolve most probably as $\sqrt{n}$ where n is the number of sequences. This experiment also shows that the space complexity does not depend on the number of sequences ($max(|V| + |E|)$ is always equal to 3,325).

Figures 5.18 and 5.19 present the effect of the variation of the sequence size over time and space.

The alphabet size is equal to 5 and the number of sequences is equal to 10. To cope with noise induced with the random intra-repetitions, at each iteration of a new DAG generation, we reuse the previous sequences and added a random character from the alphabet to them. The plot clearly shows an exponential relationship, and the correlation with it (in Table 5.2 and 5.3) is close to 1.

At first we calculated the correlation for the whole series but we found that the series are almost perfectly correlated with an exponential relationship for sequence sizes over 100. This is easily explained with some constant processing and structures to allocate that is not negligible for small sequences. Therefore the computation time and the memory footprint of Algorithm 5.7 evolve most probably as e^n where n is the size of the sequences.

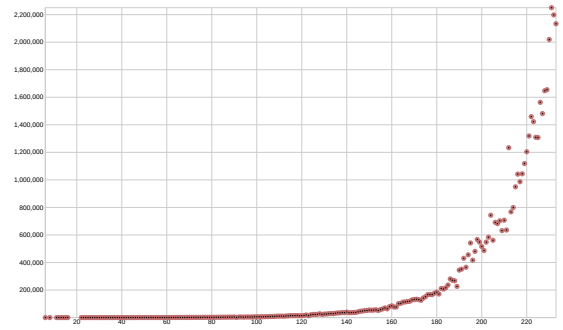


FIGURE 5.16 – Time (ms) / Sequence size, alphabet size: 5, number of samples: 10

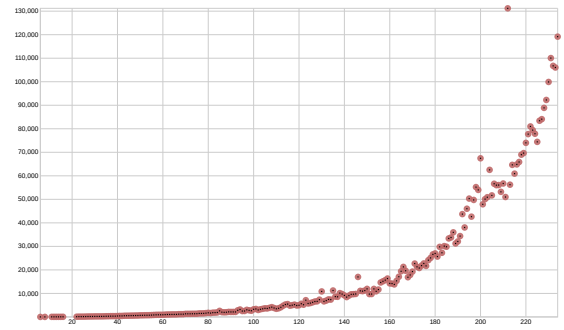


FIGURE 5.17 – $max(|V|+|E|)$ / Sequence size, alphabet size: 5, number of samples: 10

TABLE 5.2 – Correlation between the processing time and the sequence size

	$log(Y) = aX + b$ (size > 0)	$log(Y) = aX + b$ (size > 100)
a	0.053	0.043
b	2.641	4.349
r	0.982	**0.997**
$s_{\hat{Y}}$	364762.214	**78798.057**

TABLE 5.3 – Correlation between $max(|V|+|E|)$ and the sequence size

	$log(Y) = aX + b$ (size > 0)	$log(Y) = aX + b$ (size > 100)
a	0.030	0.026
b	4.682	5.388
r	0.982	**0.992**
$s_{\hat{Y}}$	14223.207	**7642.159**

Last, we measure the effect of the alphabet size over time and space (Figure 5.18 and 5.19).

The strategy here to cope with noise is to add progressively new characters with the minimal sequence change while preserving the uniform law of character random apparition. This is done, at each new DAG iteration, by replacing every character from all sequences by the new character with a probability of $\frac{1}{a+1}$ where a is the previous size of the alphabet. Let assume that a character, c, have the probability of $\frac{1}{a}$ to be present at an index, i, of a sequence. At index i, c is replaced with a probability of $\frac{1}{a+1}$. Hence, c is present at index i with a probability of $P(c) = \frac{1}{a}.\frac{a}{a+1} = \frac{1}{a+1}$. Consequently at the new iteration, characters follow a uniform law.

We tested the correlation of time and space series with an *exponential decline* and a *hyperbolic decline* in Figure 5.4 and 5.5.

The computation time and the memory footprint most probably evolve as e^{-a}, where a is the size of the alphabet.

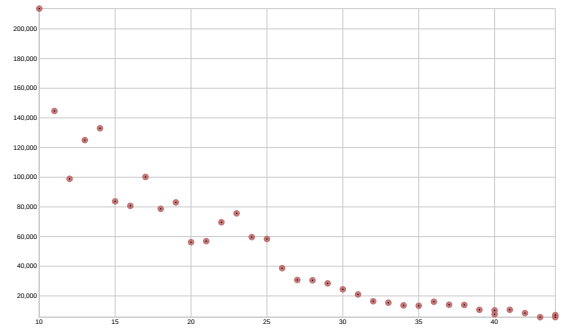


FIGURE 5.18 – Time (ms) / Alphabet size, sequence size: 200, number of samples: 10

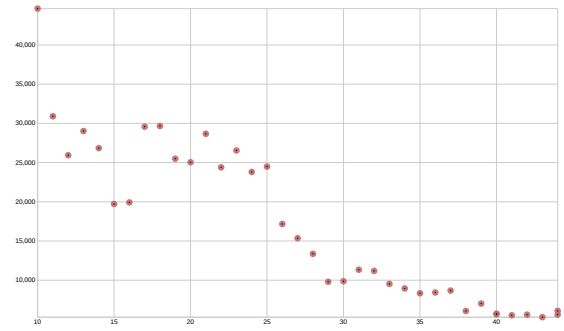


FIGURE 5.19 – $max(|V| + |E|)$ / Alphabet size, sequence size: 200, number of samples: 10

TABLE 5.4 – Correlation between the processing time and the alphabet size

	$log(Y) = aX + b$	$1/\sqrt{Y} = aX + b$
a	-0.097	0.000295
b	13.046	-0.00176
r	**-0.983**	0.958
$s_{\hat{Y}}$	**13468.820**	105989.55

TABLE 5.5 – Correlation between $max(|V|+|E|)$ and the alphabet size

	$log(Y) = aX + b$	$1/\sqrt{Y} = aX + b$
a	-0.059	0.000265
b	11.185	0.001647
r	**-0.954**	0.951
$s_{\hat{Y}}$	**4329.973**	6479.356

5.8.3 Classification Experiment

The Γ-*embedding* DAG contains common information between a group of sequences. To exploit these data for assessing if a new sequence belongs or not to a group, we need to define measures. Let us detail our approach. We collect common symbols between all sequences from a group, then we remove any non-common symbols from all sequences. These sequences are sorted by their length in increasing order. It becomes the order of sequence processing. As the shortest sequences are treated first, it is most likely that at each iteration, we get the smallest DAG possible.

We apply Algorithm 5.6 on the first two sequences and Algorithm 5.7 for the following ones. To assess if a new sequence belonging to the group, we use Algorithm 5.7 again with the DAG from the group. Then we define several measures that give indications about the changes on the DAG. The less the DAG is reduced, the most likely the new sequence belongs to the group because it shares a wide number of common subsequences.

Let N_b, E_b, P_b be respectively the number of nodes, edges and paths (from START to END) in the DAG before the new sequence processing. Let N_a, E_a, P_a be respectively the number of nodes, edges and paths in the DAG after the new sequence processing. We define three criteria: *node expansion, edge expansion* and *path expansion* noted N_{ex}, E_{ex} and P_{ex}.

$$N_{ex} = \frac{N_a - N_b}{N_b}, E_{ex} = \frac{E_a - E_b}{E_b}, P_{ex} = \frac{P_a - P_b}{P_b}$$

Algorithm 5.7 is very sensitive to outliers in sequence clusters. The sequence that share the less common subsequences with the group will reduce the graph the most. For that reason, we re-clustered malware families to get groups of sequences that are really close. To accomplish this task, we use a distance between two sequence clusters that favor common subsequences:

$$D(c_1, c_2) = \frac{1}{|A_{c_1} \cap A_{c_2}|}, \text{ the inter-cluster distance}$$

$$D(c, c) = \frac{1}{|A_c|}, \text{ the intra-cluster distance}$$

Where A_c is the set of symbols in the sequence cluster c. $0 < D(c_1, c_2) \leq 1$. When the sequences in a group share a lot a common symbols, D tends to 0, and conversely when the sequences share just a few symbols, D tends to 1.

We have designed a hierarchical clustering to group similar sequences based on the D distance. At initialization, each sequence is within a separated cluster, then at each iteration, the clusters that have the lowest distance are merged. The common symbols between both clusters become the symbol set of the new cluster. In that sense, when clusters are created we lose the information about the individual sequences. To find the optimal number of cluster we minimize a Ward criterion [13]:

$$r = \frac{\sum_{\forall c_i \in C} D(c_i, c_i)^2}{\sum_{\forall c_i, c_j \in C * C | i \neq j} D(c_i, c_j)^2}$$

Where C is the set of clusters. At each iteration r is calculated and the iteration with the lowest r is considered as the optimal iteration. To promote clusters that are consistent regarding our problem, we added several constrains to the clusters:

- At least 50% of the points must be clustered
- Clusters of size 1 are not considered (even in r calculation)
- Clusters of size below 20% of the maximal size among clusters are not considered (even in r calculation).

These rules ensure that small clusters are filtered — they are outliers regarding our problem — and that at least a majority of the malware dataset is taken into account. Another issue is that

TABLE 5.6 – Benign sequence clusters

Cluster N^o	number of sequences	number of common symbols
0	15	289
1	8	33
2	16	23
3	9	21

TABLE 5.7 – Malware sequence clusters

Cluster N^o	number of sequences	number of common symbols
0	15	34
1	28	20
2	33	75
3	11	23
4	22	17
5	12	39

TABLE 5.8 – Leave-one-out cross validation measures

Cluster N^o	0	1	4
Node expansion mean	0%	-2.455%	-2.769%
Node expansion standard deviation	0%	7.411%	12.262%
Edge expansion mean	0%	-2.469%	-2.702%
Edge expansion standard deviation	0%	7.673%	12.013%
Path expansion mean	0%	0%	-1.705%
Path expansion standard deviation	0%	0%	7.995%

TABLE 5.9 – Classification results

	Correct classification
leave-one-out malware clusters	59 / 65 (90.77%)
malware sequences	654 / 654 (100%)
benign sequences	499 / 521 (95.77%)
overall	1212 / 1240 (97.74%)

Android malware and benign applications share API patterns linked to the use of UI. The common subsequences derived from UI API calls is of little use on malware detection. Hence, to refine malware sequences, we first cluster benign applications, take the symbols sets of the valid clusters and subtract them from malware sequences 5.6.

Then we cluster malware sequences, results are presented in Table 5.7.

We note that benign sequences are more diverse — 48 / 521 benign sequences within selected clusters — than malware sequences — 121 / 719 malware sequences within selected clusters. We use only malware clusters 0, 1 and 4 because they lead to Γ-*embedding* DAG that are very fast to generate. Other clusters lead to DAG with more than 10^5 nodes and edges. They require too much computing time to generate. The approximation we get leads to errors, mostly added nodes and edges. These errors grow with the number and size of the common subsequences. For some pathological cases, the graph is constitued mostly by errors. So, before the graph is reduced by the node merging and transitive reduction phases, the intermediate graphs are already too big to be processed. A better approximation should enable more cases to be processed. This subject still requires extensive research, as it is a domain at its beginning.

Next, for each cluster we use *leave-one-out cross validation*. One sequence is removed from a cluster, then the Γ-*embedding* DAG is processed from the rest. The sequence that has been left out is added to the DAG and we measure the *node, edge and path expansion*, the results are presented in Figure 5.8.

The standard deviation of {node, edge, path} expansion is used as a criterion of a sequence belonging to a cluster. We consider that a new sequence belongs a cluster if its {node, edge and path} expansion deviation from the mean is below the respective standard deviations. We tested all benign sequences and all malware sequences — that were not already belonging to a cluster. The results are presented in Figure 5.9.

A sequence is considered to be correctly classified if a sequence is predicted to belong in the right cluster for leave-one-out malware sequences or if the sequence is predicted to not belong to any of the clusters for the rest of the sequences. The overall results of 97.74% accuracy is close to the state-of-the-art (98% [14], 99% [15], 97.3-99% [16]). The limitation is that we could not make consistent Γ-*embedding* DAG for the whole dataset. The process of DAG creation or clustering need improvement to enable a larger usage of the Γ-*embedding* DAG. However it shows that this mathematical object (Γ-*embedding* DAG) is useful for production applications.

5.9 Conclusion

This study contributes to the state-of-the-art by defining, formalizing and constructing a new representation for the common subsequences of a sequence set. It is called A_Γ DAG. We showed

that the A_Γ DAG contains all information about the common subsequences and is expressed in a very compact form. We succeed to design an algorithm that is able to build this construction for sequence without intra-repetitions. For other sequences, we have designed an algorithm that is able to construct a structure close to the solution. We have conducted an empirical study of the time and space complexity of the final algorithm. The computation time evolve as $\sqrt{n}.e^{0.043s-0.097a}$ where n is the number of sequences, s the size of the sequence and a the size of the alphabet, considering that each sequence has an equal size and each character an equal chance to appear. The memory footprint evolves as $e^{0.026s-0.059a}$. We assessed its utility for classification heuristics. With simple metrics we came to 97.74% accuracy for singling out clustered malware from other applications with the sequence of their Android API calls executed during dynamic analysis. While it does compete with state-of-the-art malware detection with machine learning, it shows that Γ-$embedding$ DAG conveys enough information for classification. The exploitation of this representation for data mining needs further research. The implementation of this algorithm is a part of *libmla* (*Machine Learning Algorithms Library*).

BIBLIOGRAPHY

[1] Wael H Gomaa and Aly A Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13) :13–18, 2013. 125

[2] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6) :341–343, 1975. DOI 10.1145/360825.360861. 126

[3] David W Mount. Bioinformatics : sequence and genome analysis. 2004. *Bioinformatics : Sequence and Genome Analysis*. 126

[4] Michelle Cheatham and Pascal Hitzler. String similarity metrics for ontology alignment. In *International Semantic Web Conference*, pages 294–309. Springer, 2013. 126

[5] J. P. D'Angelo and D. B. West. *Mathematical Thinking : Problem-Solving and Proofs, 2nd ed.* Prentice-Hall, 2000. DOI 10.4324/9781315044613. 127

[6] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2) :131–137, 1972. DOI 10.7146/math.scand.a-10849. 127, 132

[7] Jan Daciuk, Stoyan Mihov, Bruce W Watson, and Richard E Watson. Incremental construction of minimal acyclic finite-state automata. *Computational linguistics*, 26(1) :3–16, 2000. DOI 10.3115/1611533.1611538. 128, 132

[8] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin : Effective and explainable detection of android malware in your pocket. In *NDSS*, volume 14, pages 23–26, 2014. DOI 10.14722/ndss.2014.23247. 33, 48, 49, 59, 117, 137, 152, 154

[9] Paul Irolla and Alexandre Dey. The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques*, pages 1–5, 2018. 49, 117, 137, 152

[10] Paul Irolla and Eric Filiol. Glassbox : Dynamic analysis platform for malware android applications on real devices. *ForSE*, 2017. DOI 10.5220/0006094006100621. 70, 138, 154

[11] Ying-Dar Lin, Yuan-Cheng Lai, Chun-Nan Lu, Peng-Kai Hsu, and Chia-Yin Lee. Three-phase behavior-based detection and classification of known and unknown malware. *Security and Communication Networks*, 8(11) :2004–2015, 2015. DOI 10.1002/sec.1148. 138

[12] R Artusi, P Verderio, and E Marubini. Bravais-pearson and spearman correlation coefficients : meaning, test of hypothesis and confidence interval. *The International journal of biological markers*, 17(2) :148–151, 2002. 139

[13] Gilbert Saporta. *Probabilités, analyse des données et statistique*. Editions Technip, 2006. 108, 142

[14] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy : Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014*, page 3. ACM, 2014. DOI 10.1145/2556464.2556467. 143

[15] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer : Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013. DOI 10.1007/978-3-319-04283-1_6. 59, 143

[16] Suleiman Y Yerima, Sakir Sezer, and Igor Muttik. High accuracy android malware detection using ensemble learning. *IET Information Security*, 9(6) :313–320, 2015. DOI 10.1049/iet-ifs.2014.0099. 143

CHAPTER 6

Experimental Results

Contents

In this chapter, we put the neural network and feature extraction techniques presented earlier into practice. In the first section, we explain our feature selection process. In the second section, we optimize the hyperparaters of the neural network and we give final malware detection results. Last, we conclude the thesis.

6.1 Feature Selection

Feature selection is an important process in Machine Learning. Classification algorithm results are often limited by the quality of discriminative information given by the feature set. In this part we present a new algorithm for quickly evaluating a feature set. Then we optimize the feature from static and dynamic analysis.

6.1.1 Evaluation of Results

For all experiments, we use the *accuracy* as the main performance measure. It expresses the ratio of samples that are well classified, regarding their expected class (malware or benign):

- tp is the number of *True Positive*, i.e. the number of malware samples classified as such by the model.
- tn is the number of *True Negative*, i.e. the number of benign samples classified as such by the model.
- fp is the number of *False Positive*, i.e. the number of malware samples classified as benign by the model.
- fn is the number of *False Negative*, i.e. the number of benign samples classified as malware by the model.

$$\text{True positive rate :} \quad tpr = \frac{tp}{tp+fn}$$

$$\text{False positive rate :} \quad fpr = \frac{fp}{fp+tn}$$

$$\text{Accuracy}: \quad acc = \frac{tp + tn}{tp + tn + fp + fn}$$

6.1.2 FEA : Fast feature set Evaluation Algorithm

The feature set is usually optimized to enable a faster and better classification of downstream algorithms. Our feature set can take various forms and evaluating it requires to launch the training of a neural network. The neural network parameters also need to be optimized, and its parameters are dependent on the dimension and normalization procedure of the feature set. Moreover optimizing the feature set with the final neural network introduces biases in the results. And ultimately, the amount of necessary tests is too much time consuming. To handle this issue, we design a new algorithm, of linear space and time complexity regarding the feature set size, that is able to evaluate the discriminative quality of the feature set. This algorithm, named *FEA* (Fast Evaluation Algorithm), is a two-class classification algorithm which is extremely fast to train and execute. While its accuracy does not outmatch the state-of-the-art classification algorithm, it achieves results relatively close to them.

FEA speed allows to quickly evaluate the quality of a feature set. In the first part, we describe the algorithm and in the second part we show that it is as sensitive to the feature quality as any other classification algorithm. As it is as sensitive, it can serve the purpose of evaluation function.

Definition

Let $\mathscr{F} = \{\lambda_1, \cdots, \lambda_n\}$ be the finite *feature* set of n possible features from $\mathscr{S}$, the finite set of all *samples*. Let a *class* be a non-void set of samples. We define a *positive class*, C_+, and a *negative class*, C_-, such as $\{C_+, C_-\}$ is a partition of $\mathscr{S}$. Let S_+ and S_- be two known sets of *samples* such as $S_+ \subset C_+$ and $S_- \subset C_-$. Let ρ_i^s be the number of occurrences of the feature λ_i in the sample s. To simplify, we will always write ρ_i for ρ_i^s. We construct two vectors of feature occurrences, r_+ and r_- such as :

$$r_+ = \sum_{\forall s \in C_+} \begin{pmatrix} \rho_1 \\ \rho_2 \\ \vdots \\ \rho_n \end{pmatrix}$$

$$r_- = \sum_{\forall s \in C_-} \begin{pmatrix} \rho_1 \\ \rho_2 \\ \vdots \\ \rho_n \end{pmatrix}$$

Let $h : \mathbb{N} \to \mathbb{R}_+$ be a known transfer function. We can use :

$$h(x) = 1 - \frac{1}{1+x}$$

Let $H : \mathbb{N}^n \to \mathbb{R}_+^n$ be a transfer application such as :

$$H(X) = \begin{pmatrix} h(x_1) \\ h(x_2) \\ \vdots \\ h(x_n) \end{pmatrix}$$

For each *feature* λ_i there is an associated weight w_i. These weights come from the vector W that is calculated such as :

$$W = H(r_+) - H(r_-)$$

Once W is set, the training phase is done. For a *sample*, s, whose *class* is unknown we process α, the following decision criterion :

$$\alpha = \text{W.H}(r_s) = \sum_{i=1}^{n} w_i.h(\rho_i)$$

Finally :

- $s \in C_+$ if $\alpha > 0$
- $s \in C_?$ if $\alpha = 0$
- $s \in C_-$ else

Where $C_?$ denote the rare case where the decision criterion is unable to help us decide to which *class* the sample s belongs. For sake of implementation simplicity we can either decide that $C_? = C_+$ or $C_? = C_-$.

Several transfer functions have been tested :

$$h(x) = ln(1+x)$$

$$h(x) = \sqrt{1+x}$$

$$h(x) = 1 - \frac{1}{1+x}$$

The transfer function does not need necessarily to have an output value restricted in the interval $[0,1]$ but we found that using the last transfer function increase the accuracy.

Experimental Results

We evaluated *FEA* on 3 different problems :

- **Windows malware detection**. The *CSDMC2010_API* dataset [1] contains 766 sequences of windows API calls coming from the dynamic analysis of executables (320 benign and 446 malware programs). We use the frequency of the sequence trigrams as our base feature set.
- **Spam message detection**. The *ling-spam* dataset [2] contains 2893 mail messages (481 spam and 2893 benign messages). We preprocessed them by removing special characters (except for punctuation marks), normalizing spaces, removing some long words which were not part of common vocabulary, downcasing every characters and transforming any number series by special symbol (" [NUMBER] ") — as different numbers would be processed as different symbols, albeit the same conceptual object. Then we applied trigram analysis of the sequence of words (1 word is considered as 1 symbol in the sequence).
- **Android malware detection**. The *M0DROID* dataset [3] contains 400 Android applications (200 malware and 200 benign apks). We used our reverse engineering module presented in Chapter 3 to extract the sequence API calls, opcodes and strings (which are processed as sequences of characters). Then we use the combination of string bigrams, opcode trigrams and API call trigrams to build the initial feature set.

For each dataset, we set up 4 experiments :

- 10-fold cross validation on the raw feature set.
- 10-fold cross validation on the dataset of max-normalized frequencies. Each frequency is divided by the maximal frequency found in the feature vector, it spreads the values of the frequencies between 0 and 1.
- 10-fold cross validation on the dataset of max-normalized frequencies with feature vector normalization. The final feature vector is normalized (its norm is 1).

TABLE 6.1 – FEA test datasets

Alias	Dataset Name	Reference	Description	Feature
Dataset 1	CSDMC2010_API-dataset	[1]	Malware/Benign API call sequence	API trigram frequency
Dataset 2	ling-spam-dataset	[2]	Spam/Ham mails	Word trigram frequency
Dataset 3	M0DROID-dataset	[3]	Malware/Benign apk	String bigram frequency + Opcode trigram frequency + API trigram frequency

TABLE 6.2 – FEA test algorithms

Algorithm	Reference
FEA	This work
SMO	[4]
J48	[5]
BLR	[6]

TABLE 6.3 – FEA test feature sets

Alias	Description
FS1	Raw trigram frequencies
FS2	FS1 + max normalization
FS3	FS2 + normalized feature vectors
FS4	FS3 + feature selection (top 10%)

- 10-fold cross validation on the dataset of max-normalized frequencies with feature vector normalization. Then, we apply a feature selection phase. For each feature we calculate its appearance frequency (f) and its mutual information (mi). Last, we keep only the 10% most relevant features regarding the $mi \cdot f$ criterion.

The accuracy of *FEA* is compared with *SMO* (*Support Vector Machines using Sequential Minimal Optimization* [4], *J48* (C4.5 Decision Tree [5]), and *BLR* (Bayesian Logistic Regression [6]). All settings for this experimental part are summarized in Tables 6.1, 6.2, 6.3. Algorithm implementations come from *Weka*[1] with default parameters. To limit biases in the experiment, the same seed for the random split of the dataset has been used with k-fold testing.

Tables 6.4, 6.5, 6.6 and 6.7 present the accuracy results for all datasets and feature set configurations. In the Android malware detection experiment, the feature set 3 failed to produce meaningful results. Feature vectors are too large (>40,000 inputs) for normalization. In fact, feature values tend to zero with that number of features. And for several algorithms, the results were not exploitable so we discarded this experiment.

Figures 6.1, 6.2, 6.3 summarize the results for each dataset. The interpretation of the results is not obvious. *FEA* is reacting the most similarly to *SMO*. The top results are achieved for the same feature set. Moreover, results follow the same ups and downs — even if the amplitude is quite different. In the first experiment (dataset 1), *BLR* reacts badly to the vector normalization but other algorithms have similar reactions to the feature sets. So, *BLR* should be regarded as an outlier. In the second experiment, all algorithms react similarly to the change of feature set, even if the amplitude of the decrease *FEA* much bigger. In the last experiment, *FEA* results decrease from FS1 to FS2 like *SMO* and *J48*, but it increase from FS2 to FS4 like *SMO* and *BLR*. To summarize, *FEA*

1. https://www.weka.fr/

TABLE 6.4 – FEA accuracy results

	FS1	FS2	FS3	FS4
Dataset 1	**99.47%**	96.33%	93.33%	93.59%
Dataset 2	**96.85%**	78.38%	66.60%	58.73%
Dataset 3	86.84%	84.47%	N/A	**89.47%**

TABLE 6.5 – SMO accuracy results

%	FS1	FS2	FS3	FS4
Dataset 1	**99.21%**	98.82%	98.04%	98.04%
Dataset 2	**99.96%**	99.68%	93.46%	90.21%
Dataset 3	92.10%	91.84%	N/A	**92.36%**

TABLE 6.6 – J48 accuracy results

	FS1	FS2	FS3	FS4
Dataset 1	99.73%	**99.86%**	98.69%	99.34%
Dataset 2	**98.30%**	96.16%	95.12%	95.12%
Dataset 3	**88.94%**	86.84%	N/A	86.05%

TABLE 6.7 – BLR accuracy results

%	FS1	FS2	FS3	FS4
Dataset 1	99.47%	**99.60%**	83.15%	60.44%
Dataset 2	99.72%	**99.79%**	90.42%	88.52%
Dataset 3	88.15%	92.63%	N/A	**92.84%**

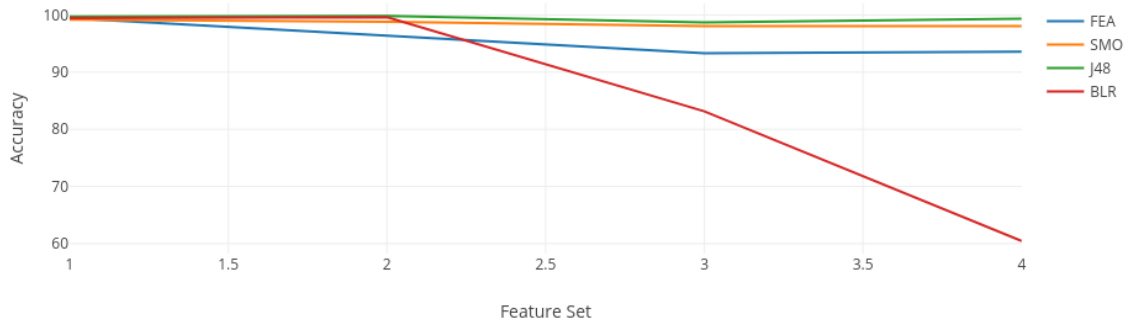


FIGURE 6.1 – Dataset 1 results

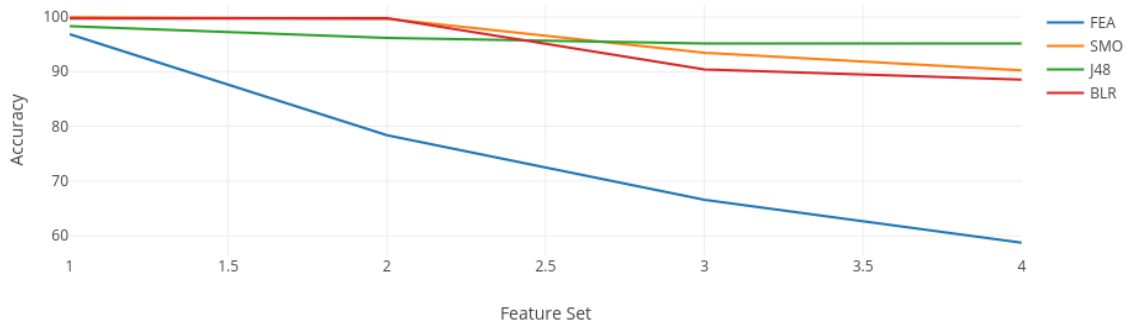


FIGURE 6.2 – Dataset 2 results

reacts to the change of feature set like the majority of the tested classification algorithm. We note that the accuracy of *FEA* drops dramatically when it is subject to vector normalization for a high vector size. Hence, overall it is a good algorithm for measuring quickly the feature set quality.

We benchmarked the processing time of every algorithm for 4 size of the feature set (Table 6.8. As *FEA* processing time is far behind every others and there is no ambiguity, we applied a simple benchmarking procedure (one benchmark for each algorithm and feature set size). We benchmarked the 10-fold testing time on a laptop with an *Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz* CPU and 2,900 samples for each experiment. *Weka* implementations are advantaged since they run on 2 CPU cores and *FEA* implementation use only 1 core. *FEA* is two to four orders of magnitude faster than its competitors.

TABLE 6.8 – Classifier processing time

FS size	**1313**	**2625**	**5353**	**13243**
FEA	193ms	239ms	294ms	432ms
SMO	9s :30ms	20s :830ms	44s :790ms	2min :30s :290ms
J48	1min :14s :160ms	2min :36s :460ms	7min :6s :330ms	19min :14s :390ms
BLR	6s :870ms	17s :10ms	36s :860ms	1min :45s

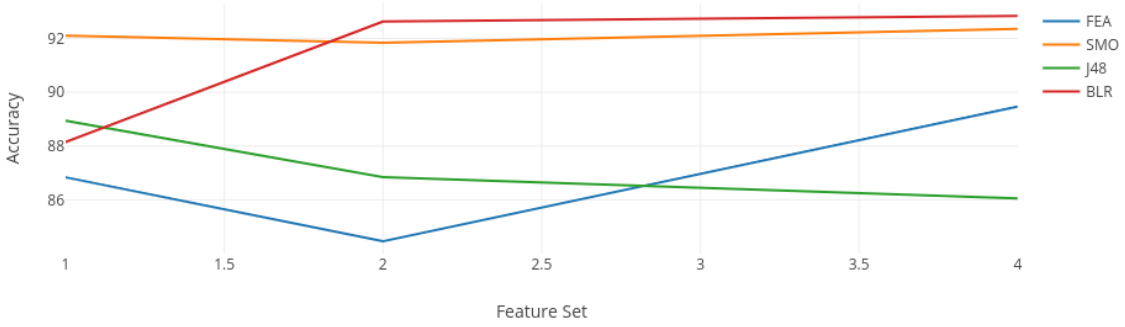


FIGURE 6.3 – Dataset 3 results

6.1.3 Feature Set

In this part we use the *FEA* algorithm to evaluate the best feature set to be used for malware detection. We have two sets of data, one coming from the static analysis of applications and the second one coming from the dynamic analysis.

We want to build the most optimized feature set for the neural network. Parameters are numerous, interdependencies are probable and time is limited. Hence, we employed a strategy borrowed from dynamic programming, which is based on common sense. From a choice of parameters, we test each one, choose the best one and carry the newly optimized feature set to the next parameter testing. At some points, we backtrack on some parameters if we suspect strong interdependencies. All accuracy measures have been achieved with *FEA* (cf. previous part), and with *SMO* when *FEA* failed to produce results.

Feature Set of Static Analysis

We ran *libmla-android-reverse* (cf. chapter 2) on 5585 malware samples from the *Drebin Dataset* [7] without duplicates [8] and 2206 benign applications from *F-droid*[2]. The tool extracts:

- The sequences of common library calls (cf. chapter 2, libmla-android-reverse). We single out common library calls from other java calls. In fact, common libraries found in the app package are not reversed.
- The sequences of Dalvik opcodes — operators of a bytecode operation.
- The sequences of strings from the DEX string table.
- A binary vector representing the presence of each permission and intent declared in the manifest.
- Metadata. The activity count, receiver count, service count, permission count, provider count, DEX file size, resources size. We also collect a complexity measure of the DEX file. We approximate the Kolmogorov complexity [9] with the ratio of the size of the compressed DEX over the size of the uncompressed DEX.

To build the best feature set, we assess the following parameters:

- Does limiting the common library list to the Android API only improve or hinder the results?
- Which window size of the N-Gram analysis produces the best results?
- Is it better that opcode sequences are the sequence of operations from a whole method or from each code block? As a reminder, code blocks are the smallest units of operation without memory jumps (cf. chapter 3, experimental part of the Glassbox section).

2. https://f-droid.org/en/

TABLE 6.9 – (FS1):Feature set optimization of common library call sequence

Exp. N°	Common libraries = Android API only?	N-Gram	Norm	FS Size	Accuracy
1	No	3gram	No	72565	0.808243
2	Yes	3gram	No	367999	0.724782
3	Yes	3gram	No	72565	0.724782
4	No	2gram	No	34495	0.803911
5	No	2gram	Max-freq	34495	0.9139
6	No	2gram	Max-freq, Std-norm	34495	0.919806
7	No	2gram	Max-freq, std-norm, vect-norm	34495	0.928862
8	No	2gram	Std-norm	34495	0.923482
9	**No**	**2gram**	**Max-freq, std-norm, vect-norm**	**5000**	**0.921514** [3]
10	No	2gram	Max-freq, std-norm, vect-norm	2000	0.910094

TABLE 6.10 – (FS2):Feature set optimization of opcode sequences

Exp. N°	Common libraries = Android API only?	Block or method opcode?	N-Gram	Norm	FS Size	Accuracy
1	No	Method	3gram	No	648	0.939887
2	No	Method	2gram	No	432	0.937262
3	No	Method	4gram	No	864	0.94015
4	No	Method	5gram	No	1080	0.9433
5	No	Method	6gram	No	1296	0.9454
6	Yes	Method	6gram	No	1296	0.948244 [4]
7	No	Block	6gram	No	1186	0.896181
8	No	Method	6gram	Max-freq	1296	0.941856
9	**No**	**Method**	**6gram**	**Std-norm**	1296	**0.945531**
10	No	Method	6gram	Vect-norm	1296	0.783042

- Which feature normalization fits the current problem? We implemented 3 normalization techniques:
  - *Max-freq*. Each feature is divided by the maximal feature in the vector.
  - *Vect-norm*. The vector is normalized and its norm becomes 1.
  - *Std-norm*. Each feature is divided by its standard deviation within the whole dataset. Hence, its variance becomes 1.
- The feature set size (noted *FS size*). We use the product of the *Mutual Information* [10] and the appearance frequency of each feature as a criterion to select the top ones.

Table 6.9 presents the feature set optimization of common library call sequences. Restricting the common libraries to the Android API only hinders the results. Here and in most of the experiments, as we will see later, bigram frequency is often above trigram frequency. Accuracy is greatly improved with full normalization — maximum frequency, feature variance normalization and vector normalization. Last, we find a balance between accuracy and processing time with a feature set size of 5000.

Table 6.10 presents the results of the feature set optimization of opcode sequences. *FEA* failed to produce results for the ngram of opcode sequences, so we used *SMO* instead which has relatively close behavior and an acceptable execution speed. Surprisingly, a large ngram window size improve the result. We notice that the feature size just slightly increase with the increase of the windows size. It means that there are not a wide amount of long opcode sequences, which is why increasing the window size is effective, the feature space does not become sparse. The variance normalization seems to be the only normalization technique that enhances the results.

3. This experiment appears to be the best compromise between accuracy and feature set size

TABLE 6.11 – (FS3):Feature set optimization of character sequences

Exp. N°	N-Gram	Norm	FS Size	Accuracy
1	3gram	No	1,521,578	N/A [5]
2	2gram	No	7,544 [6]	0.973487
3	2gram	Max-freq	7,544	0.973656
4	2gram	Max-freq, std-norm	7,544	0.973235
5	2gram	Max-freq, vect-norm	7,544	0.938312
6	**2gram**	**Max-freq**	**2000**	**0.969944** [7]

Table 6.11 presents the results of the feature set optimization of character sequences. Because of technical limitations the feature set with trigrams cannot be processed, so we directly moved to bigrams. We found that the max-frequency normalization is marginally helpful, but other normalizations are detrimental. Finally, we obtain a good compromise between feature set size and accuracy preservation, with a FS size of 2000. Last, the feature set of application metadata (FS4) and manifest data (FS5) do not need to be optimized, as there is no parameter to change.

Feature Set of Dynamic Analysis

We ran *Glassbox* [11] (cf. chapter 3) during several weeks to extract dynamic events from malware and benign applications. We use 929 malware samples from the *Drebin Dataset* [7] and 886 benign samples from *Fdroid*[8]. Dynamic analysis has many technical limitations as it is hard to make every application running properly. On this initial dataset only 756 malware samples and 535 benign samples produced meaningful results. We extracted the following data :

- The sequence of executed Java calls. From these initial sequences, we singled out the sequence of the Android API calls and the sequence of the rest of the calls — which have been defined by the developer or coming from external libraries.
- The sequence of system calls.
- The character sequence of the network communication. It is a full dump of the whole *http* protocol. It includes *http* traffic that is ciphered with *TLS* — *Glassbox* is connected to *MITM* wifi.

In this part, we want to assess the optimal feature set to classify applications. The following parameters are tested:

- Which sequence of Java calls is the best between the full sequence, the API call only sequence and the non-API call only sequence?
- Which window size of the N-Gram analysis produces the best results?
- Is the *tandem repeat removal* algorithm (cf. chapter 5) useful on these sequences? Tandem Repeats are noted *TR*.
- Which feature normalization fit the current problem? (cf.previous part).
- Which feature set size?

Note that this experiment has been made with the first version of *Glassbox*. The results could be improved, by the time we are writing this manuscript. We realized that applications, being highly multithreaded and multi-processed, write concurrent sequences of events into the same one. We

4. Opcodes from common libraries slightly increase the accuracy however, it decreases widely the accuracy from API call sequences so we chose to use common libraries
5. Even the feature reduction process is too time consuming, we did not evaluate 3grams for technical reasons
6. With 80% feature reduction
7. A large reduction of the FS size for a tiny loss of accuracy
8. https://f-droid.org/en/

TABLE 6.12 – (FS6):Feature set optimization of Java call sequences

Exp. N°	Data	N-Gram	TR	Norm	FS Size	Accuracy
1	Java calls	3gram	Yes	No	135,224	0.702419
2	Non-API calls	3gram	Yes	No	105,848	0.627618
3	API calls	3gram	Yes	No	4,1604	0.715323
4	Java calls	3gram	Yes	No	1,000	0.682258
5	Non-API calls	3gram	Yes	No	1,000	0.60861
6	API calls	3gram	Yes	No	1,000	0.696774
7	API calls	3gram	No	No	1,000	0.697581
8	API calls	4gram	No	No	1,000	0.68629
9	API calls	2gram	No	No	1,000	0.709677
10	API calls	2gram	No	Max-freq	1,000	0.833064
11	API calls	2gram	No	Vect-norm	1,000	0.805645
12	API calls	2gram	No	Vect-norm, std-freq	1,000	0.697581
13	API calls	2gram	No	Std-norm	1,000	0.858871
14	API calls	2gram	No	Max-freq, std-norm	1,000	0.875806
15	API calls	2gram	No	Vect-norm, std-norm	1,000	0.871774
16	API calls	2gram	No	Max-freq, std-norm	19,261	0.925
17	API calls	2gram	No	Max-freq, std-norm	11,001	0.930645
18	**API calls**	**2gram**	**No**	**Max-freq, std-norm**	**3862**	**0.925806** [9]

TABLE 6.13 – (FS7):Feature set optimization of system call sequences

Exp. N°	N-Gram	TR	Norm	FS Size	Accuracy
1	3gram	Yes	No	386	0.573985
2	4gram	Yes	No	523	0.58293
3	5gram	Yes	No	660	0.582068
4	2gram	Yes	No	249	0.588517
5	2gram	Yes	Std-norm	249	0.701813
6	2gram	Yes	Max-freq	249	0.590366
7	2gram	Yes	Std-norm, max-freq	249	0.71034
8	**2gram**	**Yes**	**Std-norm, max-freq, vect norm**	**249**	**0.775408**
9	2gram	Yes	Vect norm	249	0.584448
10	3gram	Yes	Std-norm, max-freq, vect norm	386	0.771538
11	3gram	Yes	Std-norm, max-freq	386	0.709565
12	2gram	No	Std-norm, max-freq, vect norm	249	0.748235

deeply enhanced the implementation of *Glassbox* to enable the following of every thread and process created. Now, one application is associated with dozens of Java call and system call sequences. Unfortunately, remaking the experiment would have pushed out of our deadlines.

Table 6.12 presents the results of the optimization of the feature set of Java calls. We start by testing the accuracy of Java call, API and non-API sequences with trigrams and without any feature reduction and normalization. The API sequences perform a little better here. As the feature set sizes are not equivalent, we select the top 1,000 most relevant features but the API sequences are still the best option. Removing tandem repeats slightly increase the results. It is expected since code loops can bloat the sequence with repeated substrings. Testing the N-Gram window size reveals that bigrams are the most suited. Next, we test the normalization to apply and the best option is normalizing the feature variance. Then we find empirically that a good feature set is around 4,000.

9. While the accuracy of the experiment 17 is above, the increase in feature size is not worth the gain

TABLE 6.14 – (FS8):Feature set optimization of network communication sequences

Exp. N°	N-Gram	TR	Norm	FS Size	Accuracy
1	**2gram**	**Yes**	**No**	**411**	**0.823349**
2	3gram	Yes	No	667	0.809524
3	2gram	Yes	Max-freq	411	0.801843
4	2gram	Yes	Std-norm	411	0.823349
5	2gram	Yes	Std-norm, vect-norm	411	0.71275
6	2gram	No	No	411	0.804916

Table 6.13 presents the results of the optimization of the feature set of system calls. Results are pretty bad until the variance normalization comes in play. We found that full normalization — with the maximum feature, variance and vector normalization — yields the best results. Interestingly, the tandem repeat removal hinders the results. It could be explained because system calls are a subdivision of a behavior unit (i.e. a function call). In fact a function call generally ends up calling many system calls at once. Hence, loops could breed very long tandem repeats that are not captured by our approximation. Moreover, the tandem repeats we remove are possibly part of a consistent behavior unit. In doing so we break the program core logic.

Table 6.13 presents the results of the optimization of network communication sequences. Here, the *FEA* algorithm failed to provide valid results (50% accuracy like coin tossing). The samples that effectively produce network requests that can capture is quite low. Only 439 malware and 212 benign application constituted the dataset for testing the network feature set. *FEA* starts to be usable after a certain number of samples, as the matrix r^+ and r^- need time to accumulate enough results — in particular, in presence of sparse data. So, we used the *SMO* algorithm. We did not find better than the feature set of bigram frequencies without any post-processing. Tandem repeat removal hinders the results but it was expected since network communication does not contain looping data, usually. So, here it just breaks the communication message.

6.2 Malware Detection with Static and Dynamic Analysis

Neural networks can be parametrized in numerous ways. In this part, we assess the following parameters :

- The number of hidden units.
- The use of *Adaptative Learning Rate* (noted *ALR*). It is a simple method to calibrate the learning rate during the training. The learning rate is modified regarding the error difference between two iterations of the learning algorithm. If the difference is positive, the learning rate is multiplied by 1.2. If the difference is negative, the learning rate is multiplied by 0.5.
- The use of *Controlled Variance* (noted *CV*) for initializing weights. See chapter 4.
- The use of the *Distributed Bias* (noted *DB*) method for initializing weights. See chapter 4.
- The use of *ADAM*[10] learning algorithm instead of the gradient descent.
- The use of Dropout (cf. chapter 4).
- The use of L2 regularization (cf. chapter 4).

We use the optimized feature sets from the previous section. Table 6.16 sumarize the feature sets we used to train the neural network. As stated before, the normalization procedure is very dependant on the algorithm used afterward. While *std-norm* and *vect-norm* are effective for most algorithms, we observed that they breed terrible performance for neural networks. Only *max-norm* is effective, so for all feature sets we apply the *max-norm* normalization.

10. kingma2014adam
11. Use of common library filtering

TABLE 6.15 – Training feature sets

Feature Set	Analysis	Data	Parameter	FS size
FS1	Static	API call sequence	2gram, CL, max-freq, std-norm, vect-norm	5000
FS2	Static	Opcode sequence	6gram, CL [11], std-norm, method opcode	1296
FS3	Static	Character sequence	2gram, max-freq	2000
FS4	Static	Manifest binary vector	N/A	476
FS5	Static	Metadata	N/A	8
FS6	Dynamic	API call sequence	2gram, no TR, max-freq, std-norm	3862
FS7	Dynamic	System call sequence	2gram, TR, std-norm, max-freq, vect norm	249
FS8	Dynamic	Network com. sequence	2gram, TR, no norm	411

Table 6.16 sumarize the results of neural network training for the different parameters. The results show that there is not a set of parameters that consistently deliver the best results for all feature sets. This is also the conclusion reached by Rojas in *Neural networks : a systematic introduction* [12]. Hence, neural network needs to be tweaked for every feature set. Despite this conclusion, we are able to draw general trends from the results :

- A number of hidden units equal to the *FS* size outmatch a number of hidden units of the square root the *FS* size. Doubling the number of hidden units often enhances the accuracy (4 cases over 7), but the tradeoff with the increase of processing time does not worth it. However, in the case of API call sequences it enhances the accuracy significantly.
- The use of *Adaptative Learning Rate* has mixed results. It enhances marginally the accuracy in 3 cases over 7 and hinders the accuracy marginally in the other cases. Hence, it has a minor impact.
- The use of *Controlled Variance* and *Distributed Bias* for initializing weights hinders the results in most of the cases. Hence, while providing strong mathematical boundaries, on our problem it should not be used.
- The use of *ADAM* [12] has mixed results. In 4 cases over 7, it significantly enhances the accuracy, in 2 cases over 7 it significantly hinders the accuracy and in the last case, the accuracy is hindered marginally.
- The use of Dropout in almost all cases significantly hinders the accuracy. The neural network we use is probably too small to take advantage of unit dropout.
- The use of L2 regularization significantly hinders the accuracy in all cases.

While multilayer perceptrons produce competitive accuracy, we have to admit they are no better than the *FEA* or the *SMO* algorithm on our problem. Even Deep Learning does not provide outmatching accuracy results (see. chapter 4) while requiring a large number of samples and processing time. As a conclusion, we observe that there is no perfect classification algorithm for Android malware detection. To build a robust antivirus solution, machine learning should be used in conjonction of pattern matching detection (i.e. the "classical" technique).

12. kingma2014adam
13. $P_0 = \{ h = \sqrt{n}$ where n is the FS size and h the number of hidden units, *SGD* learning algo., $U(-0.77; 0.77)$ weight initialization, $\mu = 0.001$, no regularization $\}$
14. Adaptative Learning Rate
15. Controlled Variance, cf.chapter 4
16. Distributed Bias, cf.chapter 4
17. L2 regularization, cf. chapter 4

TABLE 6.16 – Training feature sets

Neural network parameter	FS1	FS2	FS3	FS4	FS5	FS6	FS7	FS8
P_0 [13]	0.738813	0.92939	0.953537	0.928732	N/A	0.866935	0.677096	0.632779
$P_1 = P_0$ with $h = n$	0.916263	0.933195	0.956817	0.928338	N/A	0.874194	0.702932	0.638856
$P_2 = (P_1$ with ALR) [14]	0.916526	0.933326	**0.959182**	0.926894	N/A	0.875806	0.696363	0.636646
$P_3 = (P_2$ with CV) [15]	0.922825	0.910752	0.948286	0.92427	N/A	0.874194	0.685353	0.630566
$P_4 = (P_3$ with DB) [16]	0.918494	0.916659	0.946579	0.917183	N/A	0.873387	0.674382	0.628905
$P_5 = (P_3$ with ADAM)	0.915475	**0.940151**	0.910617	**0.932931**	N/A	0.881452	**0.732169**	**0.64106**
$P_6 = (P_5$ with L2) [17]	0.907075	0.886472	0.882925	0.916263	N/A	0.877419	0.680426	0.618423
$P_7 = (P_5$ with Dropout)	0.914293	0.912718	0.730561	0.922038	N/A	0.883064	0.710175	0.621726
$P_8 = (P_6$ with Dropout)	0.901431	0.845388	0.948286	0.899067	N/A	0.875	0.663915	0.612898
$P_8 = (P_8$ with $h = 2 \cdot n$)	**0.935556**	0.856154	0.935817	0.911408	N/A	**0.883871**	0.651773	0.611241

6.3 Conclusion

The objective of this thesis is to secure the mobile environment of the 3DNeuroSecure solution. After preliminary studies, which have been the subject of conferences and scientific articles, we have chosen to focus research on the detection of malware with neural networks.

This objective has brought me to the field of other topics such as the collection of application datasets, application analysis, feature selection and implementation of a neural network. I have covered all these topics in depth, and I have brought solutions in terms of software production, method, theory and algorithms with the intention of contributing to the current state-of-the-art. Thus, I provided :

- *Tarentula*. An Android application web crawler able to massively collect applications from shady sources.
- A reverse engineering module for Android application, faster than any other public tools.
- *Glassbox*. A controlled environment for the dynamic analysis on real devices. It was the first of his kind, since the publication other similar systems have emerged.
- *Smart Monkey*. A system that automatically and methodically tests the user interface of Android applications. I proved that it accesses more application code blocks than the *monkey* program that is usually used.
- A method for the feature selection process, with a wide range of statistical tests and data to gather an insight of features.
- *FEA*. An extremely fast bi-class classification algorithm that allows to assess the quality of a feature set.
- Algoritmic improvements of multilayer perceptrons, along with mathematical justifications.
- Implementation innovations that allow neural networks to run faster.
- *Libmla*. A library for efficient Machine Learning on CPU. Hence, it contains a fast implementation of a neural network and all tools a scientist or an engineer needs to use it.
- A critical analysis of the state-of-the-art of all topics covered.
- *The Embedding Antichain of Common Subsequences* and its applications. It is a new mathematical object that enables to represent and work with all common subsequences of a sequence set. I described this new problem fundamental mathematics and I presented an algorithmic solution to it which, even if it is still an approximation of the abstract solution, already enables to produce meaningful results on malware classification.

During this thesis, I approached an unexpected problem which is the research and the use of common subsequences of a sequence set. Wanting to find him a deterministic solution, I worked a week, this week has turned into a month, and this month into 8 months. I finally gave up this track, because I did not believe to find convincing results quickly and to continue would have endangered my thesis. Six months later, I met in a short interval a PhD in my field and a PhD in

physics. While discussing, I realized that one was looking for a solution to a problem very similar to mine that the other had also searched for a solution to a similar problem (but as it was not the main focus of his research at the time, he ended up giving up like me). They did not use the same words, the same concepts, but it was possible to transpose their problem into mine and vice versa, they were mathematically equivalent. This made me realize that my problem was a fundamental problem, that it could be useful to other people and that it was important, if not to solve it, to try to the end. This is why, *The Embedding Antichain of Common Subsequences*, is for me what I am most proud to present in this thesis. And it is, I believe, the beginning of a new field of study that has a wide range of applications.

After spending 3 years studying malware detection and machine learning, I think it is a heuristic detection that has a future and it is important to continue. Nevertheless, it will not replace the original quality of signature and rule detection techniques. This quality is the almost non-existent rate of false positive. It is therefore important that these new heuristic systems complement what already exists. A note on vulnerability detection, I do not think it is possible to have reliable heuristic detection. There will, probably, always need a manual analysis. This process is time consuming and expensive, so to have secure benign applications, it is necessary to have a dedicated market for reviewed applications, as it had been proposed with *DAVFI* (see introduction). The advantage of taking as a base object the sequence — sequence of opcodes, calls, characters, etc. — it is that the techniques that I have presented and implemented can be transposed to other types of data and operating systems. Hence, the method and implementations are applicable on Windows malware and network communications.

For someone who would like to resume my research, I advise dwelling on the *FEA* algorithm that has many potential and is far from being perfect. I thought for instance, using it to initialize the weights of a neural network. Because I was lacking the time, I have not deepened this path but I think it has a future. Moreover, the quick approximation of x^y (cf. chapter 4) could be easily finalized. Only needing very little calculation of this type in *libmla*, I did not go further. In this chapter, we have seen that each feature set should be trained with different neural network parameters. To train a neural network with all feature sets, I considered training different neural networks with different parameters and then combining their trained hidden layers connected to a new output layer. Indeed, neural networks are bad for managing features that have nothing to do with each other, in terms of variance, mean and range. And normalizing them, so that they have the same shape, is not always desirable in terms of accuracy as I have shown in this chapter. Also, the management of different types of features by different neural networks, seems to me a sensible option with high potential. And that would accomplish the feature normalization in a way that does not damage their discriminative quality. Finally, I think it is of paramount importance to explore the representation of the subsequences of a set of sequences. A purely mathematical approach would make sense and would probably unlock the algorithmic path I started.

BIBLIOGRAPHY

[1] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. A novel approach to detect malware based on api call sequence analysis. *International Journal of Distributed Sensor Networks*, 11(6) :659101, 2015. 149, 150

[2] Georgios Sakkis, Ion Androutsopoulos, Georgios Paliouras, Vangelis Karkaletsis, Constantine D Spyropoulos, and Panagiotis Stamatopoulos. A memory-based approach to anti-spam filtering for mailing lists. *Information retrieval*, 6(1) :49–73, 2003. 149, 150

[3] Mohsen Damshenas, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Ramlan Mahmud. M0droid : An android behavioral-based malware detection model. *Journal of Information Privacy and Security*, 11(3) :141–157, 2015. 149, 150

[4] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998. 150

[5] Ross Quinlan. *C4.5 : Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993. 150

[6] Alexander Genkin, David D. Lewis, and David Madigan. Large-scale bayesian logistic regression for text categorization. Technical report, DIMACS, 2004. 150

[7] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin : Effective and explainable detection of android malware in your pocket. In *NDSS*, volume 14, pages 23–26, 2014. DOI 10.14722/ndss.2014.23247. 33, 48, 49, 59, 117, 137, 152, 154

[8] Paul Irolla and Alexandre Dey. The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques*, pages 1–5, 2018. 49, 117, 137, 152

[9] Paul MB Vitanyi and Ming Li. *An introduction to Kolmogorov complexity and its applications*, volume 34. Springer Heidelberg, 1997. 152

[10] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on pattern analysis and machine intelligence*, 27(8) :1226–1238, 2005. 51, 153

[11] Paul Irolla and Eric Filiol. Glassbox : Dynamic analysis platform for malware android applications on real devices. *arXiv preprint arXiv :1609.04718*, 2016. 70, 138, 154

[12] Raúl Rojas. *Neural networks : a systematic introduction*. Springer Science & Business Media, 2013. 83, 89, 90, 102, 157

Appendix

Malware family classification

TABLE 17 – Android malware families

Family Name	Year	Motives	Techniques	Family Info	Sample Analysis	Alternative Names
fakeplayer	2010	SMS Malware	Fake App Drive-by download	i1.1		smssend
droidsms	2010	SMS Malware		Dunham and al., 2014		
fakeinst	2010	Spyware	Risk Tool	i2.1	s2.1	smstado fakebrows
tapsnake	2010	Scareware Spyware	Drive-by download	i3.1, i3.2		droidsnake
smsreplicator	2010	Spyware	Risk tool	Dunham and al., 2014	s4.1	
genimi	2010	Spyware	Backdoor Repackaging	i5.1		
adrd	2011	Spyware	Drive-by download	Dunham and al., 2014	s6.1	hongtoutou
pjapps	2011	SMS Malware Botnet	Backdoor Repackaging	Dunham and al., 2014	s7.1	
bgServ	2011	Malware	Backdoor Fake App Root exploit	Dunham and al., 2014	s8.1	
droiddream	2011	Spyware	Repackaging Root exploit	Dunham and al., 2014	s9.1	rootcager
droiddreamlight	2011	Spyware	Repackaging Root exploit	Dunham and al., 2014	s10.1	
walkinwat	2011	Spyware SMS malware	Repackaging	Dunham and al., 2014		pirater
zhash	2011	Malware	Backdoor Root exploit	Dunham and al., 2014		
zsone	2011	SMS Malware		Dunham and al., 2014	s11.1	
basebridge	2011	SMS Malware Spyware	Root exploit	Dunham and al., 2014		
droidkungfu	2011	Spyware	Backdoor Repackaging Root exploit Obfuscation	Dunham and al., 2014	s12.1	fokange fokonge gongfu
ggtracker	2011	SMS Malware Spyware		Dunham and al., 2014		
jsmshider	2011	SMS Malware	Backdoor	Dunham and al., 2014		xsider
plankton	2011	Spyware	Backdoor Repackaging	Dunham and al., 2014		tonclank
golddream	2011	SMS Malware	Backdoor Repackaging	Dunham and al., 2014	s13.1	spygold
gamblersms	2011	Spyware		Dunham and al., 2014	s14.1	
hipposms	2011	SMS Malware	Repackaging	Dunham and al., 2014		
lovetrap	2011	SMS Malware	Repackaging	Dunham and al., 2014		luvrtrap cosha
nickyspy	2011	Spyware	Backdoor Repackaging	Dunham and al., 2014	s15.1	nickybot
sndapps	2011	Spyware Adware	Repackaging	Dunham and al., 2014	s16.1	snadapps
zitmo	2011	Banking malware	Backdoor Fake App	Dunham and al., 2014	s17.1	
spitmo	2011	Banking malware	Backdoor Fake App	Dunham and al., 2014	s22.1	
dogwars	2011	SMS Malware	Repackaging	Dunham and al., 2014		dogwar dogowars
gingermaster	2011	Dropper Spyware	Backdoor Root exploit Repackaging	Dunham and al., 2014	s18.1	gingerbreak
anserverbot	2011	Dropper	Backdoor Drive-by download Root exploit Repackaging Obfuscation	Dunham and al., 2014	s19.1	anserver answerbot
droidcoupon	2011	Spyware Dropper	Backdoor Root exploit Repackaging Obfuscation	Dunham and al., 2014	s20.1	
jifake	2011	SMS malware	Fake App	Dunham and al., 2014		
asroot	2011	Malware	Root exploit	Jiang and al., 2013		
batterydoctor	2011	Scareware Spyware	Fake App	Dunham and al., 2014		fakedoc
beanbot	2011	SMS Malware	Backdoor	i32.1	s32.1	
coinpirate	2011	Spyware SMS malware	Repackaging Backdoor	Jiang and al., 2013		

Family Name	Year	Motives	Techniques	Family Info	Sample Analysis	Alternative Names
crusewin	2011	SMS malware	Backdoor	i33.1	s33.1	crusewind
droiddeluxe	2011	Dropper	Fake App Root exploit	i34.1	s34.1	
smspacem	2011	Activism SMS malware	Backdoor	i35.1	s35.1	endofday
fakeneflic	2011	Spyware	Fake App	Jiang and al., 2013	s36.1	fakenetflix
gone60	2011	Spyware	Risk tool	i37.1		gonein60seconds
kmin	2011	Spyware	Backdoor	i39.1, Jiang and al., 2013		ozotshielder
roguelemon	2011	SMS malware			s40.1	
roguesppush	2011	SMS malware		i41.1	s41.1	sppush autospsubscribe
yzhc	2011	SMS Malware			s42.1	yzhcsms uxipp wukong
adsms	2011	SMS Malware		i46.1		
antares	2011	Spyware		i47.1		antammi
fjcon	2011	Spyware Dropper SMS malware	Backdoor	i85.1		
flexispy	2011	Spyware	Risk tool	i86.1	s86.1	
foncy	2011	SMS malware	Fake App	i87.1	s87.1	
goldeneagle	2011	Spyware		i92.1		glodeagl
imlog	2011	Spyware		i97.1		ewalls
kidlogger	2011	Spyware	Risk tool	i99.1	s99.1	
lena	2011	Spyware Dropper Botnet	APT		s100.1	
mobiletx	2011	Spyware		i108.1		
mobinauten	2011	Spyware		i109.1		mobinaspy smshowu
netisend	2011	Spyware		i111.1		
rufraud	2011	SMS malware	Repackaging	i123.1		rufailed premiumtext
saiva	2011	SMS malware		i124.1		
smssniffer	2011	Spyware		i134.1		
typstu	2011	Spyware		i144.1		
updtnbot	2011	Malware	Backdoor	i145.1		
zeahache	2011	Malware	Root exploit	i153.1		
carrieriq	2011	Malware	Backdoor APT		i159.1	
fakebattscar	2011	Spyware		i160.1		
dropdialer	2011	SMS malware		i161.1		
meswatcherbox	2011	Spyware		i169.1		
airpush	2012	Adware		Dunham and al., 2014		smsstop
boxer	2012	SMS malware	Fake App Drive-by download	Dunham and al., 2014	s21.1	fakeinstaller
gappsusin	2012	Dropper	Fake App Drive-by download	Dunham and al., 2014		
leadbolt	2012	Adware		Dunham and al., 2014		
adwo	2012	Adware		Dunham and al., 2014		
counterclank	2012	Adware		Dunham and al., 2014		
smszombie	2012	Banking malware			s23.1	
notcompatible	2012	Botnet	Backdoor Ad fraud	Dunham and al., 2014	s24.1	proxytrajan nioserv
bmaster	2012	Spyware SMS Malware Botnet Dropper	Backdoor Root exploit Repackaging Drive-by download	Dunham and al., 2014	s25.1	rootsmart
luckycat	2012	Spyware		Dunham and al., 2014	s26.1	
drsheep	2012	Spyware		Dunham and al., 2014		
ackposts	2012	Spyware	Fake App	i44.1		
gpssmsspy	2012	Spyware	Risk tool		s38.1	
accutrack	2012	Spyware	Risk tool	i43.1		
acnetdoor	2012	Malware	Backdoor	i45.1		
arspam	2012	Activism SMS malware		i48.1		
carberp	2012	Banking malware		i55.1		
cawitt	2012	Spyware	Backdoor	i57.1	s57.1	twikabot ci4
coogos	2012	Spyware	Backdoor	i60.1		appleservice
dougalek	2012	Spyware		i62.1		dougaleaker
faceniff	2012	Spyware	Risk tool	i67.1		
fakeangry	2012	Spyware	Repackaging	i68.1		anzhu
fakeflash	2012	Scareware	Fake App	i73.1		
fakemart	2012	SMS malware	Fake App	i76.1	s76.1	
fakenotify	2012	SMS malware		i77.1	s77.1	
fakeregsms	2012	SMS malware	Obfuscation	i79.1	s79.1	
faketimer	2012	Spyware		i81.1		oneclickfraud
findandcall	2012	Spyware SMS malware		i83.1	s83.1	fidall findcall
gamex	2012	Spyware Dropper		i89.1		muldrop
iconsys	2012	Spyware		i96.1		
ksapp	2012	Spyware Dropper Botnet	Backdoor: JPMorgan case	i100.1		
loicdos	2012	Malware	Risk tool	i102.1		diydos
loozfon	2012	Spyware		i103.1	s103.1	lozfon
maistealer	2012	Spyware		i104.1		
mania	2012	SMS malware		i105.1		
moghava	2012	Malware		i109.1		
nandrobox	2012	SMS malware		i110.1		
opfake	2012	SMS message Spyware	Fake App	i112.1		
pdaspy	2012	Spyware	Risk tool	i113.1		
penetho	2012	Malware		i113.1		

Family Name	Year	Motives	Techniques	Family Info	Sample Analysis	Alternative Names
placms	2012	Botnet	Backdoor Keylogger	i116.1		
qicsomos	2012	SMS malware	Fake App	i119.1	s119.1	
seaweth	2012	SMS malware		i128.1		
smsspam	2012	SMS malware	Fake App		s135.1	
spyoo	2012	Spyware		i137.1	s137.1	
steek	2012	Adware	Repackaging	i139.1	s139.1	fatakr fakelottery fakeclick
tigerbot	2012	Spyware SMS malware	Backdoor	i141.1	s141.1	
tgloader	2012	SMS malware	Backdoor Root exploit	i143.1	s143.1	stiniter
updtkiller	2012	Spyware SMS malware	Backdoor AV evasion	i146.1		
vdloader	2012	Spyware	Backdoor	i150.1		
sumzand	2012	Spyware		i157.1		
fakevoice	2012	SMS malware		i158.1		
finfisher	2012	Spyware	Risk tool	i164.1		finspy
spyboo	2012	Spyware Adware		i165.1		
droidlive	2012	SMS malware Spyware	Backdoor		s166.1	
faketoken	2012	Banking malware Ransomware		i166.1		
gappll	2012	Dropper			s167.1	
jee	2012	SMS malware	Repackaging	i168.1		geofeebot
nyearleaker	2012	Spyware		i170.1		
pirates	2012	Spyware SMS malware		i171.1		cnnum
vidro	2012	SMS malware		i172.1	s172.1	
uranico	2012	Spyware		i173.1		
androrat	2012	Spyware	Backdoor Root exploit	i184.1	s184.1	
ggsmart	2013	Spyware SMS Malware Botnet Dropper	Backdoor Root exploit Drive-by download	Dunham and al., 2014		
defender	2013	Ransomware		Dunham and al., 2014, i27.1		
quadars	2013	Banking malware		Dunham and al., 2014		spy-abn
misosms	2013	Spyware SMS Malware	Fake App Obfuscation	Dunham and al., 2014		gidix krsms
technoreaper	2013	Spyware	Drive-by download	Dunham and al., 2014	s29.1	
fakerun	2013	Adware	Fake App	Dunham and al., 2014	s28.1	fakeapp adop
badnews	2013	Adware Dropper	Repackaging	Dunham and al., 2014		
obad	2013	SMS Malware Dropper	Backdoor Obfuscation	Dunham and al., 2014	s30.1, s30.2	
backflash	2013	Spyware	Backdoor	i50.1		crosate
beita	2013	Spyware		i52.1	s52.1	
chuli	2013	Spyware	Backdoor	i58.1	s58.1	
monad	2013	SMS malware Spyware	Backdoor	i66.1		extension nomead daemon
fakeav	2013	Scareware Spyware	Fake App	i69.1		
fakebank	2013	Banking malware Spyware SMS Malware	Backdoor	i70.1		
fakedaum	2013	Spyware		i71.1		
fakedefender	2013	Ransomware	Fake App	i72.1		
fakejoboffer	2013	Scareware		i74.1		
fakeplay	2013	Spyware Dropper SMS malware	Backdoor	i78.1	s78.1	
fakeupdate	2013	Dropper	Fake App	i82.1		apkqup
finspy	2013	Spyware	Fake App	i84.1		
feejar	2013	SMS malware		i88.1		fonefee
godwon	2013	Spyware		i91.1	s91.1	mobilespy
jollyserv	2013	Spyware SMS malware		i98.1		
mmarketpay	2013	Malware		i106.1		
pincer	2013	Spyware SMS malware	Backdoor	i115.1	s115.1	
repane	2013	Spyware SMS malware		i120.1		
roidsec	2013	Spyware		i122.1		sinpon
scipiex	2013	Spyware		i127.1		
skullkey	2013	Botnet	Repackaging	i130.1		
smsreg	2013	Spyware		i132.1		
smssilence	2013	Spyware	Backdoor Drive-by download	i133.1		smscatcher
tascudap	2013	Spyware Botnet	Backdoor	i140.1	s140.1	
tetus	2013	Spyware		i141.1		
uracto	2013	Spyware SMS malware		i147.1		
usbcleaver	2013	Spyware Hack tool		i148.1		
uten	2013	SMS malware Dropper	Root exploit	i149.1		
simhosy	2013	Spyware		i151.1		
zertsecurity	2013	Banking malware Spyware		i154.1	s154.1	
marcher	2013	Banking malware SMS malware	Repackaging	i196.1	s196.1	

Family Name	Year	Motives	Techniques	Family Info	Sample Analysis	Alternative Names
drivegenie	2014	Spyware	Backdoor	Dunham and al., 2014		
torec	2014	Spyware SMS Malware Banking malware	Backdoor	Dunham and al., 2014		torsm, acecard
oldboot	2014	Dropper	Backdoor APT Obfuscation	Dunham and al., 2014	s31.1	
droidpack	2014	Banking malware		Dunham and al., 2014		
avpass	2014	Spyware	AV evasion	i49.1		
badaccents	2014	Banking malware Spyware Dropper	Backdoor Tapjacking	Rasthofer and al., 2015	Rasthofer and al., 2015	
binv	2014	Banking malware		i53.1		
biige	2014	Spyware		i53.1		biigespy
booster	2014	Spyware	Fake App	i53.1		
code4hk	2014	Spyware		i59.1	s59.1	fitikser xsser mrat
sandrorat	2014	Spyware	Backdoor	i63.1	s63.1	sandroid rat droidjack
droidsheep	2014	Spyware	Risk tool	i64.1		
dsencrypt	2014	Banking malware Spyware	APT		s65.1, s65.2	
fakemarket	2014	Malware	Fake App Ad fraud		s75.1	
faketaobao	2014	Spyware		i80.1		smsspy
hehe	2014	Spyware	VM evasion	i94.1	s94.1	
oldboot	2014	Spyware Dropper	APT	i112.1		mouabad
podec	2014	Spyware Dropper Botnet SMS malware	Backdoor Fake App Repackaging Keylogger	i117.1		
raden	2014	Spyware Dropper Botnet SMS malware	Backdoor Keylogger	i120.1		
selfmite	2014	SMS malware	Repackaging	i129.1	s129.1	
ssucl	2014	Spyware SMS malware	Fake App Backdoor	i138.1		
wroba	2014	Banking malware Spyware Dropper SMS malware	Backdoor Fake App	i152.1		hijacktool
simplocker	2014	Ransomware Spyware	Backdoor Drive-by download Root exploit	i179.1	s179.1	
gunpoder	2014	Spyware SMS malware Adware	Backdoor	i183.1	s183.1	
ghostpush	2014	Spyware Adware Dropper	Root exploit	i190.1		
smsthief	2014	SMS malware Spyware		i191.1		
svpeng	2014	Banking malware Ransomware SMS malware		i193.1		
deathring	2014	Spyware	Backdoor	i200.1		
dendroid	2014	Spyware SMS malware	VM evasion	i202.1		
cajino	2015	Spyware	Backdoor	i54.1	s54.1	
gazon	2015	Spyware SMS malware		i90.1	s90.1	
hideicon	2015	Adware		i95.1		
mobidash	2015	Adware	Repackaging	i107.1		
poisoncake	2015	Spyware Dropper	Backdoor	i118.1		
samsapo	2015	SMS malware Dropper Spyware	Fake App	i125.1	s125.1	
socialpath	2015	Spyware SMS malware		i126.1		saveme
smack	2015	Spyware	Backdoor	i131.1		
titan	2015	Spyware SMS malware	Backdoor		s141.1, s141.2	
slembunk	2015	Banking malware	Drive-by download Repackaging APT Obfuscation	i176.1	s176.1	
bankosy	2015	Banking malware Spyware	Backdoor	i178.1	s178.1	
rootnik	2015	Spyware Dropper Adware	Root exploit Drive-by download Repackaging	i180.1	s180.1	
xiny	2015	Dropper Adware Spyware Banking malware	Backdoor	i189.1		
gooligan	2015	Dropper Spyware	Backdoor Root exploit Obfuscation	i195.1	s195.1	
viperrat	2015	Dropper Spyware	Backdoor	i203.1		
asacub	2015	Spyware Banking malware SMS malware	Backdoor	i205.1	s205.1	

Family Name	Year	Motives	Techniques	Family Info	Sample Analysis	Alternative Names
addown	2015	Adware Dropper Spyware	Backdoor Obfuscation	i217.1	s217.1	joymobile nativemob xavier
poriewspy	2015	Spyware		i230.1	s230.1	
zoopark	2015	Spyware		i241.1	s241.1	
gpspy	2016	Spyware		i53.1		
godless	2016	Spyware Dropper	Backdoor Root exploit Drive-by download	i156.1	Next section	
vikinghorde	2016	Botnet	Backdoor APT	i162.1		
hummingbad	2016	Botnet Dropper	Backdoor Ad fraud Root exploit	i163.1		
lokibot	2016	Spyware Dropper Adware	Root exploit Backdoor AV evasion	i175.1	s175.1	
cepsohord	2016	Dropper Botnet	Ad fraud	i177.1		
pawost	2016	Spyware SMS malware		i181.1	s181.1	
morder	2016	Spyware	Backdoor	i182.1		
twitoor	2016	Dropper Bootnet	Backdoor	i185.1	s185.1	
dresscode	2016	Botnet	Backdoor Ad fraud	i186.1		
calljam	2016	SMS malware		i187.1		
ztorg	2016	Spyware Dropper Adware	Backdoor Root exploit	i188.1		
exaspy	2016	Spyware Dropper	Backdoor Risk tool	i192.1		
gugi	2016	Banking malware		i194.1		
pluginphantom	2016	Spyware SMS malware	Backdoor	i195.1	s195.1	ihide
exobot	2016	Banking malware Spyware SMS malware Ransomware	Backdoor	i197.1	s197.1	exo
tordow	2016	Spyware Dropper SMS malware Banking malware Ransomware	Backdoor Drive-by download Repackaging Root exploit	i198.1	s198.1	
switcher	2016	Spyware	Backdoor	i199.1	s199.1	
swearing	2016	Spyware SMS malware Banking malware		i207.1		
chrysaor	2016	Keylogging SMS malware Spyware	Backdoor APT	i208.1	s208.1	pegasus
ewind	2016	Spyware Adware	Repackaging	i209.1	s209.1	
triada	2016	SMS malware Spyware	Root exploit Backdoor	i210.1		
bankbot	2017	Banking malware Spyware	Repackaging	i51.1		spy banker
copycat	2017	Adware Dropper	Backdoor Root exploit AV evasion APT	i61.1	s61.1	
slocker	2017	Ransomware		i101.1		
skyfin	2017	Malware		i201.1		
agentjl	2017	Dropper Spyware	Backdoor	i204.1		
skinner	2017	Spyware Adware	Backdoor Obfuscation	i205.1	s205.1	
chamois	2017	Adware Dropper	Obfuscation	i206.1		
smsvova	2017	Spyware SMS malware		i211.1	s211.1	
milkydoor	2017	Spyware	Backdoor	i212.1	s212.1	
charger	2017	Spyware Banking malware	Backdoor	i213.1	i213.1	
falseguide	2017	Botnet Adware	Backdoor Root exploit Drive-by download	i214.1	s214.1	
judy	2017	Botnet Adware	Ad fraud	i215.1	s215.1	
dvmap	2017	Malware	Root exploit	i216.1	s216.1	
spydialer	2017	Spyware	Backdoor Root exploit	i218.1	s218.1	
sonicspy	2017	Spyware SMS malware	Backdoor	i219.1		
ghostctrl	2017	Spyware SMS malware Ransomware	Backdoor Root exploit Obfuscation	i220.1	s220.1	
expensivewall	2017	Spyware SMS malware Adware	Repackaging	i221.1	s221.1	
sockbot	2017	Botnet	Backdoor	i222.1		
toastamigo	2017	Dropper	Backdoor Ad fraud	i223.1	s223.1	
tizi	2017	Dropper SMS malware Spyware	Backdoor Root exploit	i224.1	s224.1	
gnatspy	2017	Spyware	Backdoor Obfuscation	i225.1	s225.1	

Family Name	Year	Motives	Techniques	Family Info	Sample Analysis	Alternative Names
anubisspy	2017	Dropper Spyware	Backdoor	i226.1	s226.1	
catelites	2017	Banking malware Spyware		i227.1	s227.1	
herorat	2017	Spyware SMS message	Backdoor	i242.1	s242.1	
spybubble	2018	Spyware		i136.1		
reddrop	2018	Spyware Dropper SMS malware			s173.1	
mystertbot	2018	Ransomware Banking malware SMS malware	Backdoor Keylogger	i174.1	s174.1	
lightsout	2018	Adware		i228.1	s228.1	
ghostteam	2018	Dropper Adware	Repackaging	i229.1	s229.1	
rottensys	2018	Adware Dropper Botnet	Backdoor	i231.1	s231.1	
henbox	2018	Spyware	Repackaging	i232.1	s232.1	
telerat	2018	Spyware	Backdoor	i233.1	s233.1	
hiddnad	2018	Adware	Backdoor	i234.1	s234.1	
guerilla	2018	Spyware	Backdoor Drive-by download	i235.1	s235.1	
kevdroid	2018	Spyware	Backdoor Root exploit	i236.1	s236.1	
roamingmantis	2018	Banking malware Spyware	Backdoor	i237.1	s237.1	
desertscorpion	2018	Spyware	Backdoor Drive-by download	i238.1	s238.1	
xloader	2018	Spyware Banking malware Dropper	Drive-by download	i239.1	s239.1	
supercleanplus	2018	SMS malware	Backdoor	i240.1	s240.1	
triout	2018	Spyware		i243.1	s243.1	

Malware family information

Jiang and al., 2013 : Xuxian Jiang and Yajin Zhou. *A survey of android malware*. In Android Malware, pages 3–20. Springer, 2013

Dunham and al., 2014 : Ken Dunham, Shane Hartman, Manu Quintans, Jose Andre Morales, and Tim Strazzere. *Android Malware and Analysis*. Auerbach Publications, 2014

Rasthofer and al., 2015 : Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. *An investigation of the android/badaccents malware which exploits a new android tapjacking attack*. Technical report, Technical report, TU Darmstadt, Fraunhofer SIT and McAfee Mobile Research, 2015

i1.1 : https://www.kaspersky.com/about/press-releases/2010_first-sms-trojan-detected-for-smartphones-running-android

i2.1 : https://home.mcafee.com/VirusInfo/VirusProfile.aspx?key=366740#none

i3.1 : https://www.symantec.com/security-center/writeup/2010-081214-2657-99

i3.2 : https://www.infosecurity-magazine.com/blogs/tap-snake-infection-not-very-likely/

i5.1 : https://www.symantec.com/security-center/writeup/2011-010111-5403-99?tabid=-9

i27.1 : https://www.welivesecurity.com/wp-content/uploads/2018/02/Android_Ransomware_From_Android_Defender_to_Doublelocker.pdf

i32.1 : https://www.f-secure.com/v-descs/trojan_android_beanbot.shtml

i33.1 : https://www.symantec.com/security-center/writeup/2011-070301-5702-99?tabid=2

i34.1 : https://home.mcafee.com/virusinfo/virusprofile.aspx?key=582535

i35.1 : https://www.symantec.com/security-center/writeup/2011-052310-1322-99

i37.1 : https://www.symantec.com/security-center/writeup/2011-093001-2649-99?tabid=2

i39.1 : https://www.symantec.com/security-center/writeup/2011-091505-3230-99

i41.1 : https://fortiguard.com/encyclopedia/virus/2971291/android-roguesppush-a-tr

i43.1 : https://www.f-secure.com/sw-desc/monitoring-tool_android_accutrack.shtml

i44.1 : https://www.f-secure.com/v-descs/trojan_android_ackposts.shtml

i45.1 : https://www.symantec.com/security-center/writeup/2012-051611-4258-99

i46.1 : https://www.symantec.com/security-center/writeup/2011-051313-4039-99

i47.1 : https://fortiguard.com/encyclopedia/virus/3313414/android-antammi-a-tr

i48.1 : https://androidcommunity.com/android-arspam-is-the-latest-malware-threat-says-symantec-20111230/

i49.1 : https://www.f-secure.com/v-descs/trojan_android_avpass_c.shtml

i50.1 : https://www.symantec.com/security-center/writeup/2013-091714-0427-99

i51.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/bankbot-spybanker?rq=bankbot

i52.1 : https://www.symantec.com/security-center/writeup/2013-110111-1829-99

i53.1 : https://forensics.spreitzenbarth.de/android-malware/

i54.1 : https://www.symantec.com/en/ca/security-center/writeup/2015-040210-3746-99

i55.1 : https://securelist.com/carberp-in-the-mobile/57658/

i56.1 : https://www.f-secure.com/v-descs/trojan_android_cawitt.shtml

i57.1 : https://www.symantec.com/security-center/writeup/2012-062614-5813-99

i58.1 : https://www.symantec.com/security-center/writeup/2013-032617-1604-99

i59.1 : https://www.symantec.com/security-center/writeup/2014-093015-2830-99

i60.1 : http://www.virusradar.com/en/Android_AppleService.A/description

i61.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/copycat?rq=copycat

i62.1 : https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/androidos_dougalek.a

i63.1 : https://www.symantec.com/security-center/writeup/2014-110720-2146-99

i64.1 : https://www.symantec.com/security-center/writeup/2014-031014-3628-99

i66.1 : http://www.virusradar.com/en/Android_Damon.A/description
i67.1 : https://home.mcafee.com/virusinfo/virusprofile.aspx?key=1574799
i68.1 : https://www.f-secure.com/v-descs/trojan_android_fakeangry.shtml
i69.1 : https://www.symantec.com/security-center/writeup/2007-101013-3606-99
i70.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/fakebank
i71.1 : https://www.symantec.com/security-center/writeup/2013-061813-3630-99
i72.1 :
https://njccic.squarespace.com/threat-profiles/android-malware-variants/fakedefender?rq=fakedefender
i73.1 : https://www.f-secure.com/v-descs/trojan_android_fakeflash_c.shtml
i74.1 : https://fortiguard.com/encyclopedia/virus/5255022/android-fakejob-a-tr
i76.1 : https://fortiguard.com/encyclopedia/virus/4125290/android-fakemart-b-tr
i77.1 : https://www.symantec.com/security-center/writeup/2012-011302-3052-99
i78.1 : https://fortiguard.com/encyclopedia/virus/6050230/android-fakeplay-c-tr-spy
i79.1 : https://www.f-secure.com/v-descs/trojan_android_fakeregsms.shtml
i80.1 : https://www.symantec.com/security-center/writeup/2014-012106-4013-99
i81.1 : https://www.f-secure.com/v-descs/trojan_android_faketimer.shtml
i82.1 : https://www.symantec.com/security-center/writeup/2013-081914-5637-99
i83.1 : https://www.symantec.com/security-center/writeup/2014-031020-2906-99
i84.1 : https://fortiguard.com/encyclopedia/virus/4522213/android-finspy-a-tr-spy
i85.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/Fjcon/
i86.1 : https://www.symantec.com/security-center/writeup/2011-122006-4805-99
i87.1 : https://fortiguard.com/encyclopedia/virus/3320403/android-foncy-a-tr
i88.1 : http://www.virusradar.com/en/Android_TrojanSMS.Feejar.B/description
i89.1 : https://fortiguard.com/encyclopedia/virus/5229603
i90.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/gazon
i91.1 : https://www.symantec.com/security-center/writeup/2013-091017-1833-99
i92.1 : https://us.norton.com/online-threats/android.goldeneagle-2011-090110-3712-99-writeup.html
i93.1 : https://www.fortiguard.com/encyclopedia/virus/7174444
i94.1 : https://www.symantec.com/security-center/writeup/2014-012211-0020-99
i95.1 : https://news.drweb.com/show/review/?lng=en&i=9264
i96.1 : https://www.symantec.com/security-center/writeup/2012-081309-0341-99
i97.1 : https://fortiguard.com/encyclopedia/virus/3303619
i98.1 : https://www.symantec.com/security-center/writeup/2013-090311-4533-99
i99.1 : https://www.symantec.com/security-center/writeup/2011-122014-1927-99
i100.1 : http://androidmalwaredump.blogspot.com/2013/01/androidtrojmdk-aka-androidksapp.html
i101.1 : https://www.f-secure.com/v-descs/trojan_android_slocker.shtml
i102.1 : https://www.symantec.com/security-center/writeup/2012-022002-2431-99
i103.1 : https://www.symantec.com/security-center/writeup/2012-082005-5451-99
i104.1 : https://www.f-secure.com/v-descs/trojan_android_maistealer.shtml
i105.1 : https://www.f-secure.com/v-descs/trojan_android_mania.shtml
i106.1 : https://fortiguard.com/encyclopedia/virus/4721252
i107.1 : https://blog.malwarebytes.com/detections/android-adware-mobidash/
i108.1 : https://fortiguard.com/encyclopedia/virus/3300489/android-mobiletx
i109.1 : https://fortiguard.com/encyclopedia/virus/3650905/android-moghava-a-tr
i110.1 : https://www.f-secure.com/v-descs/trojan_android_nandrobox.shtml
i111.1 : https://fortiguard.com/encyclopedia/virus/2959807
i112.1 : https://www.f-secure.com/v-descs/trojan_android_oldboot_a.shtml
i113.1 : https://www.f-secure.com/sw-desc/monitoring-tool_android_pdaspy.shtml
i114.1 : https://www.symantec.com/security-center/writeup/2012-100110-3614-99
i115.1 : https://fortiguard.com/encyclopedia/mobile/4976097/android-pincer-a-tr-spy
i116.1 : https://www.fortiguard.com/encyclopedia/virus/3632263
i117.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/podec?rq=podec
i118.1 : https://www.symantec.com/security-center/writeup/2015-010610-0726-99
i119.1 : https://www.f-secure.com/v-descs/trojan_android_qicsomos.shtml
i120.1 : https://fortiguard.com/encyclopedia/virus/6275158
i121.1 : https://www.symantec.com/security-center/writeup/2013-090411-5052-99
i122.1 : https://www.f-secure.com/v-descs/trojan_android_roidsec.shtml
i123.1 : https://blog.malwarebytes.com/threats/sms-trojan/
i124.1 : https://www.f-secure.com/v-descs/trojan_android_saiva.shtml
i125.1 : https://www.f-secure.com/v-descs/worm_android_samsapo.shtml
i126.1 : https://www.carmelowalsh.com/2015/01/saveme-app-actually-socialpath-malware-android-phones/
i127.1 : https://www.symantec.com/security-center/writeup/2013-100814-4702-99
i128.1 : https://fortiguard.com/encyclopedia/virus/3650039
i129.1 : https://www.symantec.com/security-center/writeup/2014-101013-4717-99
i130.1 : https://www.symantec.com/security-center/writeup/2013-072322-5422-99
i132.1 : https://www.f-secure.com/sw-desc/riskware_android_smsreg.shtml
i133.1 : https://fortiguard.com/encyclopedia/virus/4882554/android-smsilence-a-tr-spy
i134.1 : https://www.symantec.com/security-center/writeup/2011-071108-3626-99
i136.1 : https://fortiguard.com/encyclopedia/virus/7874695
i137.1 : https://fortiguard.com/encyclopedia/virus/4103699/android-spyoo-a-tr-spy
i138.1 : https://www.f-secure.com/v-descs/trojan-spy_android_sscul.shtml
i139.1 : https://fortiguard.com/encyclopedia/virus/3458224/android-steek-a-tr
i140.1 : https://fortiguard.com/encyclopedia/virus/4533396/android-tascudap-a-tr
i141.1 : https://www.f-secure.com/v-descs/trojan_android_tetus.shtml
i141.1 : https://www.symantec.com/security-center/writeup/2012-041010-2221-99

i143.1 : https://www.symantec.com/security-center/writeup/2012-030903-5228-99
i144.1 : https://origin-home.mcafee.com/VirusInfo/VirusProfile.aspx?key=724078
i145.1 : https://www.symantec.com/security-center/writeup/2012-041611-4136-99
i146.1 : https://home.mcafee.com/virusinfo/virusprofile.aspx?key=2345748
i147.1 : https://www.symantec.com/security-center/writeup/2013-031805-2722-99
i148.1 : https://www.symantec.com/security-center/writeup/2013-062010-1818-99
i149.1 : https://www.symantec.com/security-center/writeup/2013-092316-4752-99
i150.1 : https://www.symantec.com/security-center/writeup/2012-080209-1420-99
i151.1 : https://www.symantec.com/security-center/writeup/2013-061013-3955-99
i152.1 : https://fortiguard.com/encyclopedia/virus/6275091
i153.1 : https://www.symantec.com/security-center/writeup/2011-032309-5042-99
i154.1 : https://www.symantec.com/security-center/writeup/2013-050820-4100-99
i155.1 : https://www.f-secure.com/v-descs/trojan_android_zsone.shtml
i156.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/godless
i157.1 : https://www.symantec.com/security-center/writeup/2012-080308-2851-99
i158.1 : https://www.symantec.com/security-center/writeup/2012-040510-3249-99
i160.1 : https://www.f-secure.com/v-descs/trojan_android_fakebattscar.shtml
i161.1 : https://www.symantec.com/security-center/writeup/2012-070909-0726-99
i162.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/viking-horde?rq=viking%20horde
i163.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/hummingbad?rq=hummingbad
i164.1 : https://www.f-secure.com/v-descs/trojan-spy_w32_finspy_a.shtml
i165.1 : https://fortiguard.com/encyclopedia/virus/3734287/riskware-spyboo-android
i166.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/faketoken?rq=faketoken
i168.1 : http://virus.nq.com/en/android/a.payment.GeoFeeBot.c/
i169.1 : https://www.symantec.com/security-center/writeup/2011-111612-2736-99
i170.1 : https://www.symantec.com/security-center/writeup/2012-010514-0844-99
i171.1 : https://www.f-secure.com/v-descs/trojan_android_pirates.shtml
i172.1 : https://fortiguard.com/encyclopedia/virus/4157003/android-vidro-a-tr
i173.1 : https://www.symantec.com/security-center/writeup/2012-052803-3835-99
i174.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/mysterybot?rq=mysterybot
i175.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/lokibot?rq=lokibot
i176.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/slembunk
i177.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/cepsohord
i178.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/bankosy
i179.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/simplocker
i180.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/rootnik
i181.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/pawost
i182.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/morder-a
i183.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/gunpoder
i184.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/androrat
i185.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/twitoor
i186.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/dresscode
i187.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/calljam
i188.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/ztorg
i189.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/xiny
i190.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/ghost-push
i191.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/sms-thief
i192.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/exaspy
i193.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/svpeng
i194.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/gugi
i195.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/gooligan
i196.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/marcher
i197.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/exo-exobot
i198.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/tordow
i199.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/switcher
i200.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/deathring
i201.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/skyfin
i202.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/dendroid
i203.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/viperrat
i204.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/agent-jl
i205.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/asacub
i206.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/chamois
i207.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/swearing
i208.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/chrysaor
i209.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/ewind
i210.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/triada
i211.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/smsvova
i212.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/milkydoor
i213.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/chargerb
i214.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/falseguide
i215.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/judy
i216.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/dvmap
i217.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/addown
i218.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/spydealer
i219.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/sonicspy

i220.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/ghostctrl
i221.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/expensivewall
i222.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/sockbot
i223.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/toast-amigo
i224.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/tizi
i225.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/gnatspy
i226.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/anubisspy
i227.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/catelites
i228.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/lightsout
i229.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/ghostteam
i230.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/poriewspy
i231.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/rottensys
i232.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/henbox
i233.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/telerat
i234.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/hiddnad
i235.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/guerilla
i236.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/kevdroid
i237.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/roaming-mantis
i238.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/desert-scorpion
i239.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/xloader
i240.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/super-clean-plus
i241.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/zoopark
i242.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/herorat
i243.1 : https://www.cyber.nj.gov/threat-profiles/android-malware-variants/triout

Sample analysis links

s2.1 : https://www.f-secure.com/weblog/archives/00002278.html
s4.1 : http://www.talkandroid.com/19466-sms-replicator-for-android-will-secretly-forward-texts/
s6.1 : http://www.antiy.net/media/reports/android_adrd_analysis.pdf
s7.1 : https://www.symantec.com/connect/blogs/android-threats-getting-steamy
s8.1 : https://www.symantec.com/connect/blogs/androidbgserv-found-fake-google-security-patch-part-ii
s9.1 : http://www.antiy.net/p/android-apps-repackaged/
s10.1 : https:
//blog.trendmicro.com/trendlabs-security-intelligence/analysis-of-droiddreamlight-android-malware/
s11.1 : https://www.f-secure.com/weblog/archives/00002373.html
s12.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu.html
s13.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/new-android-malware-on-the-road-goldd
ream-catcher/
s14.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/GamblerSMS/
s15.1 : https://www.trustwave.com/Resources/SpiderLabs-Blog/NickiSpy-C---Android-Malware-Analysis--Demo/
s16.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/SndApps/
s17.1 : https://www.strazzere.com/blog/2012/08/android-zitmo-analysis-now-you-see-my-now-you-dont/
s18.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/GingerMaster/
s19.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/pubs/AnserverBot_Analysis.pdf
s20.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/DroidCoupon/
s21.1 : https://www.welivesecurity.com/wp-content/media_files/SMS_Trojan_Whitepaper.pdf
s22.1 : https://forensics.spreitzenbarth.de/2011/12/06/detailed-analysis-of-android-spitmo/
s23.1 : http://cvo-lab.blogspot.com/2012/08/android-malware-smszombie-in-depth.html
s24.1 : http://www.tmcnet.com/tmc/whitepapers/documents/whitepapers/2014/9594-notcompatibleandroid-web-pro
xy-bot-malware-analysis-report.pdf
s25.1 : https://forensics.spreitzenbarth.de/2012/02/12/detailed-analysis-of-android-bmaster/
s26.1 :
https://citizenlab.ca/2013/04/permission-to-spy-an-analysis-of-android-malware-targeting-tibetans/
s28.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/fake-version-of-temple-run-unearthed
-in-the-wild/
s29.1 : https://www.webroot.com//blog/2013/05/10/android-technoreaper-downloader-found-on-google-play/
s30.1 : https://securityintelligence.com/diy-android-malware-analysis-taking-apart-obad-part-1/
s30.2 : http://joe4mobile.blogspot.com/2013/06/analyzing-obada-aka-most-sophisticated.html
s31.1 : http://blogs.360.cn/post/analysis_of_oldboot_b_en.html
s32.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/BeanBot/
s33.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/android-malware-acts-as-an-sms-relay/
s34.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/DroidDeluxe/
s35.1 : https://www.symantec.com/connect/blogs/android-threat-set-trigger-end-days-or-day-s-end
s36.1 : https:
//www.symantec.com/connect/blogs/will-your-next-tv-manual-ask-you-run-scan-instead-adjusting-antenna
s38.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/beta-version-of-spytool-app-for-andro
id-steals-sms-messages/
s40.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/RogueLemon/
s41.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/RogueSPPush/
s42.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/YZHCSMS/
s52.1 : https://www.jamesejr.com/android-beita-malware-analysis/
s54.1 : http://b0n1.blogspot.com/2015/03/remote-administration-trojan-using.html

s57.1 : https://securelist.com/mobile-malware-evolution-part-6/36996/
s58.1 : http://joe4mobile.blogspot.com/2013/08/learning-from-chulia-android-trojan.html
s59.1 : https://malware.lu/articles/2014/09/29/analysis-of-code4hk.html
s61.1 : https://www.checkpoint.com/downloads/resources/copycat-research-report.pdf
s63.1 : http://maldr0id.blogspot.com/2014/09/sandroid-rat-analysis-part-i-synthetic.html
s65.1 : https://borelenzo.github.io/malwares/dsencrypt/
s65.2 : https://www.fireeye.com/blog/threat-research/2014/06/what-are-you-doing-dsencrypt-malware.html
s75.1 : https://lightsec.wordpress.com/2014/03/21/android-malware-analyzing-the-fakemarket-trojan/
s76.1 : https://hackmag.com/uncategorized/droidbox-for-dynamic-malware-analysis/
s77.1 : https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/androidos_fakenotify.a
s78.1 : https://blog.malwarebytes.com/cybercrime/2017/03/mobile-menace-monday-facebook-lite-infected-wit
h-spy-fakeplay/
s79.1 : https://forensics.spreitzenbarth.de/2012/02/03/detailed-analysis-of-android-fakeregsms-b/
s83.1 : https://securelist.com/find-and-call-leak-and-spam-57/33544/
s86.1 : https://www.randhome.io/blog/2017/04/23/lets-talk-about-flexispy/
s87.1 : https://code.google.com/archive/p/androguard/wikis/AndroidMalwareAnalysis.wiki
s90.1 : https://www.adaptivemobile.com/blog/worm-gazon-want-gift-card-get-malware
s91.1 : https://versprite.com/blog/application-security/android-infostealer-godwon-analysis/
s94.1 : https:
//www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html
s99.1 : https://kidlogger.net/kidlogger-android-features.html
s100.1 :
https://blog.lookout.com/security-alert-legacy-makes-another-appearance-meet-legacy-native-(lena)
s103.1 : https://www.attacker-domain.com/2012/08/loozfon-android-malware-analysis.html
s115.1 : http://joe4mobile.blogspot.com/2013/05/analyzing-pincera-sms-based-trojan-for.html
s119.1 : https://forensics.spreitzenbarth.de/2012/01/17/detailes-analysis-of-android-qicsomos/
s125.1 : https://www.welivesecurity.com/2014/04/30/android-sms-malware-catches-unwary-users/
s129.1 : https://www.adaptivemobile.com/blog/take-two-selfmite-b-hits-the-road
s131.1 : http://blog.avlyun.com/2015/03/2222/remote-controll-trojan-with-smack-technique/
s135.1 : https://blog.cloudmark.com/2012/12/16/android-trojan-used-to-create-simple-sms-spam-botnet/
s137.1 : https://fortiguard.com/encyclopedia/virus/4103699/android-spyoo-a-tr-spy
s139.1 : https://fortiguard.com/encyclopedia/virus/3458224/android-steek-a-tr
s140.1 : https://fortiguard.com/encyclopedia/virus/4533396/android-tascudap-a-tr
s141.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/TigerBot/
s142.1 : https://versprite.com/blog/application-security/android-titan-sms-trojan-analysis-part-one/
s142.2 : https://avlab.pl/en/android-trojan-titan-undetectable-most-antivirus-software
s143.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/TGLoader/
s154.1 : https://blog.lookout.com/zertsecurity/
s155.1 : https://www.f-secure.com/v-descs/trojan_android_zsone.shtml
s159.1 : http://androidsecuritytest.com/features/logs-and-services/loggers/carrieriq/carrieriq-part2/
s162.1 :
https://blog.checkpoint.com/2016/05/09/viking-horde-a-new-type-of-android-malware-on-google-play/
s166.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/DroidLive/
s167.1 : https://www.csc2.ncsu.edu/faculty/xjiang4/GappII/
s172.1 : https://fortiguard.com/encyclopedia/virus/4157003/android-vidro-a-tr
s173.1 : https://www.wandera.com/reddrop-malware/
s174.1 : https:
//www.threatfabric.com/blogs/mysterybot__a_new_android_banking_trojan_ready_for_android_7_and_8.html
s175.1 : https://www.threatfabric.com/blogs/lokibot_the_first_hybrid_android_malware.html
s176.1 : https://www.fireeye.com/blog/threat-research/2015/12/slembunk_an_evolvin.html
s178.1 : https://www.symantec.com/connect/blogs/androidbankosy-all-ears-voice-call-based-2fa
s179.1 : https://www.welivesecurity.com/2014/06/04/simplocker/
s180.1 : https://researchcenter.paloaltonetworks.com/2015/12/rootnik-android-trojan-abuses-commercial-roo
ting-tool-and-steals-private-information/
s181.1 : https://blog.malwarebytes.com/cybercrime/mobile/2016/06/google-talk-used-to-make-malicious-phone
-calls-android-trojan-pawost/
s183.1 : https://researchcenter.paloaltonetworks.com/2015/07/new-android-malware-family-evades-antivirus-d
etection-by-using-popular-ad-libraries/
s184.1 : https://www.symantec.com/connect/blogs/remote-access-tool-takes-aim-android-apk-binder
s185.1 : https://www.welivesecurity.com/2016/08/24/first-twitter-controlled-android-botnet-discovered/
s195.1 : https://blog.checkpoint.com/2016/11/30/1-million-google-accounts-breached-gooligan/
s196.1 : https://blog.checkpoint.com/2016/04/28/marcher-marches-on-the-anatomy-of-a-banker-malware/
s197.1 :
https://www.bleepingcomputer.com/news/security/new-exo-android-trojan-sold-on-hacking-forums-dark-web/
s198.1 : https://blog.comodo.com/comodo-news/comodo-warns-android-users-of-tordow-v2-0-outbreak/
s199.1 : https://www.kaspersky.com/blog/switcher-trojan-attacks-routers/13771/
s205.1 : https://www.kaspersky.com/blog/asacub-trojan/11108/
s208.1 : https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-android-technical-analysis.pdf
s209.1 : https://researchcenter.paloaltonetworks.com/2017/04/unit42-ewind-adware-applications-clothing/
s211.1 : https://www.zscaler.com/blogs/research/android-spyware-smsvova-posing-system-update-play-store
s212.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/dresscode-android-malware-finds-succe
ssor-milkydoor/
s213.1 : https://blog.checkpoint.com/2017/05/16/the-mobile-banker-threat-from-end-to-end/
s214.1 : https://blog.checkpoint.com/2017/04/24/falaseguide-misleads-users-googleplay/
s215.1 : https:

//blog.checkpoint.com/2017/05/25/judy-malware-possibly-largest-malware-campaign-found-google-play/
s216.1 : https://securelist.com/dvmap-the-first-android-malware-with-code-injection/78648/
s217.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/analyzing-xavier-information-stealing-ad-library-android/
s218.1 :
https://researchcenter.paloaltonetworks.com/2017/07/unit42-spydealer-android-trojan-spying-40-apps/
s220.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/android-backdoor-ghostctrl-can-silently-record-your-audio-video-and-more/
s221.1 : https:
//blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/
s223.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/toast-overlay-weaponized-install-and roid-malware-single-attack-chain/
s224.1 : https://security.googleblog.com/2017/11/tizi-detecting-and-blocking-socially.html
s225.1 : https:
//blog.trendmicro.com/trendlabs-security-intelligence/new-gnatspy-mobile-malware-family-discovered/
s226.1 : https://documents.trendmicro.com/assets/tech-brief-cyberespionage-campaign-sphinx-goes-mobile-with-anubisspy.pdf
s227.1 :
https://blog.avast.com/new-version-of-mobile-malware-catelites-possibly-linked-to-cron-cyber-gang
s228.1 : https://research.checkpoint.com/malicious-flashlight-apps-google-play/
s229.1 : https:
//blog.trendmicro.com/trendlabs-security-intelligence/ghostteam-adware-can-steal-facebook-credentials/
s230.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/hacking-group-spies-android-users-india-using-poriewspy/
s231.1 : https://research.checkpoint.com/rottensys-not-secure-wi-fi-service/
s232.1 : https://researchcenter.paloaltonetworks.com/2018/03/unit42-henbox-chickens-come-home-roost/
s233.1 : https://researchcenter.paloaltonetworks.com/2018/03/unit42-telerat-another-android-trojan-leveraging-telegrams-bot-api-to-target-iranian-users/
s234.1 : https://nakedsecurity.sophos.com/2018/03/23/crooks-infiltrate-google-play-with-malware-lurking-in-qr-reading-utilities/
s235.1 : https:
//www.sophos.com/en-us/medialibrary/PDFs/technical-papers/sophos-guerilla-ad-clicker-wpna.pdf?la=en
s236.1 : https://blog.talosintelligence.com/2018/04/fake-av-investigation-unearths-kevdroid.html
s237.1 : https://securelist.com/roaming-mantis-uses-dns-hijacking-to-infect-android-smartphones/85178/
s238.1 : https://arstechnica.com/information-technology/2018/04/malicious-apps-in-google-play-gave-attackers-considerable-control-of-phones/
s239.1 : https://blog.trendmicro.com/trendlabs-security-intelligence/xloader-android-spyware-and-banking-trojan-distributed-via-dns-spoofing/
s240.1 :
https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/Sophos-Super-Clean-Plus-wpna.pdf?la=en
s241.1 : https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/05/24122414/ZooPark_for_public_final_edited.pdf
s243.1 : https://labs.bitdefender.com/wp-content/uploads/downloads/triout-the-malware-framework-for-android-that-packs-potent-spyware-capabilities/

Résumé en français

Le travail présenté dans cette thèse met l'accent sur la création d'algorithmes, de méthodes et d'outils nouveaux offrant des avantages par rapport à l'état de l'art actuel, mais plus important qui peuvent être utilisés efficacement dans un contexte de production. C'est pourquoi, ce qui est proposé ici est souvent un compromis entre ce qui peut être fait théoriquement et son applicabilité. Les choix algorithmiques et technologiques sont motivés par une relation d'efficacité et de performance résultats. Cette thèse contribue à l'état de la technique dans le domaines suivants :

- ***Analyse statique et dynamique d'applications Android***
  Tout d'abord, pour rechercher des activités malveillantes et vulnérabilités, il faut concevoir les outils qui extraient informations pertinentes provenant d'applications Android. C'est la base de toute analyse. De plus, tout classificateur ou détecteur est toujours limitée par le pouvoir informatif des données sous-jacentes. Une part importante de cette thèse est la conception d'outil d'analyse statique et dynamique d'applications, tels qu'un module de retro-ingénierie, un outil d'analyse de la communication réseau, un OS Android instrumenté, etc.
- ***Sélection des caractéristiques pour Apprentissage Artificiel***
  Le choix des caractéristiques est une étape cruciale pour n'importe quel algorithme de classification. Un ensemble de fonctionnalités optimisées réduit la durée totale et améliore la

précision du classificateur. Nous avons fait étude expérimentale approfondie pour choisir le meilleur ensemble de caractéristiques parmi un ensemble de candidats.

- ***Algorithmes d'initialisation, d'apprentissage et d'antisaturation des réseaux de neurones***
  Les réseaux de neurones sont initialisés de manière aléatoire. Il est possible de contrôler la distribution aléatoire sous-jacente afin de réduire le saturation, le temps d'entrainement et la capacité à atteindre le minimum global. Nous avons développé une procédure d'initialisation qui permet de contrôler la variance des niveaux d'activation initiaux. Finalement, nous utilisons des techniques anti-saturation et nous montrons qu'elles sont nécessaire pour former correctement un réseau de neurones.
- ***FEA : un algorithme de classification très rapide***
  Nous avons conçu un algorithme permettant une classification de 2 à 4 ordres de magnitude plus rapide que les classifieurs usuels. Nous l'utilisons afin d'évaluer rapidement la qualité discriminative d'un ensemble de caractéristiques.
- ***Un algorithme pour représenter les séquences communes dans un groupe de séquences***
  La manière classique de détecter un échantillon d'une famille de logiciels malveillants est la suivante : rétro-ingénierie et analyse manuelles des échantillons de logiciels malveillants. A partir de cette analyse, les parties malveillantes communes sont extraites pour faire une règle de détection. Cet algorithme est une nouvelle voie de recherche vers la création automatique de règles de détection de familles de programmes malveillants. Le code d'un logiciel malveillant peut être représenté comme plusieurs séquences d'opcodes, et l'analyse dynamique des logiciels malveillants génère des séquences d'événements. Trouver des similitudes dans un groupe de séquences implique souvent d'étudier leur sous-séquences commune ou leurs sous-chaînes communes. Nous proposons une représentation compacte de l'ensemble de ces sous-séquences communes, baptisée *Antichaine Englobante*. Nous présentons un algorithme capable d'en construire une approximation. Cette représentation peut être utilisée pour caractériser un groupe de séquences, et donc définir des rêgles d'appartenance.

Tout au long de cette thèse, les outils suivant ont été conçus :

- ***Glassbox : Analyse dynamique des applications de logiciels malveillants Android sur des appareils réels***
  Les chercheurs s'appuient sur une analyse dynamique pour analyser les comportements malveillants et utilisent souvent des émulateurs pour le faire. Cependant, l'utilisation d'émulateurs apporte ses propres faiblesses. Les logiciels malveillants peuvent détecter l'émulation et par conséquent n'exécute pas la charge utile pour empêcher l'analyse. Traiter avec l'évasion de périphérique virtuel est une guerre sans fin et est livré avec un coût de calcul non négligeable. Pour surmonter cet état de fait, nous proposons un système qui n'utilise pas de périphériques virtuels pour l'analyse comportement des logiciels malveillants. Glassbox est un prototype fonctionnel pour l'analyse dynamique des applications de logiciels malveillants. Il exécute des applications sur des appareils réels dans un environnement surveillé et contrôlé.
- ***Smart Monkey : test automatisé d'applications Android***
  L'analyse dynamique d'une application Android nécessite des actions de l'utilisateur pour exécuter les fonctions de l'application. Google fournis le programme *monkey* chaque téléphone Android. Il déclenche des événements utilisateur aléatoires ciblant une application. Malheureusement, des événements aléatoires rendent difficile d'accéder aux fonctions profondes du code d'application. C'est pourquoi nous avons développé un outil d'analyse *Grey Box* qui installe et teste applications. Il est capable d'identifier plusieurs champs d'intérêt (email / mot de passe, etc.) et de les remplir avec des données crédibles. De plus, il teste systématiquement toutes les fonctions visibles et ne teste pas une fonction deux fois. Nous avons expérimenté *Smart Monkey* et nous montrons qu'il est capable de couvrir plus de code source des applications que le programme *monkey*.

- ***Libmla : Bibliothèque d'algorithmes d'apprentissage automatique***
  L'apprentissage artificiel commence à être et sera une science utilisée dans tous les domaines technologiques, sans avoir recours à un expert. Les avancées de l'apprentissage en profondeur évitent l'exigence d'un processus d'extraction / de sélection de caractéristiques. Cependant, l'algorithme d'apprentissage et d'autres paramètres internes doivent encore être modifiés. L'apprentissage artificiel doit encore être conçu et configuré par une personne ayant de l'expérience pour être efficace. Enfin, bien que l'apprentissage en profondeur soit très efficace, il nécessite une forte quantité de puissance CPU / GPU. *Libmla* est une autre direction de contribution à l'apprentissage artifiel. Il est capable de construire des algorithmes d'IA avec de bonnes performances, mais avec beaucoup moins de ressources. C'est pourquoi *Libmla* est un open source qui promeut l'utilisation d'algorithmes d'apprentissage automatique optimisés, efficaces et faciles à utiliser. À l'heure actuelle, libmla est fortement centré sur l'apprentissage automatique adapté au domaine de la détection de logiciel malveillants, mais nous avons l'intention d'étendre son cadre d'origine avec des cas d'utilisation réels dans d'autres domaines.

Nous avons présenté nos travaux à travers des publications, des conférences et des articles de vulgarisation scientifique : :

- **Articles internationaux**
  - Paul Irolla and Alexandre Dey, *The duplication issue within the Drebin dataset*, Journal of Computer Virology and Hacking Techniques, DOI 10.1007/s11416-018-0316-z.
- **Workshops et conferences internationaux avec article**
  - Paul Irolla and Éric Filiol, *(In)Security of Mobile Banking... And of Other Mobile Apps*, Black Hat Asia 2015 [18].
  - Paul Irolla, *Glassbox : Dynamic Analysis Platform for Malware Android Applications on Real Devices*, ICISSP ForSE 2017, DOI 10.5220/0006094006100621
  - Paul Irolla. *Systematic Characterization of a Sequence Group*. ForSE 2019.
  - Abhilash Hota and Paul Irolla. *Deep Neural Networks for Android Malware Detection*. ForSE 2019.
- **Conferences internationales sans articles**
  - Paul Irolla and Éric Filiol, *(In)Security of Mobile Banking*, Chaos Communication Congress 2014 [19].
  - Paul Irolla, *(In)Security of Mobile Banking... And of Other Mobile Apps*, C0c0n 2015 [20].
- **Publications et articles de vulgarisation nationaux**
  - Paul Irolla, *Applications bancaires pour smartphone : une sécurité bâclée*, SecuriteOff 2015 [21].
  - Paul Irolla, *La surveillance mondiale du Wifi*, SecuriteOff 2015 [22].
  - Paul Irolla, *User tracking et Facebook : l'ère de la surveillance globale*, SecuriteOff 2016 [23].
  - Éric Filiol, Baptiste David and Paul Irolla, *Virus informatiques et autres infections informatiques*, Techniques de l'Ingénieur 2017, Ref H5440 V3.

18. https://www.blackhat.com/docs/asia-15/materials/asia-15-Filiol-InSecurity-Of-Mobile-Banking-wp.pdf
19. https://media.ccc.de/v/31c3_-_6530_-_en_-_saal_6_-_201412272145_-_in_security_of_mobile_banking_-_ericfiliol_-_paul_irolla
20. http://is-ra.org/c0c0n/2015/speakers.php#Irolla_Paul
21. https://www.securiteoff.com/applications-bancaires-une-securite-baclee-12/
22. https://www.securiteoff.com/dangers-dune-cartographie-mondiale-points-dacces-wi-fi/
23. https://www.securiteoff.com/user-tracking-facebook-lere-de-surveillance-globale/