Titre: Title:	Un modèle de détection de logiciels malveillants pour terminaux mobiles
Auteur: Author:	Corentin Rodrigo
Date:	2020
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Rodrigo, C. (2020). Un modèle de détection de logiciels malveillants pour terminaux mobiles [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/5339/

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Un modèle de détection de logiciels malveillants pour terminaux mobiles

CORENTIN RODRIGO

Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

Génie informatique

Août 2020

POLYTECHNIQUE MONTRÉAL

Affiliée à l'Université de Montréal

Ce mémoire intitulé :

Un modèle de détection de logiciels malveillants pour terminaux mobiles

présenté par **Corentin RODRIGO**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Martine BELLAÏCHE, présidente
Samuel PIERRE, membre et directeur de recherche
Ronald BEAUBRUN, membre et codirecteur de recherche
Alejandro QUINTERO, membre externe

DÉDICACE

« Tous les progrès sont précaires, et la solution d'un problème nous confronte à un autre problème »

King, Martin Luther, and Jean Bruls. La force d'aimer. Casterman, 1964.

« La seule chose que l'on puisse décider est quoi faire du temps qui nous est imparti »

Le Seigneur des Anneaux (1954), John Ronald Reuel Tolkien (trad. Francis Ledoux), éd. Christian Bourgois, 1972, p. 68

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui ont contribué au succès de ma maîtrise recherche et qui m'ont aidé lors de la rédaction de ce mémoire.

Je voudrais dans un premier temps remercier, mon directeur de mémoire Monsieur le Professeur Samuel PIERRE, professeur à l'École Polytechnique de Montréal et directeur du Laboratoire de Réseautique et Informatique Mobile (LARIM), pour m'avoir offert l'opportunité de faire ma maîtrise dans son laboratoire de recherche.

Dans un second temps, j'aimerais remercier mon co-directeur de mémoire, Monsieur le Professeur Ronald BEAUBRUN pour ses conseils qui m'ont permis d'alimenter ma réflexion ainsi que pour la relecture complète du présent mémoire.

Je remercie également tout le personnel de Flex Groups, pour m'avoir offert un cadre permettant d'allier le monde de la recherche et le monde professionnel ainsi qu'un financement.

Je remercie également les membres du LARIM pour l'aide fournie.

Je tiens à témoigner toute ma reconnaissance pour leur aide dans la réalisation de ce projet de maîtrise à:

- Madame le Docteur Franjieh EL KHOURY, coordonnatrice du LARIM et superviseure du projet ATISCOM, qui m'a beaucoup encadré tout au long du mémoire. Elle a partagé ses connaissances et expériences dans ce milieu. Elle a aussi participé à la relecture de mon mémoire.
- Mon père pour avoir relu et corrigé mon mémoire. Ses conseils en rédaction ont été très précieux.
- Ma famille, mes amis ainsi que ma petite amie pour leur soutien constant.

RÉSUMÉ

Un modèle de détection de logiciels malveillants pour terminaux mobiles contribue au domaine de la sécurité informatique. La cybersécurité est une problématique actuelle majeure principalement motivée par le nombre croissant de cyberattaques. En effet, les pertes de données à cause des brèches informatiques ont coûté 45 milliards de dollars canadiens en 2018. En plus, du point de vue financier, des problèmes éthiques apparaissent également si des informations personnelles de clients et utilisateurs sont divulguées. En raison de la popularité des téléphones intelligents et des tablettes, les terminaux mobiles deviennent la cible de cyberattaques. Il est donc essentiel d'étudier de nouveaux moyens de prévenir, de détecter et de contrer les cyberattaques. Dans ces mécanismes de détection, l'apprentissage machine est utilisé pour créer des classificateurs qui permettent de déterminer si une application est dangereuse ou non. L'avantage d'un réseau de neurones est qu'il permet de s'adapter à des situations inédites. Contrairement à un système de règles de sécurité fixes, nous allons utiliser cette nouvelle technologie afin de pouvoir identifier des types de comportements malveillants et de pouvoir le généraliser à des programmes malveillants futurs.

L'objectif de cette recherche consiste à proposer un modèle de détection hybride de programmes malveillants sur Android basé sur deux réseaux de neurones de classification entraînés par des ensembles de caractéristiques statiques et dynamiques. Nous avons tout d'abord procédé à une revue de littérature afin de connaître les techniques de détection existantes. Cette étude n'est pas exhaustive, mais permet de cerner les principaux enjeux rencontrés ainsi que les solutions proposées par la communauté scientifique. Ces dernières peuvent se répartir en deux groupes; les méthodes statiques consistent à examiner le code de l'application mobile, tandis que les méthodes dynamiques analysent le comportement d'une application lorsque cette dernière est exécutée roule sur un terminal mobile. Notre but est d'utiliser ces deux méthodes afin de profiter de leurs avantages respectifs. Pour ce faire, nous avons choisi d'utiliser la base de données hybride Omnidroid composée de 25,999 caractéristiques statiques et de 5,932 caractéristiques dynamiques. Nous montrons lors de nos travaux que 22,636 caractéristiques statiques ainsi que 2,210 caractéristiques dynamiques de la base de données d'Omnidroid sont vides. Nous menons également un plan d'expérience composé de centaines d'entraînements afin de régler les valeurs

des hyperparamètres améliorant l'apprentissage sur ce jeu de données ainsi que pour sélectionner les caractéristiques restantes les plus pertinentes.

Nous présentons finalement l'application mobile **BrainShield** développée en guise de prototype d'implémentation des meilleurs modèles entraînés. Les résultats finaux sont les suivants : le premier réseau de neurones statique atteint une justesse de 92.9% avec une précision de 91.1% entraîné sur 840 caractéristiques statiques. Le deuxième réseau de neurones dynamique atteint une justesse de 81.1% avec une précision de 83.4% entrainé sur 3,722 caractéristiques dynamiques. Le troisième réseau de neurones proposée est hybride atteignant une justesse de 91.1% avec une précision de 91.0% entrainée sur 7081 caractéristiques statiques et dynamiques.

ABSTRACT

A malware detection model for mobile devices contributes to the field of computer security. Cybersecurity is a major current problem mainly motivated by the growing number of cyber attacks. Indeed, data loss due to computer breaches cost Canada $45 billion in 2018. In addition, ethical problems also arise if personal information of customers and users is disclosed. Due to the popularity of smartphones and tablets, mobile devices are becoming the target of cyberattacks. It is therefore essential to explore new ways to prevent, detect and counter cyberattacks. In these detection mechanisms, machine learning is used to create classifiers that determine whether an application is dangerous or not. The advantage of a neural network is that it allows you to adapt to new situations. Unlike a system of fixed security rules, we will use this new technology in order to be able to identify types of malicious behavior and to be able to generalize it to future malicious programs.

The goal of this research is to propose a hybrid malware detection model on Android itself based on two classification neural networks driven by sets of static and dynamic features. We first conducted a literature review to find out about existing detection techniques. These can be divided into two groups; static methods consist of examining the code of the mobile application while dynamic methods analyze the behavior of an application when it is running on a mobile terminal. Our goal is to use these two methods to take advantage of their respective advantages. To do this, we chose to use the hybrid database “Omnidroid” composed of 25,999 static features and 5,932 dynamic features. We show that 22,636 static features as well as 2,210 dynamic features of the Omnidroid database are empty. We are also carrying out an experiment plan composed of hundreds of trainings in order to adjust the values of the hyperparameters improving the learning on this dataset as well as to select the most relevant remaining features.

We finally present the mobile application “BrainShield” developed as a prototype implementation of the best trained models. The first static neural network reaches an accuracy of 92.9% with an accuracy of 91.1% trained on 840 static features. The second dynamic neural network reaches an accuracy of 81.1% with an accuracy of 83.4% trained on 3,722 dynamic features. The third neural network proposed is hybrid, reaching an accuracy of 91.1% with an accuracy of 91.0% trained on 7081 static and dynamic features.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AOSP	Android Open Source Project
API	Application Programming Interface
APK	Android Package Kit
ART	Android Runtime
ATISCOM	Architecture et Technologies Innovantes de Soutien au COmmerce Mobile
AUC	Area Under the Curve
BYOD	Bring Your Own Device
CBG	Component Behaviour Graphs
CSV	Comma Separated Values
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CUDNN	Compute Unified Deep Neural Network
DEX	Dalvik Executable
ET	Extra Tree
FP	False Positive ; Faux Positif
FN	False Negative ; Faux Négatif
GPU	Graphical Processing Unit
HTTP	Hypertext Transfer Protocol
ICC	Inter-Component Communication
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-Process Communication
JDK	Java Development Kit
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LARIM	LAboratoire de recherche en Réseautique et Informatique Mobile
LSTM	Long Short-Term Memory
MDM	Mobile Device Management
OS	Operating System

PHA	Potentially Harmful Application
QEMU	Quick Emulator
RAM	Random Access Memory
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
RPC	Remote Procedure Call
SVM	Support Vector Machine
TFP	Taux de Faux Positif
TP	True Positive
TN	True Negative
UI	User Interface
VPN	Virtual Private Network
WEKA	Waikato Environment for Knowledge Analysis
XML	Extensible Markup Language

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

En raison de leur popularité, les terminaux mobiles font de plus en plus partie de notre quotidien avec, entre autres, les téléphones intelligents [1], les tablettes tactiles [2], les montres connectées [3], les TV connectés [4] et les voitures connectées [5]. Cependant, ces terminaux qui manipulent à la fois des données privées et confidentielles, tant dans le cadre professionnel que personnel, sont vulnérables. En effet, des personnes mal intentionnées utilisent des **logiciels malveillants** afin de récolter ces informations. On parle alors de cyberattaques. Parmi les exemples récents de cyberattaque les plus connus, nous pouvons citer l'attaque par déni de service distribué (DDoS) **Mirai Botnet** [6] et le détournement massif de données mené par le ransomware **WannaCry** [7]. Cette situation rend les techniques de détection de logiciels malveillants dignes d'être étudiées et améliorées. C'est pourquoi il convient de mettre en place **un modèle de détection de logiciels malveillants pour terminaux mobiles.**

Ce premier chapitre présente, dans un premier temps, les définitions et les concepts de base pour s'assurer une bonne compréhension de la problématique et des objectifs de recherche. Par la suite, les éléments de la problématique sont exposés. Nous définirons alors les objectifs de recherche. Enfin, nous terminerons le chapitre par la présentation du plan du mémoire.

1.1 Définitions et concepts de base

Commençons par définir les termes reliés aux applications mobiles. Notre recherche repose sur la détection de logiciels malveillants [8]. Leur utilisation sert des buts illégaux, comme l'espionnage ou l'extorsion. Concernant les logiciels dont le comportement pourrait ressembler à celui d'un logiciel malveillant sans avoir été conçu comme tel, comme par exemple la collecte abusive de données ou un envoi trop important de messages, nous parlerons de logiciel malveillant gris. C'est souvent le cas des logiciels de publicités qui envoient énormément de publicités. Dans notre cas, lorsque nous parlerons de logiciels malveillants, il s'agira généralement d'applications mobiles, et par conséquent de paquets permettant de les installer.

Nous considérerons aussi des applications **bénignes**. Il s'agit d'applications qui sont considérées comme légitimes et inoffensives. Notons que la certitude de la classification d'une application comme étant bénigne ou malveillante est sujette aux méthodes utilisées pour établir cette classification. Bien souvent, nous considèrerons toute application provenant d'une source sûre ou

vérifiée par des moyens tiers comme bénigne, mais une incertitude existe toujours. Au contraire, une application qui présente des fonctionnalités ou des comportements malveillants et classée comme application malveillante par des experts ne laisse généralement aucun doute. Ce manque de certitude absolue constitue l'une des possibles failles d'un classificateur.

Afin de caractériser le comportement d'une application donnée, il est possible d'utiliser la technique **d'analyse statique [**9**]**. Cette technique consiste à inspecter le package, décompiler le code ou accéder aux différents fichiers contenus dans le package. Cela permet de rassembler un ensemble de fonctionnalités pertinentes et utiles, telles qu'une liste d'appels d'API (Application Programming Interface) invoqués dans le code ou l'ensemble d'autorisations requises pour déployer l'ensemble des fonctionnalités. L'un des principaux avantages d'une approche statique réside dans sa facilité d'utilisation. Cette méthode, plus légère en temps de calcul et dans l'extraction des caractéristiques, est plus facile à mettre en œuvre qu'une méthode dynamique.

En revanche, **l'analyse dynamique [**9**]** choisit de capturer les actions réellement déclenchées par l'application analysée lors de son émulation ou bien lors de son roulement sur un terminal physique pour exécuter l'application, tandis qu'un agent de surveillance capture une série d'indicateurs, tels que les composants matériels utilisés, le trafic réseau ou les appels systèmes invoqués. L'un des principaux avantages d'une approche dynamique réside dans sa capacité à capturer les événements invoqués dans des sections de code masquées ou incluses dans du code chargé dynamiquement.

L'analyse de détection de logiciels malveillants **hybride** consiste à utiliser à la fois des caractéristiques extraites d'une analyse statique et des caractéristiques extraites d'une analyse dynamique. C'est la méthode théoriquement la plus prometteuse. En effet, ces deux types de méthodes sont complémentaires. L'analyse statique est vulnérable à certains vecteurs d'attaques, comme l'obfuscation de code, le chargement dynamique de code ou bien encore le chiffrement. Ce sont ces techniques qui se trouvent être les points forts d'une analyse dynamique. La combinaison de ces méthodes permet de pallier les lacunes de l'une et de l'autre et de pouvoir couvrir une grande variété de vecteurs d'attaques. De plus, l'analyse dynamique nécessite de faire exécuter l'application qui nécessite plus de capacité de calcul que de faire une analyse sur le code source de l'application, comme peut le faire l'analyse statique. En fait, ces méthodes sont complémentaires et doivent être utilisées ensemble.

L'apprentissage machine [10] (machine learning) est une discipline qui est constituée de nombreuses méthodes et objectifs différents. Le point commun à ces méthodes est de leur fournir un très grand nombre de caractéristiques, étiquetées ou non, en entrée de l'algorithme d'apprentissage. Dans le cas où les données sont étiquetées, nous parlerons d'apprentissage supervisé, et d'apprentissage non supervisé dans le cas contraire. Cet algorithme a pour but de donner un sens aux données afin de définir des règles pour traiter de futures données qui leur ressemblent. L'apprentissage machine permet de se passer d'une analyse exacte, impossible à mener à cause de sa complexité et intervient dès lors que le nombre de variables et de données est trop important pour être traité par un humain ou par un algorithme dont les calculs prennent trop de temps. Dans tous les cas, la quantité et l'équilibre des données sont très importantes pour bâtir un bon modèle de classification. De plus, différents algorithmes de régression sont plus adaptés à différents types de données et de tâches. Dans notre cas, il s'agira de prendre certaines caractéristiques connues de nombreuses applications identifiées comme étant des logiciels malveillants ou non par des dizaines d'antivirus. Le but est de déterminer si de nouveaux échantillons fournis à notre modèle de classification sont des logiciels malveillants ou non à partir des règles précédemment établies par l'algorithme d'apprentissage.

Les **caractéristiques** (features) **[10]** sont choisies par l'humain dans le cas de l'apprentissage supervisé, ou par l'algorithme dans le cas de l'apprentissage non supervisé. Elles permettent de représenter une application de la manière la plus fidèle possible. Les **caractéristiques statiques** sont celles obtenues à l'aide d'outil statique tandis que les **caractéristiques dynamiques** sont celles obtenues via des outils dynamiques.

Les outils de détection de programmes malveillants basés sur l'apprentissage machine nécessitent des **jeux de données [10]** (ou dataset) d'exemplaires étiquetés comme logiciels malveillants ou inoffensifs. **L'étiquetage** est le fait de considérer une application comme application malveillante ou comme application bénigne. L'utilisation de ces ensembles de données est fondamentale pour créer des méthodes fiables de détection et de classification des logiciels malveillants.

Un problème de **classification [11]** consiste à déterminer la classe d'une application. Les différentes classes pour notre sujet de recherche sont de deux types; application malveillante et application bénigne. Il s'agit donc d'un problème de **classification binaire**. En effet, une application peut être bénigne ou bien malveillante.

Les métriques d'évaluation **[**11**]** sont des mesures quantifiables nous permettant de savoir si le modèle de détection permet effectivement de différencier les applications malveillantes des applications bénignes. Parmi ces métriques, citons la **justesse** (accuracy) **[**7**]** qui est la proportion de prédictions correctes. La **précision** (precision) **[**7**]** est la proportion de prédictions positives correctes. Un modèle de détection ne produisant aucun faux positif a une précision de 1. Le **rappel** (recall) **[**7**]** est la proportion de résultats positifs réels qui a été identifiée correctement. Le rappel est aussi appelé le taux de vrai positif (TPR). Un modèle de détection ne produisant aucun faux négatif a un rappel de 1. Le **score F1** [12] est la moyenne harmonique de la précision et du rappel. Par conséquent, ce score prend en compte à la fois les faux positifs et les faux négatifs.

Android [13] est un système d'exploitation en libre accès pour les terminaux mobiles dirigé par Google Inc. Nous pouvons trouver le code source de ce projet sous la dénomination AOSP (Android Open Source Project) [14]. Vous trouverez l'architecture Android en Annexe A.

Le paquet Android (ou APK, pour Android Package Kit) [15] est un format de fichiers pour le système d'exploitation Android. Un APK (ex. : « NomApplication.apk ») est une collection de fichiers compressés pour Android. Chaque application est en fait un APK.

1.2 Éléments de la problématique

Dans le contexte du commerce mobile, les problématiques de sécurité des terminaux mobiles sont importantes car ces terminaux implémentent des applications de paiement mobile, comme Apple Pay et Google Pay.

Les logiciels malveillants, sous leurs différentes formes, représentent un problème majeur affectant différentes plates-formes, des ordinateurs personnels aux smartphones, en passant par l'Internet des objets (IoT). En effet, les logiciels permettent, entre autres, de prendre le contrôle du terminal mobile ou bien de voler des informations. Depuis l'apparition du premier virus conçu pour les ordinateurs dans les années 70, les logiciels malveillants ont mis au point différentes parades pour faire face aux contre-mesures imposées par les systèmes d'exploitation et les anti-virus [16]. Cela a conduit à une course où les experts en sécurité poursuivent toujours les Black Hat, les pirates informatiques mal intentionnés.

Pour résoudre ce problème, les outils de détection de logiciels malveillants essaient d'extraire et de modéliser le comportement d'un échantillon suspect, qui est comparé à des modèles bénins ou malveillants, afin de prendre une décision concernant sa nature. De plus, les méthodes de détection actuelles nécessitent des mises à jour continues afin de pouvoir détecter les logiciels malveillants venant d'être découverts. Les nouvelles méthodes de détection que beaucoup de chercheurs proposent se basent sur les **techniques d'apprentissage machine**. En effet, les réseaux de neurones peuvent aider à satisfaire cette demande en construisant des classificateurs permettant de déterminer si une application est bégnine ou malveillante. L'avantage de l'utilisation de cette technique est qu'elle est capable de s'adapter à des situations inédites contrairement aux systèmes de détections traditionnels fonctionnant avec des règles de sécurité. En outre, le nombre de propositions des méthodes de détection sur Android ne cesse d'augmenter dans la littérature tandis que les solutions sur iOS sont impossibles à cause de la politique d'Apple restrictives.

Celui-ci n'a pas pleinement confiance en cette méthode de paiement. En effet, le **risque zéro n'existe pas** et plus particulièrement dans les nouvelles technologies.

Chaque terminal mobile roule sur un système d'exploitation donné. Concernant le marché mobile par exemple, deux systèmes d'exploitation se partagent aux alentours de 98% du marché Android (architecture Android disponible en Annexe A) représente plus de 80% du marché mobile tandis qu'iOS d'Apple Inc. représente un peu moins de 20% du marché selon le Worldwide Quarterly Mobile Phone Tracker [17] et selon le Statcounter [18]. Un aperçu de la **sécurité Android** a été présenté par Kaspersky **[**19**]** qui est discutable, principalement parce que personne ne la possède et donc personne ne réglemente ce qui peut ou ne peut pas être offert comme application Android, ni même ce qui peut être vendu comme un téléphone Android. Mais, comme le souligne Forbes, vous pouvez sécuriser votre téléphone Android en le mettant à jour et en évitant de télécharger des applications d'origine inconnue ou douteuse. **Ce système d'exploitation est vulnérable**. De plus, étant le plus utilisé il est aussi une cible privilégiée. Le système d'exploitation mobile iOS d'Apple est étroitement contrôlé par Apple, qui contrôle également les applications disponibles sur l'App Store d'Apple, la seule et unique source d'applications iOS [20]. Ce contrôle permet aux terminaux Apple d'offrir une bonne sécurité prête à l'emploi, au prix de certaines restrictions imposées aux utilisateurs. Par exemple, iOS n'autorise qu'une copie d'une application sur chaque appareil.

Dans ce contexte et sans vouloir faire de généralité, nous chercherons à proposer un modèle de détection de logiciels malveillants pour terminaux mobiles. En effet, ce système d'exploitation est ouvert, contrairement à iOS, ce qui offre plus d'opportunités ainsi qu'une large communauté. De plus, le système d'exploitation Android peut être utilisé pour différents types de terminaux mobiles. Notre question de recherche sera donc :

Pouvons-nous mettre en place des stratégies permettant de détecter plus efficacement les logiciels malveillants sur les terminaux qui utilisent le système d'exploitation Android ?

1.3 Objectifs de recherche

L'objectif de notre recherche est de proposer un modèle de détection hybride de logiciels malveillants dédié aux terminaux Android.

De manière spécifique, ce mémoire vise à :

- Analyser les méthodes existantes de détection de logiciels malveillants sur Android, en mettant l'accent sur les différentes approches de détection;
- Proposer un modèle de détection hybride de logiciels malveillants, basé sur des réseaux de neurones de classifications qui sont entraînés par des ensembles de caractéristiques statiques et dynamiques;
- Implémenter et évaluer le modèle proposé avec les métriques d'évaluation, telles que la justesse et la précision.

1.4 Plan du mémoire

La suite du mémoire est organisée de la manière suivante. Le chapitre 2 est une **revue de littérature** du domaine. La section 2.1 traite des méthodes d'analyses du système d'exploitation Android. La section 2.3 est une présentation des produits d'entreprises que nous pouvons trouver sur le Google Play Store et une présentation de la solution proposée par Google Inc. La section 2.3 s'intéresse aux articles de recherche scientifique publiés dans les journaux scientifiques. Puis la section 2.4 expose les lacunes des méthodes présentées. L'analyse de l'ensemble de ces méthodes et des articles publiés récemment permettra de concevoir notre propre méthode

Le chapitre 3 présentent les **méthodes de détection proposées**. La section 3.1 présente la base de données sur laquelle s'entraîne les méthodes de détection proposées. La section 3.2 expose les techniques d'intelligence artificielle utilisés. Il s'agit plus précisément d'un réseau de neurones où tous les neurones sont connectés entre eux. Nous présenterons par la suite des techniques d'apprentissage populaires comme la régularisation. La section 3.3 expose l'architecture client/serveur du modèle proposé.

Le chapitre 4 fait apparaître l'ensemble des méthodes et des **résultats** des réseaux de neurones pour la détection de logiciels malveillants. Nous retrouverons, dans la section 4.1, les conditions d'environnements et le matériel utilisé qui nous ont permis d'avoir les résultats à des fins de transparence et de reproductibilité. On fera connaître en section 4.2 le **prototype** d'implémentation BrainShield intégrant les méthodes de détection. La section 4.3 contient le code d'implémentation des réseaux de neurones tandis que la section 4.4 vise à montrer la méthode des expérimentations menées pour régler les hyperparamètres. La section 4.5 et 4.6 montrent les méthodes pour choisir les caractéristiques pertinentes. La section 4.7 met en avant l'impact des sections précédentes au cours de l'apprentissage du réseau de neurones statique. La section 4.8 introduit les méthodes et les résultats des ré-étiquetages de la base de données. Enfin, les résultats finaux en termes de métriques d'évaluation des réseaux de neurones statique, dynamique et hybride sont soumis en section 4.9.

Le chapitre 5 est une **conclusion** comprenant une synthèse de l'ensemble des travaux tout en énonçant les limitations des travaux entrepris ainsi que les travaux futurs envisagés.

CHAPITRE 2 DÉTECTION DE LOGICIELS MALVEILLANTS SUR LES TERMINAUX MOBILES ANDROID

Ce chapitre présente une revue de littérature sur la détection de logiciels malveillants ciblant les terminaux mobiles qui utilisent le système d'exploitation Android. La première section de la revue de littérature présente les différentes méthodes d'analyses du système d'exploitation Android. La deuxième partie aborde les solutions proposées par les entreprises disponibles sur le Google Play Store ainsi que la solution proposée par Google Inc. La troisième section se concentre sur les différentes approches des travaux de recherche. Enfin, nous trouverons dans la quatrième et dernière section les lacunes des solutions précédemment présentées.

2.1 Méthodes d'analyses du système d'exploitation Android

Cette section présente les différentes méthodes d'analyses du système d'exploitation Android.

2.1.1 Analyse par méta-informations

Beaucoup de méthodes basent leurs mécanismes de détection de logiciels uniquement sur les **méta-informations** fournies par les développeurs. Tel est le cas d'ADROIT [21], un système qui forme des algorithmes d'apprentissage de machines avec un ensemble de fonctionnalités extraites du fichier AndroidManifest.xml et exécute un processus d'extraction de texte sur le texte de description de l'application Android. Manilyzer [22] est un autre exemple, axé sur l'utilisation des informations pouvant être recueillies à partir du fichier AndroidManifest.xml d'Android pour former des algorithmes d'apprentissage automatique.

Les **permissions systèmes** sont des méta-informations qui sont largement reprises dans la littérature. En effet, il s'agit de déclarations dans le fichier AndroidManifest.xml d'une application Android importantes puisqu'elles indiquent ce que l'application peut faire ou non. Elles sont aisément lisibles. Les permissions sont l'une des caractéristiques les plus pertinentes pour réaliser une détection de logiciels malveillants Android. Par exemple les attaques induites par les permissions systèmes Android sont des violations de la sécurité et font partie des problèmes les plus critiques. A. Sadeghi et al. [23] proposent TERMINATOR qui est un cadre d'analyse des autorisations temporaires et d'application pour Android. En exploitant la vérification du modèle logique temporel, l'analyseur identifie les menaces induites par l'autorisation dynamique des états

des applications. Au moment de l'exécution, une autorisation temporaire des applications est réalisée lorsque le système est dans un état sûr et révoque les autorisations lorsque le système se déplace à un état non sécurisé réalisant les menaces identifiées. Enfin, TERMINATOR ne nécessite pas de modification du Framework Android ou de logique de mise en œuvre des applications.

2.1.2 Analyse du code natif

Le code natif est un code machine directement exécuté par un processeur ; par opposition au bytecode qui est exécuté dans une machine virtuelle telle que la JVM (machine virtuelle Java) ou anciennement Dalvik-VM. Toutes les applications fonctionnent au niveau du bytecode Java. Cependant, de nombreuses applications Android contiennent des fonctionnalités implémentées en code natif et non en Java. Alors que le modèle de programmation standard pour les applications Android consiste à écrire l'application en Java et à distribuer le bytecode, les développeurs peuvent également inclure des fichiers binaires de code natif dans leurs applications. Cela est souvent fait pour utiliser une bibliothèque existante ou pour écrire du code dépendant du matériel, tel qu'un moteur graphique de jeu, un lecteur vidéo, une application de traitement de signal, une simulation physique, etc...

Les études menées par V. Afonso et al. Et par M. Sun et G. Tan estiment [24] que 37% de toutes les applications Android et 86% des applications les plus populaires contiennent du code natif [25]. Étant donné que les systèmes existants de détection de logiciels malveillants basés sur l'analyse statique fonctionnent au niveau du bytecode, cela signifie que ces applications ne peuvent pas être analysées complètement. Dans le cas où le contenu malveillant de l'application est inclus dans le code natif, les systèmes existants ne peuvent pas le détecter. De plus, le code natif a plus de fonctionnalités que le bytecode. En effet, le code natif a un accès direct à la mémoire du processus en cours et peut donc lire et modifier le comportement du bytecode.

Par exemple, DroidNative [26], construit des signatures sémantiques multi-plateformes (x86 et ARM) pour Android et fonctionne au niveau du code natif. En effet, les applications peuvent télécharger certaines familles de codes malveillants en code natif au moment de l'exécution. En effectuant une analyse statique sur ces produits malveillants à code natif chargé dynamiquement, DroidNative peut détecter ces actions et ainsi peut être utilisé dans le cadre d'un système hybride.

2.1.3 Analyse de clones

Les clones d'applications Android peuvent voler le revenu des développeurs d'applications en les copiant puis en les redistribuant gratuitement tout en y insérant du code malveillant. Ceux-ci étant souvent cachés par leurs créateurs mal intentionnés, une identification triviale de ces applications reconditionnées n'est pas possible. Le problème est intensifié par l'usage répandu des bibliothèques. Dans de nombreuses applications, la taille du code total est essentiellement déterminée par la taille des bibliothèques. Par conséquent, deux applications dont l'une est la reconditionnée de l'autre ont des codes très similaires. Pour détecter de manière fiable les applications reconditionnées CodeMatch [27] propose une approche en deux étapes qui se concentre d'abord sur l'identification et la suppression des données, puis sur les codes de la bibliothèque des clones. Cette approche s'appelle LibDetect et s'appuie sur des représentations de code qui résument en plusieurs parties, du code sous-jacent pour résister à certaines techniques d'obscurcissement. Après la suppression du code de la bibliothèque d'une application, CodeMatch effectue ensuite un hachage fuzzy (de manière aléatoire) qui est une représentation la plus abstraite du code de l'application restante pour s'assurer d'identifier les applications reconditionnées même si des techniques d'obscurcissement très avancées sont utilisées. En utilisant cette technique, ils ont trouvé que 15% de toutes les applications dans les magasins d'applications Android sont des reconditionnements.

DroidCC [28] est une autre approche de détection de clone dans les applications Android, qui permet de détecter différents types de clones à partir du code source de l'APK. Un outil de détection en libre accès[1] a été développé et mis en œuvre sur le jeu de données de près de 30 000 applications Android les mieux notées. DroidCC détecte au niveau du code ainsi que des fragments de code similaire, qui ont été injectés dans de nombreuses applications ce qui peut être une indication de propagation de logiciels malveillants.

[1] Qingyu Mao (2019), DroidCC, https://github.com/maoqyhz/DroidCC, (Consulté en Juin 2019)

2.1.4 Analyse du trafic réseau

Le trafic réseau généré par les applications Android doit être compris à des fins de gestion du réseau, d'analyse du trafic d'applications et de détection de logiciels malveillants. Cependant, il est fastidieux d'installer et d'exécuter manuellement des applications Android pour générer du trafic réseau. AndroGenerator [29] est un système de génération de trafic réseau automatique sur Android. Ce dernier reproduit le trafic réseau en fonction des caractéristiques de trafic d'applications Android. Une fois le réseau généré, il peut être analysé. C'est ce qu'a fait Q. Wang et al. [30]. Cet article montre que même en présence de chiffrement, un attaquant sans accès aux clés de cryptage peut déterminer le comportement des utilisateurs, c'est-à-dire l'utilisation de l'application. Cette attaque est effectuée à l'aide d'une analyse de trafic au niveau paquet et montre qu'en en collectant et en analysant de petites quantités de trafic sans fil, il est possible de savoir quelle application est utilisée. De ce fait, les applications génèrent plus de modèles de trafics identifiables. En utilisant des forêts aléatoires pour classer les applications, ils sont capables d'identifier les applications utilisées même en présence de bruit. Étant donné que la plupart des services en ligne fournissent maintenant des applications natives pouvant être identifiées par cette méthode, ces attaques constituent une menace sérieuse pour la vie privée des utilisateurs et servent comme porte d'entrée pour une attaque plus sérieuse.

2.1.5 Analyse des publicités et des librairies

Les bibliothèques tierces sont intégrées de manière omniprésente dans les applications Android. Les bibliothèques tierces offrent des fonctionnalités telles que les annonces, les services de localisation et les services de réseaux sociaux pour rendre le développement d'applications multifonctionnelles beaucoup plus productif. Les bibliothèques peuvent également nuire à l'ensemble de l'écosystème mobile entraînant divers problèmes de sécurité. Par conséquent, l'identification des bibliothèques est devenue un problème important. LibD [31] propose une nouvelle approche pour identifier les bibliothèques Android tierces. Basée sur le hachage des fonctionnalités, la méthode permet de mieux gérer le code dont les noms de paquets et de méthodes sont obscurcis. Elle utilise le code interne comme les dépendances d'une application Android pour détecter et classer la bibliothèque analysée.

2.1.6 Analyse des ICC

Un autre vecteur de défense consiste à analyser les informations transmises via le service d'ICC (Inter-Component Communication) qui peut être utilisé par un logiciel malveillant pour effectuer des actions malveillantes telles que l'installation d'une nouvelle application.

Les composants d'application Android peuvent communiquer les uns avec les autres, à la fois au sein d'applications uniques et aussi entre différentes applications. Malheureusement, les techniques permettant d'analyser statiquement les ICC génèrent de nombreux liens inter-composants et inter-applications potentiels, dont la plupart sont des faux positifs. À grande échelle, examiner tous les liens potentiels n'est tout simplement pas faisable. D. Octeau et al [32] superposent un modèle probabiliste d'ICC sur les résultats des analyses statiques. Les auteurs ont conçu un algorithme pour effectuer la résolution de liens avant de calculer tous les liens potentiels dans un corpus de 11 267 applications en 30 minutes et de les trier selon leur approche probabiliste. Ils ont constaté que plus de 95,1% des 636 millions de liens potentiels sont associés à des valeurs de probabilité inférieures à 0,01 et sont donc probablement des liens non réalisables.

2.1.7 Analyse par vulnérabilité

Les « patchs à chaud » sont des patchs destinés à être déployés plus rapidement que les correctifs permanents qui eux sont utilisés pour bloquer les exploits des vulnérabilités nouvellement découvertes. InstaGuard [33] révèle un obstacle majeur qui empêche les méthodes de patch à chaud existantes d'être efficaces sur les terminaux mobiles : après leur développement, les correctifs à chaud pour les terminaux mobiles doivent passer par de longs tests de compatibilité. Les partenaires Android imposent toutes les mises à jour du code système. Le processus de test et de publication peut prendre des mois et, par conséquent, effacer l'avantage principal des correctifs à chaud d'être rapide. InstaGuard est une nouvelle approche de patchs à chaud pour dispositifs mobiles permettant un déploiement instantané des correctifs et développement rapide de correctifs pour vendeurs. Les auteurs ont également construit RuleMaker, un outil permettant de générer automatiquement des règles pour InstaGuard basé sur une vulnérabilité de haut niveau facile pour écrire des descriptions. InstaGuard peut gérer tous les CVE critiques d'Android Bulletins de sécurité rapportés en 2016 [34].

2.1.8 Analyse par appels API

Les **appels d'API** font partie des fonctionnalités les plus utilisées avec les permissions Android. Qu'elles soient statiques ou dynamiques, elles permettent de modéliser le comportement d'une application et de caractériser les actions qu'elle peut entreprendre. Par exemple, DroidMat [35] effectue le suivi des appels API à partir des différents composants de l'application. DroidAPIMiner [36] se concentre également sur les appels API considérés comme critiques par les auteurs et incluant les arguments utilisés lors de l'appel pour former un modèle de détection utilisant les techniques de l'apprentissage automatique. **DroidMiner** [37] construit des graphes de comportement appelés CBG (Component Behaviour Graphs) basés sur la communication entre les fonctions API et les ressources Android sensibles et différents algorithmes de classification tels que SVM ou Random Forest. **DroidSIFT** [38] est un classificateur basé sur la sémantique, qui utilise des graphes de dépendance d'API pour former un classificateur Naïf de Bayes.

2.1.9 Opcode et autres méthodes de détection

D'autres possibilités impliquant également les caractéristiques statiques extraites reposent sur l'utilisation de codes d'opération ou de commandes système. Droid Analytics [39] est un système conçu pour aider à récupérer les informations de niveau **d'opcode** provenant de logiciels malveillants Android.

Dans Dendroid **[**40**]**, les structures de code servent à comparer des échantillons et à former un algorithme de classification.

DroidLegacy [41] procède à la classification des familles en extrayant des signatures que nous pouvons trouver dans des applications malveillantes et qui peuvent être utilisées dans des échantillons réemballés, à l'origine bénins. Ensuite, des mesures de similarité sont utilisées pour comparer des échantillons.

En plus, il existe un nombre important d'utilitaires centrés sur les processus de rétroingénierie capables d'extraire des groupes de caractéristiques.

Smali/BakSmali **[**42**]** est un autre outil bien connu, un assembleur et un désassembleur des fichiers DEX (Dalvik Exécutable) contenant le bytecode de l'application.

Apktool [43] permet de décoder les ressources contenues dans le fichier exécutable. Il fournit également d'autres utilitaires puissants, tels que le remballage de l'échantillon.

Dex2jar [44] est destiné à convertir les fichiers de bytecode de Dalvik en fichiers Java compilés avec extension JAR. Cela permet de décompiler le code ultérieurement à l'aide d'autres outils tels que Jd-gui [45] qui reconstruit le code et permet de le visualiser.

2.2 Solutions des entreprises

Les solutions des entreprises proviennent de deux sources que nous allons développer dans deux sous-sections :

1. les applications provenant du Google Play Store.
2. la solution proposée par Google Inc.

2.2.1 Applications provenant du Google Play Store

Les solutions de détection de logiciels malveillants proposées par les entreprises et disponibles sur le Google Play Store sont bien souvent payantes pour avoir accès à l'intégralité de leurs fonctionnalités. En plus de cet inconvénient pour le grand public, les méthodes de détection ne sont pas révélées. Cette opacité ne permet pas de se baser sur ces travaux pour développer notre propre méthode de détection, mais permet de savoir ce qui se fait sur le marché afin de nous éclairer et de nous guider pour notre étude. La grande majorité de ces applications Android fournit des fonctionnalités supplémentaires à la détection de logiciels malveillants, comme un scanneur réseau, un service VPN, un AppLock, un scanneur de permissions, un antivol. Généralement, ces fonctionnalités s'obtiennent via un abonnement mensuel ou annuel payant, laissant ainsi une version épurée de détection de logiciels malveillants gratuits. Leur source de revenue est alors la publicité. Le Tableau 2.1 fournit une liste non exhaustive des applications Android les plus populaires sur le Google Play Store. Nous avons obtenu ce tableau comparatif par nos propres expériences. Chaque application a été téléchargé et les informations provenant du tableau sont les informations descriptives de chaque application, selon le Google Play Store en automne 2019. Les prix affichés sont les prix proposés par chaque publieur d'application.

Tableau 2.1 Présentation des différentes solutions d'entreprises disponibles au grand public

Nom Google Store	Publieur	Disponibilité d'une version gratuite	Prix de la version payante	Nombre de téléchargements (en juin 2019)
Security Master	Cheetah Mobile	Oui	20 $ / an	500 M+
AVG Antivirus	AVG Mobile	Oui	16 $ / an	100 M+
Avast Antivirus	Avast Software	Oui	13 $ / an	100 M+
Kaspersky Mobile Antivirus	Kaspersky Lab.	Oui	20 $ / an	50 M+
Security Center	Hyper speed	Oui	No	50 M+
Mobile Security	ESET	Non	7 $ / an	10 M+
McAfee	McAfee LLC	Non	41 $	10 M+
Malwares Bytes	Malwarebytes	Non	14 $ / an	10 M+
Norton Antivirus et Sécurité	NortonMobile	Oui	20 $ / an	10 M+
Sophos Mobile Security	Sophos Limited	Oui	No	1 M+

2.2.2 Solution proposée par Google Inc.

Google Inc. prend la sécurité de la plateforme Android très au sérieux et propose de nombreuses informations afin de paraître transparent. En effet, nous pouvons retrouver des « rapports de transparences » [46] comprenant de nombreuses statistiques pour la sécurité de l'écosystème

Android. Cependant, il s'agit d'introduire la notion de PHA[2] (Potentially Harmful Application) qui réside dans des applications Android représentant potentiellement un risque. En d'autres termes, la probabilité que cette application soit considérée comme application malveillante est plus élevée que la normale. Nous retrouvons dans ces rapports trimestriels, par exemple, le pourcentage des terminaux avec des PHA en fonction de la version Android (Voir Annexe B) ou bien encore le pourcentage d'installations de PHA par catégorie (Voir Annexe E). De plus, il y a eu, fin 2017 et début 2018, un changement majeur dans leur politique de gestion de PHA comme nous pouvons le voir dans le blog Google Inc. des développeurs Android [47]. Les conséquences de cette nouvelle politique sur le nombre d'applications disponibles dans le Google Play Store sont clairement visibles (Voir Annexe F).

Même le géant Google Inc. ne peut se prononcer de manière certaine sur un taux de détection de 100%. Bien qu'en 2019 Google Inc. ait fait d'énormes progrès, son système de détection **Google Bouncer** n'en est qu'à ses débuts. En effet, une annonce officielle de son existence en février 2012 [48] a provoqué un engouement dans le domaine de la recherche. Depuis une multitude de chercheurs a étudié Google Bouncer pour en savoir plus. Le 4 juin 2012, Jon Oberheide et Charlie Miller [49] ont présenté des résultats intéressants. Ils ont pu explorer le système en utilisant un système de commandes pour rechercher des attributs de l'environnement Bouncer, tels que la version du noyau en cours d'exécution, le contenu du système de fichiers ou des informations sur certains des périphériques émulés par l'environnement Bouncer. En regardant dans le répertoire /sys, les auteurs ont remarqué le répertoire qemu_trace, exposant ainsi le fait que leur application soit exécutée dans l'environnement émulé basé sur QEMU de Google Bouncer. De cette manière, Google Bouncer teste les applications en les exécutant dans un environnement virtualisé, c'est-à-dire un téléphone simulé créé dans un logiciel qui analyse automatiquement les applications pour en observer le comportement réel sur les terminaux des utilisateurs, avant de les approuver pour la publication sur le Google Play Store. Une autre de leurs découvertes est que les instances de téléphones virtualisés dans Google Bouncer sont associées au même compte et ont exactement un

[2] Google Developers (2019), Potentially Harmful Applications (PHAs), https://developers.google.com/android/play-protect/potentially-harmful-applications, (Consulté en Juin 2019)

contact et deux photos sur le périphérique simulé. De ce fait, il est pensable que les personnes mal intentionnées puissent esquiver cette protection en ne dévoilant pas son caractère malveillant durant l'émulation, comme le mentionne le blog TrendMicro [50] le 20 juillet 2012. Nous observons également que Google Bouncer n'effectuait que l'analyse dynamique.

Il a été nécessaire de n'attendre que quelques semaines pour que le 22 Septembre 2015, l'équipe de recherche d'ESET [51] découvre une application Android malveillante téléchargeable depuis le Google Play Store montrant ainsi qu'elle a échappé au Google Bouncer. Cette dernière utilisait le nom d'un jeu classique en y ajoutant une autre application à l'intérieur, qui une fois installée à l'insu de l'utilisateur, roulait en arrière-plan comme un service. Nous pouvons retrouver régulièrement de nouvelles annonces similaires comme celle du 21 Octobre 2018 [52]. Encore plus récemment, le 5 Juin 2019 [53], des personnes mal intentionnées qui se trouvaient être des fabricants de téléphones chinois, ont conçu des adwares et se sont données du mal pour cacher leur méfait. Il s'agissait d'un plugin de clavier qui était une application à part entière. D'autres encore ont infecté les téléphones dès leurs sorties d'usine comme l'a révélé le magazine Bloomberg Businessweek [54] en octobre 2018. Si le matériel est compromis, aucun logiciel, si parfait soit-il, ne pourra rivaliser.

Cette série de faits sur la famille PHA continue avec la famille Triada qui a été découverte pour la première fois dans un article de blog sur le site Web de Kaspersky Lab en mars 2016 [55]. Le but principal des applications **Triada** était d'installer des applications Android anti-spam sur un appareil affichant des annonces. Les créateurs de Triada ont collecté les revenus des annonces affichées par les applications de spam. Les méthodes utilisées par Triada étaient complexes et inhabituelles pour ces types d'applications. Les applications Triada ont été contraintes de s'adapter pour passer par une porte dérobée de l'image système. Google Inc. donne des informations techniques supplémentaires concernant Triada pouvant être trouvées sur le blog officiel de sécurité de Google® [56]. Contre toutes ces nouvelles menaces de plus en plus virulentes, Google Inc. a revu sa politique et a instauré **Google Play Protect [**57**]**, la plateforme de protection intégrée contre les logiciels malveillants pour la plupart des terminaux Android. Cette dernière est soutenue par les techniques d'apprentissage machine afin d'analyser plus de 50 milliards d'applications par jour.

Pour pallier les déficiences de la plateforme Google Inc., de nombreuses solutions complémentaires voire alternatives sont proposées par la communauté scientifique en libre accès sur GitHub disponibles pour l'analyse d'applications Android. Il y en a tellement qui sont apparues que le besoin de les répertorier s'est fait ressentir. Nous pouvons trouver ce répertoire GitHub [58] bien connu qui regroupe une liste d'outils pour la sécurité Android. Nous retrouverons dans la section suivante différents types d'analyses du système d'exploitation Android.

2.3 Approches et outils de recherche

Cette sous-section présente tout d'abord une vue d'ensemble des solutions de détection de logiciels malveillants proposées par le milieu académique. Par la suite, il est décrit les différentes approches statiques et dynamiques suivies de leurs avantages et de leurs inconvénients respectifs.

2.3.1 Vue d'ensemble

La vue d'ensemble présente :

1. Une vue d'ensemble générale des solutions proposées
2. Une vue d'ensemble des solutions se basant sur l'apprentissage machine

2.3.1.1 Vue d'ensemble générale

La taxonomie et la comparaison qualitative des techniques d'analyse de sécurité des logiciels Android, publiées en octobre 2016 par Sadeghi et al. [59] sont un bon début dans l'étude des méthodes de détection de logiciels malveillants sur mobiles. Les articles de revue de la littérature sont excellents pour donner une vue d'ensemble du domaine puisqu'ils ne s'arrêtent pas une méthode de détection spécifique. Tout d'abord, il s'agit de préciser que cette étude se restreint à l'étude des méthodes destinées à Android. Cela s'explique par la part de marché majoritaire des téléphones intelligents utilisant ce système (80%) comparativement à iOS (15%) et les autres. De plus, les téléphones qui ne sont pas intelligents ne sont pas pris en compte, puisque la majorité des logiciels malveillants s'attaquent aux téléphones intelligents. L'utilisation d'Android est aussi préférée par les chercheurs car c'est un OS ouvert permettant ainsi plus de possibilités. Les auteurs de la taxonomie traitent à la fois des méthodes de détection de logiciels malveillants et des méthodes de détection de vulnérabilité. Ils dressent de nombreux tableaux dont les catégories sont

édifiées par la lecture des articles et de précédentes revues de littérature et taxonomies. Les articles étudiés sont classés en fonction des menaces considérées (fuite d'information, spoofing, élévation de privilège, etc.), en fonction des spécifications du problème (couche de déploiement de la solution, finesse de la menace, objets étudiés dans le système, etc.), en fonction de l'approche (analyse statique contre dynamique ou hybride) et des méthodes supplémentaires (apprentissage automatique, analyse formelle) et du niveau d'automatisation. Cependant, les méthodes basées sur l'analyse dynamique sont à nouveau séparées selon le niveau d'inspection (applicatif, noyau, machine virtuelle) et la génération des entrées de l'application, tandis que les méthodes statiques peuvent être basées sur différentes structures de données, peuvent représenter le code de diverses manières et ont une sensibilité d'analyse variable.

La revue complète et l'analyse des applications malveillantes pour smartphones M. Talal et al. [60] vise à donner un aperçu des analyses de détection de logiciels malveillants en sondant et en faisant la taxonomie de la littérature et des solutions de sécurité à jour disponibles qui sont proposées par d'autres chercheurs. Une catégorisation des solutions de sécurité disponibles a ainsi été faite basée sur leurs mécanismes. Quatre catégories distinctes ont été établies, à savoir 1) les révisions et enquêtes, 2) les solutions de sécurité pour smartphones, 3) les smartphones études de sécurité et classement, 4) le regroupement et la classification. Les tableaux présentent des informations détaillées des techniques de détection de logiciels malveillants se basant sur de l'intelligence artificielle ou non. Ainsi pour la trentaine d'articles dont la détection se base sur une intelligence artificielle, nous pouvons trouver la technique de classification utilisée (Classification ou Clustering), les algorithmes d'apprentissage machine (Random Forest, Support Vector Machine (SVM), régression logistique etc…), les jeux de données (standard, collectés par les auteurs) et bien d'autres. Ensuite, plusieurs questions et lacunes personnelles des chercheurs sont soulignées dans l'analyse et les recommandations pour plusieurs parties (les chercheurs, les utilisateurs, les développeurs et enfin les entreprises et les gouvernements) pour faire avancer les recherches dans chaque domaine respectif.

2.3.1.2 Vue d'ensemble des solutions se basant sur l'apprentissage machine

A. Souri et R. Hosseini [61] est une revue de la littérature sur les approches de détection des logiciels malveillants utilisant l'apprentissage machine. Les articles étudiés ont été classés en deux

catégories principales. La première est celle qui regroupe les approches basées sur des signatures et la deuxième regroupe les approches basées sur les comportements. Les méthodes de détection de logiciels malveillants ont été comparées et analysées en fonction des facteurs tels que les méthodes de classification, les méthodes d'analyse des données, la quantité de données d'entrée et le facteur de précision. Une grande partie des articles sélectionnés utilisent des techniques de détection de logiciels malveillants sur Android basées sur le comportement. Les résultats expérimentaux montrent que 65% de ces approches de détection ont utilisé la méthode d'analyse dynamique alors que les approches basées sur les signatures sont plus orientées vers des méthodes de détection statique. Enfin, les auteurs affirment que l'utilisation d'algorithmes méta-heuristiques peut réduire le temps d'exécution et augmenter la précision du processus d'exploration des données.

Toutes les méthodes basées sur l'apprentissage supervisée, quel que soient les algorithmes utilisés, requièrent des jeux de données comportant des caractéristiques extraites de logiciels malveillants et d'applications bénignes afin d'entraîner leurs modèles.

Dataset

Les outils de détection de logiciels malveillants basés sur l'apprentissage automatique nécessitent la formation de jeux de données dont les échantillons sont étiquetés comme malveillants ou bénins. L'utilisation de ces ensembles de données est essentielle pour créer des méthodes fiables de détection et de classification des logiciels malveillants. À cette fin, différents jeux de données d'échantillons Android ont été rendus publics au cours des dernières années.

Le projet **Génome** [62] a été lancé en 2012 contenant 1,260 logiciels malveillants, mais a été abandonné en 2015.

Drebin [63] comporte 5,560 logiciels malveillants de 179 familles différentes ainsi que des applications bénignes extraites de différentes sources. Drebin constitue une base de données bien connue utilisée par de nombreux chercheurs pour tester la précision de leurs propres méthodes de détection., mais est trop ancienne car les logiciels malveillants ont été collectés d'août 2010 à octobre 2012.

Contagio [64] publie périodiquement de nouvelles applications malveillantes telles quelles.

Plus récemment, **l'Android Malware Dataset** [65] a été publié. Ce jeu de données contient 24, 553 échantillons collectés de 2010 à 2016 de 71 familles de logiciels malveillants.

En plus des jeux de données, il existe également des services en ligne qui permettent de récupérer des applications bénignes out malveillantes. Par exemple, de nombreux chercheurs collectent des échantillons directement auprès de magasins d'applications, tels que Google Play ou Aptoide. Bien que ces jeux de données offrent des ensembles d'échantillons bruts, d'autres projets fournissent des fonctionnalités ou des journaux déjà extraits d'échantillons. C'est le cas de VirusTotal [66] qui est un agrégateur de résultats en ligne composé d'une soixantaine d'antivirus. À l'aide du hash de l'application, nous pouvons trouver le nombre d'antivirus qui considère cette application comme malveillante.

Le portail **Koodous** [67] propose un vaste ensemble d'échantillons malveillants et inoffensifs uniquement à des fins de recherche, incluant dans ce cas un rapport sur les fonctionnalités.

- 55,535,003 applications en mars 2020.
- Entre 20 000 et 50 000 nouvelles applications journalières
- 15,498,822 logiciels malveillants en mars 2020.
- 252,363 PHA en mars 2020.

Le projet **AndroZoo** [68] propose un vaste ensemble de données contenant plus de 5 700 000 échantillons en cours d'analyse avec différents moteurs antivirus.

- 10,502,608 applications en mars 2020.
- Différentes sources (Google Play, Anzhi, AppChina, 1mobile, torrents, Genome, FDroid…)

F.-X. Geiger et I. Malavolta [69] proposent une étude de 36 bases de données d'applications Android. Les auteurs ont classé les bases de données trouvées en trois catégories. La première catégorie contient les jeux de données qui utilisent les données d'applications marché (comme Google Play). La deuxième catégorie rassemble les jeux de données fournissant des APK exécutables (comme Androzoo) et enfin la troisième catégorie regroupe les ensembles de données

ayant accès au code source basé sur FDroid. Pour eux, avoir à la fois un jeu de données complet de toutes les applications disponibles et avoir accès au code source n'est pas conciliable.

Omnidroid [70] est une base de données contenant un ensemble de données complet sur les caractéristiques dynamiques et statiques d'applications Android obtenues grâce à l'extracteur de caractéristiques AndroPyTool. Vous trouverez en Annexe G, les filtres et processus appliqués pour construire le jeu de données Omnidroid. Omnidroid est composée de 22 000 applications réparties en deux groupes : 11 000 applications malveillantes et 11 000 applications bénignes. Chaque application possède un total de 26 000 caractéristiques statiques et 5,932 caractéristiques dynamiques. Les caractéristiques de cet ensemble de données la rendent appropriée pour être utilisée en tant qu'ensemble de données de référence pour la communauté scientifique pour tester différents algorithmes et techniques. Afin de rendre le jeu de données Omnidroid facile à utiliser, toutes les données sont fournies aux formats json et csv et sont accessibles publiquement sur Aida[3] sous une licence internationale Creative Commons Attribution-Non Commercial-Share Alike 4.0. Les applications qu'elle contient proviennent de deux sources : Koodous (–2020) [67] et Androzoo (2011 – 2020) [68].

SandDroid [71] fournit également un jeu de données de caractéristiques extraites d'échantillons de logiciels malveillants et notamment des analyses statiques et dynamiques. Si nous le comparons à Omnidroid, SandDroid inclut uniquement des échantillons de logiciels malveillants et le jeu de fonctionnalités n'est pas aussi étendu.

2.3.2 Approches par analyse statique

Les méthodes d'analyses présentées peuvent être regroupées en deux groupes : les méthodes d'analyses statiques et les méthodes d'analyses dynamiques.

Les méthodes d'analyses statiques ne nécessitent pas l'exécution de l'application sur un terminal. Elles s'intéressent plutôt à son code plus qu'à son comportement réel lors de son exécution puis

[3] Applied Intelligence And Data Analysis Research Group, https://aida.ii.uam.es/datasets/ (Consultée le 7 juin 2019)

que son code est censé être fidèle aux fonctionnalités de l'application. Nous allons maintenant présentées différentes solutions statiques proposées par la communauté scientifique.

La détection de logiciels malveillants dédiée aux terminaux mobiles [72] a été publiée au service de publication de l'École Polytechnique de Montréal nommé PolyPublie[4]. Sa particularité réside dans la méthode de détection statique basée sur les 151 autorisation système Android. Premièrement, deux modèles neuronaux ont été entraînés avec comme caractéristiques d'entrées un vecteur de 151 dimensions; chaque dimension correspondant à une autorisation système Android. Le premier modèle comporte un ensemble d'entraînement de 22 000 applications dont 17 000 sont bénignes. Les résultats ont confirmé qu'il était préférable d'avoir un ensemble d'entraînement équilibré. Ceci a mené à son deuxième modèle qui a été basé sur l'entraînement d'un ensemble de 10 000 applications composé de 5000 applications bénignes et de 5000 applications malveillantes. La précision annoncée sur son ensemble de test est de 94,42%. Nous devons mentionner que les applications sur lesquelles le modèle a été entraîné sont des applications malveillantes provenant de la base de données Drebin [63] datant entre 2010 et 2012. À la vitesse qu'évolue ce domaine, il s'agit donc d'applications malveillantes très datées. Les applications bénignes proviennent du top 500 de chaque catégorie du Google Play Store pendant l'été 2018. Aucune vérification de sécurité n'a été proposée pour vérifier qu'il s'agissait bien d'applications non malveillantes. Par la suite, Arthur a essayé une méthode de régression justifiée de deux manières différentes. La première stipule que les méthodes de régression comportent des algorithmes différents des méthodes de classification et la seconde est de traiter la classe de l'application comme un nombre compris entre -100 et 100 au lieu d'un binaire équivalent à bénigne ou malveillante permettant ainsi une plus grande flexibilité pour la combinaison avec d'autres méthodes et l'indication d'un indice de certitude à l'utilisateur. À notre connaissance, c'est la seule méthode de détection s'aventurant dans une méthode de régression. Enfin, son modèle neuronal a été construit avec WEKA (Waikato Environment for Knowledge Analysis), un projet en libre accès [73].

---

[4] POLYPUBLIE, répertoire institutionnel de l'École Polytechnique de Montréal (2020), https://publications.polymtl.ca/, (Consulté en Avril 2020)

IntelliAV [74] est un système de détection d'applications malveillantes **sur l'appareil** qui utilise l'analyse statique couplée à l'apprentissage automatique. L'application est disponible sur le Google Play Store ce qui permet d'éventuelles comparaisons de performance. A partir d'un set d'entraînement et de validation de 19,722 applications dont 9,664 malveillantes, les auteurs ont obtenu un taux de vrai positif de 92.5% et un taux de faux positif de 4.2% sur **l'ensemble de validation** avec 1 000 attributs générés par l'entraînement ce qui est beaucoup plus petit que les méthodes d'analyses statiques basées sur l'apprentissage automatique hors-appareil. Les auteurs ont aussi évalué leur modèle sur un set de 2,898 applications bénignes et 2,311 applications malveillantes provenant de VirusTotal de février 2017. Une application malveillante est considérée comme telle si au moins cinq antivirus la détectent comme malveillante. Le modèle a atteint une justesse de 71.96%

[75] lit les règles de sécurité KSL (Kirin Security Language), un langage spécifiquement créé pour Kirin, et les compare à la configuration de l'application. C'est la première méthode statique qui ait existé. Son avantage est qu'elle ne nécessite aucune permission, mais en revanche il s'agit d'une ancienne méthode obsolète aujourd'hui.

MaMaDroid [76] détecte les logiciels malveillants d'un point de vue comportemental, modélisé comme la séquence d'appels d'API abstraits. Son système est basé sur une analyse statique qui collecte les appels d'API effectués par une application pour leur classe, leur package ou leur famille et construit un modèle à partir de leurs séquences obtenues à partir du graphe d'appels d'une application sous forme de chaînes de Markov. Cela garantit que le modèle résiste mieux aux modifications de l'API et que le jeu de fonctionnalités est d'une taille gérable. MaMaDroid a été testé à l'aide d'un ensemble de données de 8 500 applications bénignes et de 35 500 applications malveillantes collectées sur une période de six ans avec F-mesure atteignant 99%. Une des conséquences est que nous sommes incapables de prédire sur un autre jeu de données inconnu. MaMaDroid conserve ses capacités de détection deux ans après la formation atteignant une F-mesure de 87%. Dans le but de déterminer si l'efficacité de MaMaDroid provient de l'abstraction de l'API ou de la modélisation de séquençage, une variante de MaMaDroid a été proposée qui utilise la fréquence d'appels d'API abstraits au lieu des séquences. Cette variante n'est pas aussi précise et ne parvient pas à détecter le code malveillant comprenant des appels d'API utilisés autant ou plus fréquemment par des applications anodines.

DroidSieve [77] avance l'idée qu'une combinaison de plusieurs caractéristiques est cruciale pour une détection robuste des logiciels malveillants simples et obfusqués. Ainsi, des caractéristiques syntaxiques telles que les appels API et les permissions systèmes sont intégrées dans leur méthode de détection. Bien qu'étant déjà connues pour détecter les logiciels malveillants non obfusqués, ces caractéristiques ont été utilisées pour construire un classificateur qui est robuste à la fois pour les anciens logiciels malveillants ainsi que pour les nouveaux qui ont tendance à être de plus en plus obscurcis. Pour enrichir l'ensemble des fonctionnalités syntaxiques, de nouvelles caractéristiques à base d'intentions explicites, de méta-informations et de fichiers DEX ont été rajoutées. Les auteurs ont aussi créé un classement des caractéristiques les plus pertinentes pour la détection de logiciels malveillants. Les permissions systèmes et les intentions arrivent en tête. Ces caractéristiques statiques sont les vecteurs d'entrée de leur modèle de détection basé sur l'apprentissage machine et plus spécifiquement sur les modèles à vecteur de machines.

FlowDroid [78] effectue du taint tracking (suivi de traces) sur les applications Android. Il s'agit de la première méthode utilisant ce principe comportemental et est entièrement en libre accès. Ce qui justifie son grand nombre de citations (1,243 citations en août 2019). Le taint tracking fournit des informations détaillées sur les différents flux d'informations qui se produisent pendant le cycle de vie de l'application.

RevealDroid [79] est un outil de détection de logiciels malveillants Android dont le code est disponible[5]. Il consiste en une approche d'apprentissage automatique basée sur l'utilisation des APIs Android, y compris la résolution des APIs appelées à l'aide de la réflexion et des appels de fonction (comme les appels systèmes) effectués par des fichiers binaires natifs au sein d'une application Android. Aucun travail précédent n'a inclus l'extraction de fonctionnalités du code natif pour détecter les logiciels malveillants ce qui constitue leur innovation principale.

Z. Ma et al. [80] proposent une méthode de détection de logiciels malveillants Android basée sur l'apprentissage machine et aussi sur FlowDroid dont le code source a été modifié pour remplacer

---

[5] Joshua Garcia, Mahmoud Hammad, and Sam Malek (2018), Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware, https://www.ics.uci.edu/~seal/projects/revealdroid/index.html, (Consulté en Juin 2019)

l'ancienne méthode de taint tracking par leur propre méthode. C'est FlowDroid qui a permis de construire le graphe de flux de contrôle qui a servi pour obtenir l'API information. À partir de cette dernière, les auteurs ont construit trois types de jeux de données API. Ce sont les premiers à construire les ensembles de données de séquence API d'une application Android. Les auteurs ont estimé avoir assez de données pour faire de l'apprentissage profond et utiliser des cellules LSTM (Long Short Term Memory) dans une architecture RNN (Recurrent Neural Network). Les expériences ont été menées sur 10 000 applications bénignes et 11 000 applications malveillantes, ce qui est convenable pour faire de l'apprentissage machine mais léger pour parler d'apprentissage profond. Enfin, les auteurs ont utilisé une méthode largement utilisée pour la combinaison de modèles qui est boosting, un méta-algorithme d'ensemble d'apprentissage automatique pour améliorer la précision et qui permet d'identifier les parties d'apprentissages faibles afin de mieux les faire apprendre.

Maldozer [81] repose sur la classification des séquences brutes des appels aux méthodes API à l'aide de techniques d'apprentissage profond. Maldozer peut servir de système de détection de logiciels malveillants sur des serveurs, mais également sur des terminaux mobiles et même des périphériques IoT.

AndroGuard [82] est une bibliothèque Python qui extrait des informations variées à partir de code, de ressources ou du fichier AndroidManifest.xml d'Android**.**

2.3.3 Approches par analyse dynamique

Toutes les méthodes de détection qui ont été présentées précédemment sont basées sur des techniques d'analyse statique. En d'autres termes, elles ne nécessitent pas l'exécution du code de l'application tandis que les techniques d'analyse dynamique si. Bien que les fonctionnalités statiques soient faciles à extraire et fournissent des informations détaillées, elles présentent certains **inconvénients**. Des techniques telles que la cryptographie, l'obscurcissement du code, le chargement dynamique du code ou la réflexion peuvent être mises en œuvre pour éviter la détection par l'analyse statique. De manière générale, il est impossible avec une analyse statique, de suivre le comportement réel d'un échantillon lorsqu'il est exécuté. Ce qui empêche de révéler le contenu malveillant d'une instance de logiciels malveillants dans de nombreux cas. Un nombre

important de recherches tente de contourner ce problème en surveillant les actions de l'application dans un émulateur ou bien sur un appareil réel.

TaintDroid [83] introduit et prototype une méthode très reprise par la suite, le taint tracking. En effet TaintDroid décrit une nouvelle méthode de détection des logiciels malveillants par analyse dynamique utilisant le taint tracking (le taint tracking n'implique pas l'analyse dynamique). Un des inconvénients de TaintDroid est que les auteurs ont dû explorer manuellement les applications. Ce qui limite grandement le nombre d'applications que nous pouvons manipuler. En d'autres termes, leur méthode n'est pas automatique. De plus, cette méthode date de 2010 et est pionnière dans l'analyse dynamique.

DroidScope [84] est un outil d'instrumentation binaire pour la criminalistique des logiciels malveillants. Il surveille le comportement des applications sur trois couches de plateformes différentes. Comme TaintDroid, DroidScope ne permet pas une exploration automatique et nécessite un effort manuel important.

AppsPlayground [85] reprend le concept de taint tracking de TaintDroid et développe une méthode intelligente de génération d'input et de parcours d'application pour l'analyse dynamique, ce qui la rend automatique. En revanche, comme TaintDroid, il requiert une modification du système Android pour traquer les données via le taint tracking, et les tests ne sont effectués que sur émulateur et non sur un téléphone réel.

CopperDroid [86] est un système dynamique basé sur une machine virtuelle dont le but est de reconstruire de manière uniforme et automatique les comportements d'applications Android malveillantes en se basant sur une méthode automatique d'association entre les appels systèmes et les interactions IPC (Inter-Process Communication) et RPC (Remote Procedure Call).

Z.-G. Chen et al. [87] proposent la détection des systèmes basés sur l'exploration de données par ransomware pour la détection automatique. Les comportements réels des applications sont contrôlés et générés dans l'API call flow graph sous la forme d'un ensemble de fonctionnalités.

DroidScribe [88] utilise l'analyse dynamique et plus particulièrement les appels d'API invoqués lors de l'exécution via le protocole Binder et l'apprentissage automatique, pour identifier la famille à laquelle appartient l'application malveillante. Cependant, DroidScribe ne fait pas de la détection d'applications malveillantes. En d'autres termes, cette méthode ne détermine pas si une application

est bénigne ou malveillante. De plus comme les appels d'API font partie du flux d'appels système, DroidScribe pourrait être vulnérable aux attaques de mimétisme.

EMULATOR vs REAL PHONE [89] propose un système permettant d'effectuer les analyses adaptées aux émulateurs sur des téléphones intelligents réels avec quelques adaptations, afin de pallier les différentes limitations dues aux émulateurs. Il se base sur Dynalog pour l'extraction de caractéristiques et Monkey pour l'exploration automatique des applications. En effet, il est possible d'éviter la détection lors d'une analyse dynamique sur émulateur. Les logiciels malveillants sont de plus en plus innovants et possèdent des techniques de détection d'émulateurs (par exemple empreinte digitale) pour dissimuler la partie malveillante du code tant que l'exécution se fait sur un émulateur. Enfin, il est proposé une étude détaillée des différences entre les environnements d'exécution pour conclure qu'il est préférable de rouler la détection sur un appareil réel.

DroidCat [90] est une approche de classification des applications Android basée sur un nouvel ensemble de fonctionnalités qui capture les comportements des applications au niveau application au moment de l'exécution. Les fonctionnalités ont été déterminées à partir d'une étude révélant des différences de comportement entre les applications bénignes et malveillantes en ce qui concerne les appels systèmes et les appels ICC. Cette approche se veut être robuste vis-à-vis des techniques d'obscurcissement (des ressources et des appels systèmes) et d'autres techniques d'évasion. Le code source est disponible en note de bas de page[6].

DroidBox[7] [91] permet de surveiller une large série d'événements tels que des accès à des fichiers, du trafic réseau ou des fichiers DEX chargés dynamiquement au moment de l'exécution. En revanche, la faiblesse de DroidBox réside dans le fait que son implémentation n'a pas été faite dans la version la plus récente de la plateforme Android. En effet, DroidBox utilise l'API 16 mais qui couvre 99,6% des téléphones intelligents selon Android Studio (Voir Annexe C).

---

[6] Haipeng Cai (2019), DroidCat - Dynamic Android malware detection and categorization based on behavioral characterization, https://bitbucket.org/haipeng_cai/droidcat/src/master/, (Consulté le 19 juin 2019)

[7] Droidbox source code repository, 2011, https://github.com/pjlantz/droidbox, (Consulté le 21 juin 2019)

2.3.4 Avantages et inconvénients

Les avantages et les inconvénients des deux méthodes statiques et dynamiques sont présentées dans le Tableau 2.2.

Tableau 2.2 Les avantages et les inconvénients des méthodes statique et dynamique

	Statique (code)	Dynamique (éxécution)
Avantages	• Nécessite une capacité de calcul faible • Extraction facile des caractéristiques • Rapidité d'extraction • Ne dépend pas de l'environnement	• Obscurcissement du code • Chargement dynamique • Réflexion • Chiffrement
Inconvénients	• Obscurcissement du code • Chargement dynamique • Réflexion • Chiffrement • Code natif	• Nécessite une capacité de calcul élevée • Extraction difficile des caractéristiques • Lenteur d'extraction • Détection de faux environnement

- **Obfuscation [92]** :cette technique permet de réduire la taille de l'application en réduisant le nombre des classes. Le code devient illisible pour l'être humain et nécessite l'utilisation de méthodes de reverse engineering pour revenir à une version lisible.
- **Réflexion [92]** : permet de faire un pont entre le statique et le dynamique. Elle permet de charger différents codes au moment de l'exécution. En fonction du terminal Android et de sa version Android, nous allons faire appel à telle fonction. Ce qui permet de développer des applications pour Android 4.4 (API 16).
- **Chiffrement** [93] : le chiffrement permet de rendre le code inaccessible à toute personne ne possédant pas la clé de déchiffrement. Ceci permet de protéger les informations sensibles ainsi que du code appartenant à une entreprise qui ne souhaite pas que celui-ci

soit divulgué. Au moment de l'exécution de l'application, un déchiffrement est réalisé à l'aide de la clé privée. L'utilisation du chiffrement réduit les performances mais est plus sécuritaire puisqu'elle empêche que des informations sensibles ne soient interceptées lors de leur transit sur le réseau. En revanche, cette technique peut être utilisée pour exécuter du code malveillant au moment de l'exécution de l'application.

- **Code natif [94] :** étudie le byte code qui est exécuté par JVM. Le code natif a davantage de fonctionnalités comme accéder directement à la mémoire. Il est aussi chargé dynamiquement.

D'autres techniques pour échapper aux analyses statiques et dynamiques sont :

- **Bombe logique [95] :** exécution d'un code lorsque certaines conditions sont réunies. Les bombes temporelles sont un type de bombe logique qui exécute du code à partir d'une certaine date.
- **Serveur distant [96] :** le terminal mobile communique avec un serveur distant qui lui donne des commandes à exécuter. Si l'émulateur ou le téléphone n'est pas connecté à Internet il n'y aura aucun résultat. L'environnement d'exécution peut faire varier considérablement les résultats.

2.3.5 Approches par analyse hybride

MADAM [97] est un framework utilisant l'apprentissage automatique pour détecter les logiciels malveillants. Il les classifie à partir des comportements suspects observés à différents niveaux d'Android : noyau, application, utilisateur et package. L'ancienne version de l'article citée par Alzaylaee et al. [98] ne proposait qu'une validation sur peu de logiciels malveillants analysés sur émulateurs, mais la version améliorée de 2018 offre des expérimentations sur terminaux réels plus poussées sur 2,800 logiciels malveillants de 125 familles différentes provenant de trois bases de données. De plus, 9804 applications légitimes ont été testées pour montrer que le taux de faux positifs est très bas : 2.8*10−5. MADAM nécessite des privilèges administrateurs sur le téléphone utilisé puisqu'il fonctionne au niveau noyau. Ainsi, les auteurs de MADAM précisent donc que leur solution n'est pas destinée au grand public mais cherche à prouver la force d'une telle approche (multi-niveaux, dynamique et sur l'appareil). Ceci emmènerait éventuellement à faire en

sorte que les constructeurs de systèmes pour les smartphones intègrent MADAM dans leur OS. Ceci constitue une approche invasive qui peut ne pas correspondre à tous.

SAMADroid [99] utilise aussi l'apprentissage automatique pour détecter les logiciels malveillants. Il fonctionne à la fois sur l'hôte local pour effectuer l'analyse dynamique et l'hôte distant pour obtenir l'analyse statique et les prédictions. L'application client SAMADroid est développée pour les terminaux Android. La base de données pour les entraînements des réseaux de neurones est Drebin [63] (2010-2012).

Martín A., Lara-Cabrera R. et Camacho D. [100] présentent deux outils qui sont d'une grande aide pour notre propre méthode de détection, le framework **AndroPyTool** et la base de jeux de données **Omnidroid.** Il existe actuellement une panoplie d'outils d'extraction de fonctionnalités statiques et dynamiques à partir d'applications Android. Cependant, chacune d'entre elles se concentre sur un type précis de caractéristiques. L'obtention d'un ensemble complet de fonctionnalités statiques et dynamiques devient alors une tâche demandant de longues investigations. Cela implique de configurer chaque outil avec ses fichiers de configuration et son environnement respectif, ainsi que de regrouper les résultats obtenus individuellement pour constituer un jeu de données complet. AndroPyTool est une solution tout-en-un destinée à la communauté de chercheurs s'intéressants sur les applications malveillantes. Développé en Python, il est capable d'exécuter une extraction complète de caractéristiques statiques et dynamiques. Il intègre les outils d'analyse des logiciels malveillants Android les plus utilisés (FlowDroid [78], DroidBox [91], AndroGuard [82] et Strace [101]) pour effectuer une inspection du code source et pour récupérer des informations sur le comportement lorsque l'échantillon est exécuté dans un environnement contrôlé. Le code source du projet est hébergé sur GitHub[8]. De plus, AndroPyTool peut être utilisé via un conteneur Docker, ce qui permet d'exécuter l'outil en seulement deux étapes. Pour savoir comment installer et lancer l'outil en utilisant le code source ou bien en utilisant le docker, se référer au README sur le dépôt GitHub. Vous trouverez également le processus d'extraction des caractéristiques d'AndroPyTool en Annexe F.

---

[8] Alejandro Martín García, A framework for automated extraction of static and dynamic features from Android applications, https://github.com/alexMyG/AndroPyTool, (Consultée le 7 juin 2019)

GroddDroid [102] propose une méthode pour améliorer l'exécution du code malveillant de logiciels malveillants inconnus en ciblant particulièrement les logiciels malveillants dotés de protections de déclenchement. Par exemple, une condition de déclenchement pourrait être l'attente d'une valeur spécifique pour une variable. Cependant, le but de GroddDroid est de forcer le déclenchement du code malveillant en combinant leurs deux contributions. Tout d'abord, la première contribution est la conception d'un algorithme identifiant automatiquement le potentiel code malveillant. La deuxième contribution est la reprise et l'amélioration de Monkey pour donner naissance à GroddDroid qui stimule l'interface graphique d'une application et force l'exécution de certaines conditions de déclenchement. Le code source de GroddDroid est publié en tant que logiciel libre[9].

2.4 Lacunes des méthodes présentées

Nous relevons certaines lacunes communes de l'étude précédente. Un nombre conséquent de propositions n'utilisent qu'une approche statique ou bien seulement une approche dynamique. Ceci n'est pas envisageable pour proposer un modèle de détection complet puisque comme nous l'avons vu dans le Tableau 2.2, les avantages et les inconvénients des analyses statiques et dynamiques se complètent.

Les approches qu'elles soient statiques, dynamiques ou hybrides peuvent présenter les lacunes suivantes :

- des caractéristiques peu ou aucunement diversifiées [72], [74], [81];
- une évaluation du modèle et/ou une base de données pauvre en termes de quantités d'échantillons [72];
- une évaluation du modèle et/ou une base de données contenant des applications anciennes [72], [103];
- des méthodes obsolètes dû aux nouvelles versions Android [75];

[9] Adrien Abraham, Pierre Graux, Jean-Francois Lalande, Mourad Leslous, Valérie Viet Triem Tong et Pierre Wilke (2015), GroddDroid, http://kharon.gforge.inria.fr/grodddroid.html (Consulté le 25 juin 2019)

- Aucun code en libre accès n'est disponible ne permettant pas de reprendre les travaux présentés afin de les poursuivre [74], [97].

Enfin, pour les analyses dynamiques seulement :

- une intervention manuelle peut être requise [83], [84] pour garantir la pleine exploration de l'application;
- l'application pourrait déterminer si l'environnement d'exécution est une émulation. Dans ce cas, le code malveillante ne se déclencherait pas ce qui empêcherait sa détection [85].

Les approches hybrides peuvent avoir les lacunes précédentes comme la solution qui semble la plus aboutie MADAM [97] qui n'a pas de code en libre accès mais possède aussi les lacunes suivantes :

- des performances moyennes;
- le terminal doit être nécessairement rooté.

Enfin, tous les résultats présentés précédemment n'ont de valeur que s'ils sont associés à :

1. un nombre important d'applications dans la base de données;
2. des applications malveillantes récentes.

En effet, si les modèles précédents présentent des taux de détections excellents, il est envisageable qu'un nombre d'applications insuffisants sur l'ensemble de test et/ou des anciennes applications malveillantes biaisent les résultats. C'est le cas de SAMADroid [99] promettant 98.97% de justesse par approche statique et 82.76% de justesse par approche dynamique mais qui ne possède que des applications malveillantes provenant de Drebin [63] datant de 2010 à 2012.

Au terme de notre revue de littérature nous avons remarqué qu'il est important pour les chercheurs de partager leurs travaux à la fois pour la reproductibilité et l'évaluation par leurs pairs mais aussi pour leur notoriété car les articles les plus cités sont toujours ceux qui fournissent leur code et permettent aux autres chercheurs de s'appuyer sur leur proposition voire de se baser sur leur solution.

Puisque les avantages et les inconvénients des analyses statiques et dynamiques se complètent, nous avons opté pour une approche hybride. En conséquence, nous avons choisi l'utilisation d'un outil pour extraire des caractéristiques hybrides. L'utilisation des réseaux de neurones est privilégiée puisqu'elle présente des avantages d'adaptation à des situations inédites qui ne peuvent être négligés contrairement aux systèmes de détection traditionnelles fonctionnant avec des règles de sécurité [75]. La base de données permettant l'entraînement du réseau de neurones sera également hybride. La base de données Omnidroid [70] répond à toutes ses exigences. De plus, le nombre d'applications de cette base de données est conséquent puisqu'il y a 22 000 exemplaires ce qui nous permet d'utiliser l'apprentissage automatique dont l'entraînement est fondamental. Le nombre de caractéristiques est lui aussi conséquent : 25,999 caractéristiques statiques et 5,932 caractéristiques dynamiques. Les échantillons d'Omnidroid datent de 2012 à 2018 permettant ainsi de pouvoir détecter d'anciens logiciels malveillants mais également les plus récents.

CHAPITRE 3 MODÈLE DE DÉTECTION PROPOSÉ

Ce chapitre présente le modèle de détection de logiciels malveillants proposé appelé « **BrainShield** » qui est divisé en trois modules. Le premier module est la base de données Omnidroid choisie pour entraîner les réseaux de neurones. Le deuxième module sont les techniques d'apprentissages que nous avons mis en place ainsi que les métriques permettant d'évaluer les réseaux de neurones. Le troisième module est l'architecture client/serveur du modèle proposé.

3.1 Module 1- Base de données Omnidroid

Omnidroid a été obtenue grâce à l'outil d'extractions de caractéristiques AndroPyTool [100].

3.1.1 Méthodes d'obtentions d'Omnidroid

Omnidroid [70] est une base de données datant de 2018 obtenue grâce à AndroPyTool [100] constituée de caractéristiques statiques et dynamiques extraites d'applications Android malveillantes et bénignes. Elle est, à notre connaissance, la base de données existante la plus diversifiée en termes de types de caractéristiques. Elle est aussi équilibrée en termes de nombre d'exemplaires, de dates d'échantillons et de caractéristiques.

Le Tableau 3.1 présente les différents filtres utilisés sur les applications initiales reçues par les chercheurs d'AndroZoo [68] et Koodous [67]. Il a été nécessaire, dans une **première étape, de** retirer les doublons afin de n'avoir que des éléments uniques. Ensuite, le **deuxième filtre** retire les applications dont le fichier .APK n'est pas valide. Il s'agit d'un outil intégré à AndroGuard [82] qui permet de sélectionner uniquement les applications sans défaut. Nous remarquons que la plupart des applications sont acceptées. Enfin, il s'agit de retirer les applications dont la version minimale est au moins égale à Android 4.1, soit **l'API 16**, afin que DroidBox [91] puisse la faire rouler. En d'autres termes, si une application est codée de telle sorte que l'API minimale acceptée soit supérieure à 16, DroidBox ne sera pas en mesure de lancer l'application, mais pourra toutefois l'installer, comme nous avons pu le remarquer. La **troisième étape** consiste à valider l'étiquetage de chaque application, c'est-à-dire à bien vérifier qu'une application provenant d'un ensemble d'applications malveillantes est bien malveillante. Pour ce faire, les auteurs ont fait appel au service **VirusTotal** [66]. VirusTotal est un agrégateur de résultats d'antivirus en ligne (Voir

Annexe L). Il est possible de l'utiliser avec une API publique, mais une API académique permet d'accéder à des taux de téléchargements plus élevés. En effet, l'API académique permet de faire 1 000 demandes par minute pour un maximum de 20 000 demandes par jour. Une demande est la soumission d'un APK ou d'un autre identificateur (hash, md5…), afin d'obtenir un fichier json contenant les résultats des antivirus. Le seuil Ɛ est égal à 1. En d'autres termes, sur la soixantaine d'antivirus disponibles, si un seul détecte l'application comme malveillante, alors elle sera considérée comme telle. Il s'agit du « Ground Truth », une base nous servant de vérité absolue. C'est un choix des auteurs A. Martín, R. Lara-Cabrera et D. Camacho [100] qui a ses avantages et ses inconvénients. Notons, par exemple, qu'il est possible qu'un antivirus soit le seul à se tromper.

Tableau 3.1 Filtres et processus appliqués pour construire le jeu de données Omnidroid

	Initial	Sans **APK**s avec nom de paquet identique	Sans **APK**s invalides (AndroGuard)	Sans **APK**s n'ayant pas l'analyse DroidBox	Seuil VirusTotal positives Ɛ = 1
Androzoo	18,785	18,620	18,490	17,223	**11,000**
Koodous	4,386	3,976	3,922	3,795	
Applications bénignes	13,619	12,998	12,844	11,973	**11,000**

3.1.2 Caractéristiques

3.1.2.1 Caractéristiques statiques d'Omnidroid

Omnidroid est constituée d'un total de 31,927 caractéristiques. Vous trouverez la répartition exacte dans le Tableau 3.2 pour les caractéristiques statiques d'Omnidroid et dans le Tableau 3.3 pour les caractéristiques dynamiques.

Tableau 3.2 Répartition des caractéristiques statiques d'Omnidroid

Répartition Initiale		
Caractéristiques	Nombre	Pourcentage
Receivers	6,415	24.7%
Activités	6,089	23.4%
Autorisations	5,501	21.2%
Services	4,365	16.8%
API calls	2,129	8.2%
FlowDroid	961	3.7%
Opcodes	224	0.9%
API Packages	212	0.8%
Commandes systèmes	103	0.4%
Total	**25,999**	**100%**

Comme on peut le constater dans le Tableau 3.2, Omnidroid est constituée de 25,999 caractéristiques, soit 25,999 colonnes. Il a 9 groupes de caractéristiques statiques différentes provenant de plusieurs sources.

Provenant des méta-datas [104]

- **Autorisations :** Les autorisations Android forment des barrières entre l'application et l'utilisation de certaines fonctionnalités du terminal et doivent être déclarées dans le manifeste. Elles sont obtenues en utilisant AndroGuard.

- **Receivers :** Ce composant d'application permet aux applications de recevoir des intentions diffusées par le système ou d'autres applications. Nous nous concentrons sur les récepteurs déclarés dans le manifeste Android. Une liste des intentions déclarées pour chaque récepteur est incluse.
- **Services :** Ce composant permet d'effectuer des opérations sans l'interaction de l'utilisateur ou expose des fonctionnalités à utiliser à d'autres applications. Cette fonction est obtenue en analysant le fichier XML du manifeste. Une liste des intentions déclarées pour chaque service est incluse.
- **Activités :** Les activités implémentent des interfaces qui permettent à l'utilisateur d'interagir avec l'application. Toutes les activités doivent être déclarées dans le AndroidManifest.xml, qui peut être analysé à l'aide d'Androguard. Une liste des intentions déclarées pour chaque activité est incluse.

Provenant des fichiers Smali [104]

- **API calls :** Ils permettent de modéliser le comportement attendu de l'échantillon. Certains appels d'API se trouvent généralement parmi les échantillons de logiciels malveillants. Ces appels API sont extraits en analysant les fichiers Smali et en recherchant les opcodes d'invocation. Smali / BakSmali est un assembleur / désassembleur pour le format dex utilisé par Dalvik qui est l''implémentation Java VM d'Android. Les noms "Smali" et "BakSmali" sont les équivalents islandais d'assembleur et désassembleur respectivement.
- **API package** : Les appels d'API peuvent être regroupés par classe dans laquelle ils sont définis ou par leur package.
- **Opcodes** : La collecte d'un compteur pour chaque opcode Dalvik qui apparaît dans les fichiers Smali peut aider à comprendre la complexité et le comportement de l'échantillon. De plus, les fichiers Smali sont analysés pour récupérer tous les opcodes déclarés.
- **Commandes système :** Les commandes système sont des indicateurs de certaines activités qui doivent être contrôlées, comme l'escalade de privilèges ou le processus de root du terminal. Ces commandes système peuvent être stockées sous forme de chaînes.

Provenant de FlowDroid [78]

- **FlowDroid :** Un rapport créé à l'aide de l'outil FlowDroid est également inclus dans cette section. Il s'agit d'un outil utile qui effectue une analyse de trace (taint analysis) sur le code d'application, afin de découvrir les connexions entre une source et un évier. Ces sources et puits, qui ont été précédemment définis par le Framework SuSi [105], permettent de modéliser les fuites de données subies pendant le cycle de vie de l'application. Par exemple, cela permet de découvrir les connexions où l'IMEI du terminal est envoyé à un tiers, à l'aide du réseau. FlowDroid a été exécuté en limitant la mémoire RAM à 10 Go et le temps d'exécution à 5 min.

3.1.2.2 Caractéristiques dynamiques d'Omnidroid

Le Tableau 3.3 illustre la répartition des caractéristiques dynamiques d'Omnidroid.

Tableau 3.3 Répartition des caractéristiques dynamiques d'Omnidroid

Répartition Initiale		
Caractéristiques	Nombre	Pourcentage
Opennet	1,483	25.0%
Sendnet	1,186	20.0%
Fdaccess	937	15.8%
Dataleaks	867	14.6%
Recvnet	837	14.1%
Cryptousage	488	8.2%
Dexclass	134	2.3%
Total	**5,932**	**100%**

Comme on peut le constater dans le Tableau 3.3, Omnidroid est constituée de 5,932 caractéristiques soit 5,932 colonnes. Il a 7 groupes de caractéristiques dynamiques différentes.

Pour chaque type de caractéristiques dynamiques présentées dans le Tableau 3.3, nous allons prendre un exemple dans la base de données de l'application pour servir d'illustration.
Il s'agit de l'application dont le hash est :

00A4E2E16E030DCB36B1059542DE909D0C80ACCE88F26EA710F5CE94970B307

Vous pouvez retrouver le rapport de cette application sur le service VirusTotal en bas de page[10].

Les caractéristiques dynamiques suivantes sont celles de DroidBox [91].

- **Cryptousage :** représente le facteur de différenciation le plus pertinent entre les applications malveillantes et les applications bénignes. En effet, le nombre d'opérations cryptographiques effectuées par l'ensemble de logiciels malveillants est en moyenne de 120% supérieur aux applications bénignes d'après [100].

```
"cryptousage": {
    "2.8922500610351562": {
        "algorithm": "AES",
        "key": "-35, -71, -93, -104, -72, 117, -8, -3, -69, 86, 85, -96, 34, 125, -19, -94",
        "operation": "keyalgo",
        "type": "crypto"},
```

- **Dataleaks :** représente le nombre de fuites d'information et est en moyenne de 115% supérieur pour les applications malveillantes.

```
"dataleaks": {
    "10.750847101211548": {
        "data": "504f5354202f736572766963652f6d6f62696c652e68746d3f726571756574446174613
d2537422532326f7325323225334125323234 2e312e31253232532432532326d6f64656c25323225334125323
246756c6c2b416e64726f69642b6f6e2b456d",
        "desthost": "120.55.239.119",
        "destport": "80",
        "fd": "23",
        "operation": "send",
        "sink": "Network",
        "tag": [
            "TAINT_IMEI",
            "TAINT_IMSI"],"type": "net write"}
```

[10] VirusTotal (2020), Rapport d'une application, https://www.virustotal.com/gui/file/000a4e2e16e030dcb36b1059542de909d0c80acce88f26ea710f5ce94970b307/detection, (Consulté en Avril 2020)

- **Dexclass :** peut être utilisé pour exécuter du code non installé dans le cadre d'une application.

```
"dexclass": {
    "1.2047600746154785": {
        "path": "/data/app/com.leixun.taofen8_boyangjiafang-1.apk",
        "type": "dexload"
            },
    "2.7910921573638916": {
        "path": "/data/app/com.leixun.taofen8_boyangjiafang-1.apk",
        "type": "dexload"}}
```

- **Fdaccess :** représente le nombre d'accès en écriture et en lecture d'une application.

```
"fdaccess": {
    "0.10704803466796875": {
            "data": "efbfbd060000efbfbdefbfbd0c00050000000000000000",
            "id": "824861684",
            "operation": "write",
            "path": "/dev/input/event0",
            "type": "file write"},
```

- **Opennet :** désigne les opérations d'ouverture de socket. L'adresse IPv4 et le port de destination sont précisés.

```
"opennet": {
    "10.368289947509766": {
            "desthost": "120.55.239.119",
            "destport": "80",
            "fd": "23"},
```

- **Recvnet :** désigne les opérations entrantes via le réseau. L'adresse IPv4, le port de destination, le type souvent « net read » et les données sont précisés.

```
    "recvnet": {
    "11.086910963058472": {
            "data":
"485454502f312e3120323030204f4b0d0a446174653a2053756e2c203133204d617920323031382032323a303
03a333420474d540d0a436f6e74656e742d547970653a206170706c69636174696f6e2f6a736f6e3b636861727
365743d5554462d380d0a",
            "host": "120.55.239.119",
            "port": "80",
            "type": "net read"},
```

- **Sendnet :** représente les opérations sortantes via le réseau. L'adresse IPv4, le port de destination, le type et les données sont précisés. Si possible, un tag taint est ajouté.

```
44  "sendnet": {
45     "10.750847101211548": {
46             "data":
    "504f5354202f736572766963652f6d6f62696c652e68746d3f726571756573744461746 13d2537422532326f7
    3253232253341253232342e312e31253232253243253232 6d6f64656c25322253341253232 6756c6c2b416e6
    4726f69642b6f6e2b456d",
47             "desthost": "120.55.239.119",
48             "destport": "80",
49             "fd": "23",
50             "operation": "send",
51             "sink": "Network",
52             "tag": [
53                    "TAINT_IMEI",
54                    "TAINT_IMSI"],
55             "type": "net write"}
```

3.1.2.3 Ingénierie des caractéristiques

L'ingénierie des caractéristiques (Feature engineering) [106] consiste à étudier les données pour mieux les comprendre. Nous pouvons par exemple :

- **Visualiser** : tracer des histogrammes, classer les valeurs de la plus fréquente à la moins fréquente;
- **Déboguer** : identifier des exemples en doublon, des valeurs manquantes, des valeurs aberrantes, des données d'apprentissage et de validations similaires… Ceci a été fait en partie par les créateurs de la base de données (cf. §3.1.1). En effet, leur processus de filtres a permis d'enlever les applications (nombre d'exemplaires) en doublon, ainsi que les applications non compatibles avec DroidBox.

Une des bonnes pratiques concernant la qualité d'une bonne caractéristique est de vérifier sa fréquence. Une caractéristique doit apparaître à peu près au moins cinq fois dans un ensemble de données pour être considéré comme bonne d'après [107]. En effet, de cette manière, un modèle peut apprendre comment cette valeur de caractéristique est liée à l'étiquette.

La sélection des caractéristiques est un processus par lequel nous sélectionnons les caractéristiques des données afin d'améliorer les métriques d'évaluation du modèle. La sélection des caractéristiques présente les trois avantages suivants [108] :

1. **Réduit le temps d'entraînement** : moins de données signifie moins de calculs.
2. **Améliore la précision** : moins de données trompeuses signifie une précision accrue.

3. **Réduit le sur-apprentissage** : moins de données redondantes signifie moins de possibilités de prendre des décisions basées sur le bruit.

3.2 Module 2- Réseaux de neurones

Les réseaux de neurones sont entraînés sur la base de données Omnidroid avec les techniques d'apprentissages supervisées et sont en charge de la détection des applications malveillantes.

Notre module 2, le réseau de neurones, est composé de trois sous-modules :

1. du type des réseaux de neurones;
2. des techniques d'apprentissage pour réaliser un apprentissage juste et précis, tels que le découpage de la base de données en trois groupes et la régularisation;
3. des métriques d'évaluation, tels que la justesse, la précision et le rappel.

3.2.1 Type des réseaux de neurones

La méthode de détection proposée est basée sur les techniques d'intelligence artificielle. Il s'agit plus particulièrement des méthodes utilisant les réseaux de neurones entièrement connectés ou « dense layer » [109]. Dans ce type de couches de neurones, chaque neurone est connecté à chaque neurone de la couche précédente et chaque connexion a son propre poids. Il s'agit d'un modèle de connexion qui ne fait aucune hypothèse sur les caractéristiques de la base de données. **C'est pourquoi nous avons opté pour un réseau de neurones entièrement connecté.**

En effet, la couche convolutionnelle [109] est un autre type de couche de neurones dans laquelle chaque neurone n'est connecté qu'à quelques neurones proches (les locaux) de la couche précédente et le même ensemble de poids est utilisé pour chaque neurone ainsi que la configuration de connexion locale. Ce modèle de connexion n'a de sens que dans les cas où les données peuvent être interprétées comme étant spatiales, les entités à extraire étant spatialement locales sont susceptibles de se produire également à n'importe quelle position d'entrée. Le cas d'utilisation typique pour les couches convolutionnelles est le cas où l'on travaille avec des images. Les caractéristiques sont locales et susceptibles de se produire n'importe où, ce qui n'est pas notre cas.

Cependant, le nombre réduit de connexions et de poids rend la couche convolutive relativement plus avantageuse par rapport à la couche entièrement connectée, en termes de mémoire et de puissance de calcul nécessaires. N'étant pas dans une problématique de performances lors de l'entraînement du réseau de neurones, nous écartons ce désavantage.

3.2.2 Techniques d'apprentissage

Nous nous situons dans un problème d'apprentissage supervisé de classification. Les étiquettes y sont dans notre cas :

$$\begin{cases} application\ malveillante, & y = 1 \\ application\ bénigne, & y = 0 \end{cases}$$

3.2.2.1 Découpage de la base de données en trois groupes

Un modèle d'apprentissage consiste à faire de bonnes prédictions sur des données inédites. Mais, si nous construisons un modèle à partir de l'ensemble des données, comment obtiendrons-nous des données inédites ? Une façon consiste à diviser l'ensemble de données en deux sous-ensembles : un **ensemble d'entraînement** et un **ensemble de validation [**110**]**. Ainsi, nous pouvons entraîner le modèle sur l'ensemble d'entraînement puis valider notre entraînement sur l'ensemble de validation afin de rectifier les paramètres lors de l'apprentissage. Cela va conduire à un risque de sur-apprentissage. Le sur-apprentissage est un phénomène qui arrive lorsque le modèle cherche à trop coller aux données d'entraînement. Il devient trop spécifique et donc incapable de prédire dans une situation différente. De ce fait, nous définissons un autre **ensemble** de **test** [111] qui nous permet d'éviter le sur-apprentissage. Il s'agit d'un ensemble utilisé en toute fin d'apprentissage et une seule fois contrairement aux deux autres, afin de vérifier que le modèle peut s'adapter à une situation inédite. Nous avons donc choisi de répartir notre base de données conformément à la manière décrite à la Figure 3.1.

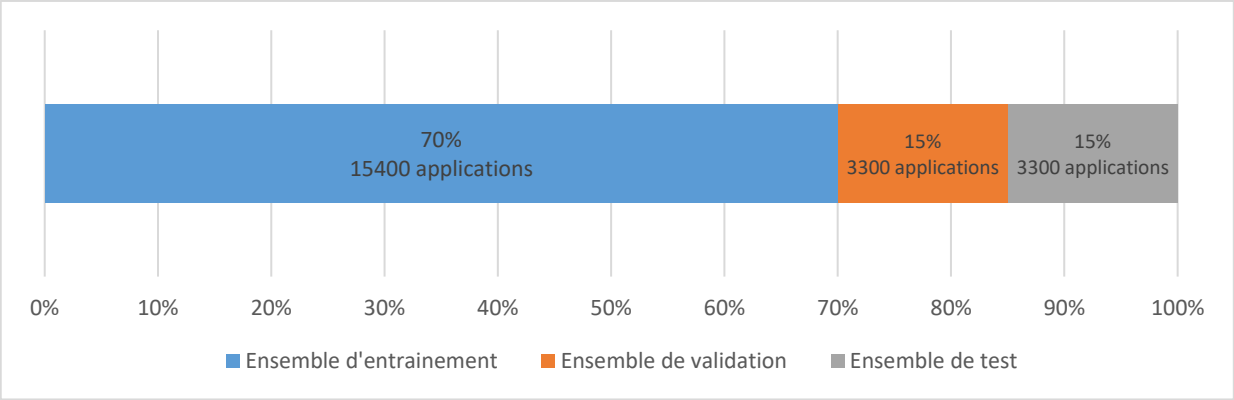


Figure 3.1 Partitionnement d'un ensemble de données en trois sous-ensembles

Nous avons choisi 70% (15,400) pour l'ensemble d'entraînement, 15% (3,300) pour l'ensemble de validation et enfin 15% (3,300) pour l'ensemble de test.

La Figure 3.2 montre comment nous avons entraîné les réseaux de neurones. Nous chargeons tout d'abord la base de données puis la mélangeons de manière aléatoire pour avoir des sets différents d'applications malveillantes et bénignes entre chaque entraînement différent. Après avoir choisi les caractéristiques, réglé quelques paramètres, créé notre réseau de neurones et découpé la base de données en trois groupes, l'entraînement du réseau de neurones peut avoir lieu. Le nombre d'itération peut être variable puis nous terminons par évaluer le réseau de neurones sur l'ensemble de test.

Algorithm 1: Algorithme d'entraînement

```
Chargement en mémoire de la base de données Omnidroid;
Mélange aléatoire des applications;
Sélection des caractéristiques voulus;
Déclaration des hyperparamètres;
Création du réseau de neurones;
Découpage de la base de données en trois groupes;
for 200 epochs do
    entraînement sur l'ensemble d'entraînement;
    validation sur l'ensemble de validation;
end
Évaluation sur l'ensemble de test;
```

Figure 3.2 Pseudo Code d'entraînement du réseau de neurones

3.2.2.2 La fonction de perte

La fonction de perte (loss function) [112] vise à pénaliser le modèle pour une mauvaise prédiction. Autrement dit, la perte est un nombre indiquant à quel point la prédiction du modèle était mauvaise sur un seul exemple. C'est la fonction que l'on cherche à minimiser lors de l'apprentissage. Il existe plusieurs fonctions de perte pour différents cas.

Une fonction de perte communément utilisée est celle qui se réfère à l'erreur quadratique moyenne (MSE) qui est la perte quadratique moyenne par exemple sur l'ensemble de données.

La perte d'**entropie croisée**, ou perte logarithmique, est une autre fonction de perte qui mesure les performances d'un modèle de classification dont la sortie est une valeur de probabilité comprise entre 0 et 1. La perte d'entropie croisée augmente à mesure que la probabilité prédite diverge de l'étiquette réelle. C'est cette fonction de perte que nous avons choisie. La nôtre sera binaire comme dit précédemment avec :

$$\boldsymbol{Entropie\ croisée\ binaire} = -(ylog(p) + (1-y)\log(1-p))^{11} \quad (3.1)$$

$$\text{Où} \quad \begin{cases} application\ malveillante, y = 1 \\ application\ bénigne, y = 0 \\ p, prédiction \end{cases}$$

3.2.2.3 Régularisation

La régularisation [113] est une technique permettant d'éviter le sur-apprentissage via des pénalités correspondant à une complexité.

Au lieu de seulement chercher à minimiser la fonction de perte :

$$minimiser(Perte(Données|Modèle))$$

On va :

$$minimiser(Perte(Données|Modèle)) + complexité(Modèle)$$

---

[11] Machine Learning Cheatsheet (2017), Documentation, https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy, (Consulté en Juin 2019)

Le sur-apprentissage fait référence à un modèle qui modélise trop bien les données d'entraînement. Ce dernier se produit lorsqu'un modèle apprend les détails et le bruit dans les données d'apprentissage, dans la mesure où cela a un impact négatif sur les métriques d'évaluation du modèle sur l'ensemble de test. De ce fait, le bruit ou les fluctuations aléatoires des données d'apprentissage sont captés et appris en tant que concepts par le modèle.

La **régularisation L1** [113] **(**régression Lasso) pénalise les pondérations proportionnellement à la somme de leurs valeurs absolues. Elle aide à mettre à zéro les pondérations des caractéristiques peu ou non pertinentes, ce qui a pour effet de supprimer celles-ci du modèle.

La **régularisation L2** [113] (régression Ridge) pénalise les pondérations proportionnellement à la somme de leurs carrés. Elle aide la pondération des anomalies à rapprocher de zéro.

Une autre méthode de régularisation est **l'arrêt prématuré [**114**] (**early stopping) qui implique de terminer l'entraînement du modèle avant que la perte de l'apprentissage ne remonte.

Le **dropout** ou « **l'abandon** » [109] est une forme de régularisation. Elle consiste à ce que le réseau de neurones abandonne de manière aléatoire ce qu'il a appris. Le dropout varie entre 0 (aucun abandon) et 1 (abandon total). En conséquence, une plage de valeurs acceptables communément admise est comprise entre 0.1 et 0.4 ce qui correspond à 10% et 40% d'abandon. En d'autres termes, plus la valeur d'abandon est forte plus la régularisation est forte.

Nous avons choisi d'adopter le dropout comme technique de régularisation ainsi que **l'arrêt prématuré**. En effet, ces techniques sont largement répandues et acceptées par l'ensemble de la communauté scientifique. Elles ont fait leurs preuves et nous étudierons l'impact du dropout sur les métriques d'évaluation.

3.2.3 Métriques d'évaluation

Les métriques d'évaluation **[**11**]** sont des mesures quantifiables nous permettant de savoir si le modèle de détection permet effectivement de différencier les applications malveillantes des applications bénignes. Nous présentons ci-après les plus pertinentes : la justesse, la précision, le rappel. Elles sont calculées à partir de quatre autres métriques : le nombre de vrais positifs, de vrais négatifs, de faux positifs et de faux négatifs décrites.

La matrice de confusion **[**11**]** est un tableau de dimension N*N qui résume le succès des prédictions d'un modèle de classification; c'est-à-dire la corrélation entre l'étiquette et la classification du modèle. Un axe d'une matrice de confusion est l'étiquette prédite par le modèle et l'autre axe est l'étiquette réelle. N représente le nombre de classes. Dans notre cas, N est égal à 2, ce qui donne le Tableau 3.4:

Tableau 3.4 Matrice de confusion N= 2

Vrai positif (VP) L'application malveillante est prédite malveillante	**Faux positif (FP)** L'application bénigne est prédite malveillante
Faux négatif (FN) L'application malveillante est prédite bénigne	**Vrai négatif (VN)** L'application bénigne est prédite bénigne

La matrice de confusion permet de calculer les métriques suivantes : justesse, précision, rappel et F1 score.

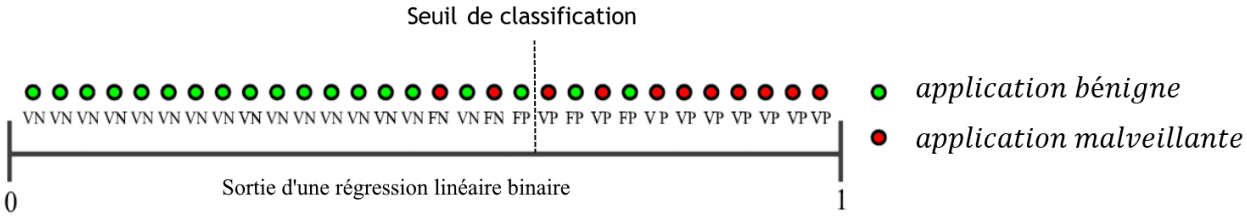


Figure 3.3 Classement binaire des applications comme bénigne ou malveillante

La **justesse** (accuracy) **[**7**]** est la proportion de prédictions correctes d'un modèle de classification, soit :

$$\textbf{Justesse} = \frac{Nombre\ de\ prédictions\ correctes}{Nombre\ total\ de\ prédictions} \tag{3.2}$$

Pour une classification binaire :

$$\textbf{Justesse} = \frac{VP+VN}{VP+VN+FP+FN} \tag{3.3}$$

Où VP = Vrais positifs, VN = Vrais négatifs, FP = Faux positifs, et FN = Faux négatifs.

La **précision** (precision) **[**7**]** permet de répondre à la question suivante :

Quelle proportion d'identifications positives était effectivement correcte ? Pour une classification binaire, nous avons :

$$\textbf{Precision} = \frac{VP}{VP+FP} \tag{3.4}$$

Le **rappel** (recall) **[**7**]** permet de répondre à la question suivante :

Quelle proportion de résultats positifs réels est identifiée correctement ? Pour une classification binaire, cela correspond à :

$$\textbf{Recall} = \frac{VP}{VP+FN} \tag{3.5}$$

Le rappel est aussi appelé le taux de vrai positif (TPR).

Le **score F1** [12] est la moyenne harmonique de la précision et du rappel. Par conséquent, ce score prend en compte à la fois les faux positifs et les faux négatifs.

$$\textbf{F1 score} = 2 * \frac{Precision * Rappel}{Precision+Rappel} \tag{3.6}$$

L'AUC [12**]** [115] est aussi une métrique d'évaluation. Elle signifie « Aire sous la courbe ROC ». La courbe ROC montre les résultats d'un modèle de classification à tous les seuils de classification. Autrement dit, l'AUC mesure toute la zone bidimensionnelle sous toute la courbe ROC. L'AUC fournit donc une mesure globale des résultats à travers tous les seuils de classification possibles. Une façon d'interpréter l'AUC est la probabilité que le modèle classe un exemple positif aléatoire plus fortement qu'un exemple négatif aléatoire.

Les métriques d'évaluation précédentes, souvent appelées « résultats », sont à absolument distinguer de la **performance** qui est utilisée pour décrire la rapidité d'entraînement, l'espace de stockage et la mémoire vive utilisé de l'objet dont il est question. Nous pouvons distinguer la différence de performance entre la phase d'entraînement de l'algorithme plus lente et la phase de prédiction d'un nouvel échantillon plus rapide.

3.2.4 Les hyperparamètres

Pour réaliser un bon apprentissage sur un jeu de données, il est nécessaire de régler certains paramètres. En effet, les paramètres suivants ont un impact significatif sur les résultats.

- **Nombre d'itérations** : le nombre d'itérations est le nombre de cycles d'apprentissage du réseau de neurones. Un cycle correspond à un entraînement sur l'ensemble d'entraînement que l'on vient améliorer en le validant sur l'ensemble de validation.
- **Taux d'abandon** : le taux d'abandon correspond au taux de connexions que l'on oublie de manière aléatoire dans le réseau de neurones. Cela revient à mettre la valeur zéro aux poids.
- **Taux d'apprentissage** : le taux d'apprentissage correspond à la vitesse à laquelle apprend le réseau de neurones. Un taux d'apprentissage trop faible risque de ne pas converger. Il en va de même pour un taux d'apprentissage trop grand.
- **Nombre de neurones** : le nombre de neurones correspond au nombre de neurones d'entrée sur la première couche L1.
- **Fonction d'activation** : la fonction d'activation est la fonction d'activation de la couche de neurones L1.

Nous étudierons l'impact de ces hyperparamètres lors de leur réglage.

3.3 Module 3-Client/Serveur

Le client Android permet d'envoyer les applications analysées et d'afficher les résultats. Le serveur Python effectue l'extraction des caractéristiques et la prédiction des résultats à partir des vecteurs de caractéristiques qu'il envoie au client.

Notre module 3 est divisé en deux sous-modules :

1. **le client** Android sur lequel les applications doivent être analysées.
2. **le serveur** Python sur lequel se fait la détection d'applications malveillantes.

Nous avons opté pour une architecture client-serveur afin de décharger les calculs sur le serveur et de pouvoir faire de l'analyse dynamique. Il est possible de faire une analyse dynamique sur le terminal mobile puisque, par définition, pour pouvoir la faire, il est nécessaire de faire rouler une

application. Cependant, il est absurde de faire rouler une application sur les terminaux vulnérables à protéger, sans avoir préalablement effectué une analyse de détection de logiciels malveillants. De plus, les outils d'analyses présents dans la littérature sont tous disponibles sur ordinateur seulement. Ils permettent d'effectuer des calculs aussi bien plus importants, en termes de capacités de calcul. En d'autres termes, si nous partons sur une solution sur le terminal mobile, nous nous cantonnons à une méthode statique basée sur les autorisations et/ou les appels API.

En effet, les deux seules références qui proposent une extraction des caractéristiques statiques sur le terminal mobile ont des inconvénients qui ne peuvent pas être ignorés dans le cadre de ces travaux. L'un possède des résultats aux alentours de 72% de détection [74], tandis que l'autre propose une méthode de détection statique basée sur les autorisations systèmes seulement [72] et s'évaluant sur des applications malveillantes de 2010 à 2012 qui sont trop anciennes.

La Figure 3.4 propose l'architecture du modèle proposé. L'activité « Main Activity.kt » envoie au serveur la liste des applications au format .apk que l'on souhaite scanner. Une fois le téléchargement terminé, l'outil d'extraction des caractéristiques statiques et dynamiques AndroPyTool est utilisé pour obtenir les caractéristiques dans des fichiers au format .json. Ensuite, des scripts pythons sont exécutés pour formater les caractéristiques choisies afin que le réseau de neurones ait en entrée un vecteur de dimension égal au nombre de caractéristiques. Ce réseau de neurones au format .h5 entraîné avec Omnidroid fait les prédictions. Les résultats des prédictions sont des nombres « float » compris entre 0 (bénin) et 1 (malveillant) qui sont formatés au format .json pour pouvoir envoyer les résultats de prédiction à l'activité cliente « ResultActivity.kt ».

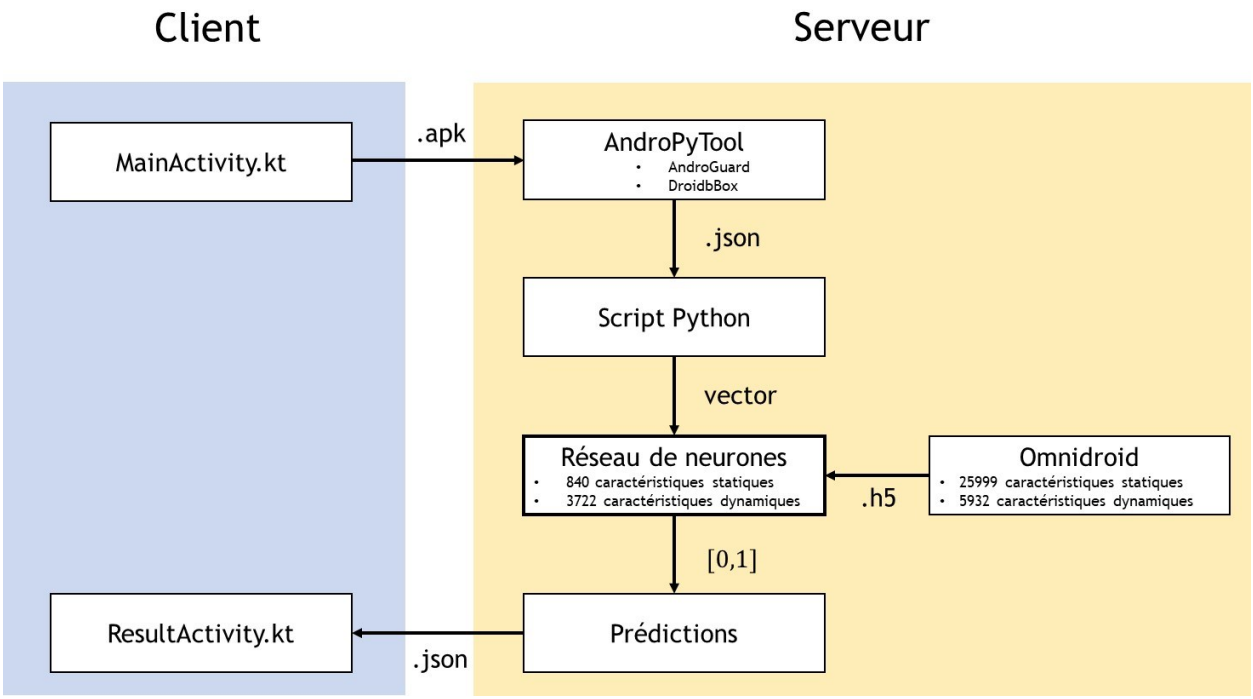


Figure 3.4 Architecture du modèle proposé

À chaque application correspond un vecteur de caractéristiques auquel correspond une prédiction.

L'architecture est commune à la méthode statique et à la méthode dynamique. Les différences entre ces deux méthodes résident dans les caractéristiques statiques et dynamiques disponibles dans Omnidroid et dans l'outil d'AndroPyTool utilisé qui extraira les caractéristiques correspondantes. En effet, AndroGuard [82] est utilisé pour les caractéristiques statiques, tandis que DroidBox [91] est utilisé pour extraire les caractéristiques dynamiques.

CHAPITRE 4 IMPLÉMENTATION ET RÉSULTATS

Ce chapitre présente l'ensemble des résultats pour la détection de logiciels malveillants en termes de métriques d'évaluation, telles que la justesse et la précision. Il présente également le prototype d'implémentation des meilleurs réseaux de neurones entraînés : BrainShield. Tout d'abord, nous exposons les outils utilisés et l'environnement logiciel à des fins de reproductibilité et de transparence. Ensuite, nous présentons le prototype BrainShield de ces réseaux de neurones ayant une architecture client/serveur. Puis, nous montrons comment nous avons implémenté les réseaux de neurones. Nous présentons aussi la méthode pour régler les hyperparamètres des entraînements des réseaux de neurones. Puis, nous exposerons l'étude menée pour sélectionner les caractéristiques pertinentes. Par la suite, nous montrons les effets d'un ré-étiquetage pour finir sur les résultats finaux des réseaux de neurones statiques, dynamiques et hybrides.

4.1 Environnement et matériel

Dans cette section, nous allons définir l'environnement logiciel et le matériel utilisé qui serviront à obtenir les résultats escomptés.

4.1.1 Environnement logiciel

Tensorflow[12] : Tensorflow est une bibliothèque en libre accès pour le calcul numérique et l'apprentissage automatique à grande échelle créée par l'équipe de Google Brain. Tensorflow regroupe une multitude de modèles et d'algorithmes d'apprentissage automatique.

Keras[13] : Keras est une API de réseaux neuronaux de haut niveau. Écrite en Python et capable de s'exécuter sur Tensorflow. Keras est développée dans le but de permettre une implémentation rapide et efficace. En effet, créer un réseau de neurones et son entraînement peut se contenir en une dizaine de lignes de code seulement.

[12] https://www.tensorflow.org/
[13] https://keras.io/

Python[14] **:** le langage de programmation le plus populaire est Python et c'est aussi le cas pour les chercheurs qui affectionnent particulièrement Jupyter Notebook. En effet, de nombreuses librairies sont disponibles pour ce langage de programmation et surtout des librairies scientifiques.

Kotlin[15] **:** le langage de programmation officiel annoncé par Google Inc. pour la programmation Android est Kotlin depuis le **8 mai 2019**. C'est le langage de programmation choisi pour développer le prototype « BrainShield ». Avant Kotlin, le langage de programmation officiel était Java. Ces deux langages de programmation sont compatibles.

Flask[16] **:** Flask est une librairie Python permettant de créer un serveur.

CUDA/CuDNN[17] **:** CUDA et CuDNN sont des outils mis à la disposition de NVIDIA permettant de faire basculer l'entraînement du réseau de neurones du CPU vers le GPU afin de réduire les temps de calcul.

Le Tableau 4.1 montre la liste et les versions des langages, libraires et outils utilisées dans cette recherche pour des fins de reproductibilité.

Tableau 4.1 Liste des outils utilisés pour l'entraînement et l'implémentation

Outil	Version
TensorFlow	2.1.0
Keras	2.2.4-tf
Kotlin	1.3.70
Flask	1.1
Python	3.7.6
DESIGNED FOR NVIDIA CUDA cuDNN	10.2 et 7.6

[14] https://www.python.org/
[15] https://kotlinlang.org/
[16] https://flask.palletsprojects.com/
[17] https://developer.nvidia.com/

C'est la librairie Tensorflow qui est choisie pour l'implémentation du réseau de neurones. Une autre alternative pour construire le réseau de neurones aurait été d'utiliser PyTorch qui est une bibliothèque en libre accès similaire mais créée par l'équipe de Facebook. Étant donné qu'une application Android est envisagée, Tensorflow est privilégié à des fins de compatibilité puisqu'il s'agit des mêmes équipes de développement de Google Inc. qui sont à l'initiative de Kotlin et Tensorflow. Keras est choisi pour sa facilité d'utilisation et sa compatibilité avec Tensorflow. De plus, étant largement adoptée par l'ensemble des développeurs, nous pouvons trouver beaucoup d'aides en ligne. Ayant à réaliser de nombr eux entraînements, nous basculons les calculs du CPU vers le GPU à l'aide de CUDA et CuDNN [116] [117] afin de pouvoir réduire les temps d'entraînement. Enfin, nous allons choisir le langage de programmation Kotlin pour pouvoir permettre à l'entreprise partenaire de pouvoir maintenir le code à jour sur le long terme. En effet, Kotlin et Java sont compatibles mais étant donné que Kotlin est un langage très récent, il est probable que celui-ci soit adopté à long terme palliant les lacunes des plus anciens langages.

4.1.2 Matériel

Dans le Tableau 4.2, nous illustrons la liste du matériel utilisé pour réaliser les entraînements.

Tableau 4.2 Matériel physique utilisé pour l'entraînement

Ordinateur Windows 10	Processeur	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 2201 Mhz, 6 Cores, 12 Logical Processors
	Mémoire physique (RAM)	**16 GB**
	GPU	NVIDIA **GeForce GTX 1060** Max-Q
Téléphone Huawei P20 Lite (ANE-LX3)	Processeur	Hisilicon **Kirin 659** @ 2.36GHz, 2360 MHz, 8 Cores
	Mémoire physique (RAM)	**4 GB**
	GPU	ARM **Mali T830 MP2**
	Android	**9.1**

4.2 Prototype d'implémentation

Les réseaux de neurones sont intégrés au sein d'une application Android « BrainShield » développée par nous-même. Cette application (le client) communique avec un serveur, lui aussi entièrement développé, afin de réaliser une solution complète et fonctionnelle. Nous informons le lecteur que ce travail est réalisé et testé au sein de l'entreprise Flex Group Inc. dans le cadre du projet ATISCOM.

Le fonctionnement de l'application est la suivante. L'application « BrainShield » permet de sélectionner les applications installées du terminal mobile pour les envoyer sur le serveur afin d'y faire l'extraction des caractéristiques statiques et/ou dynamiques, puis d'effectuer la prédiction. Le serveur renvoie l'ensemble des prédictions au format json. L'application cliente traite le json afin d'afficher un logo vert pour une application bénigne ou bien un logo rouge pour une application malveillante.

4.2.1 Architecture du prototype BrainShield

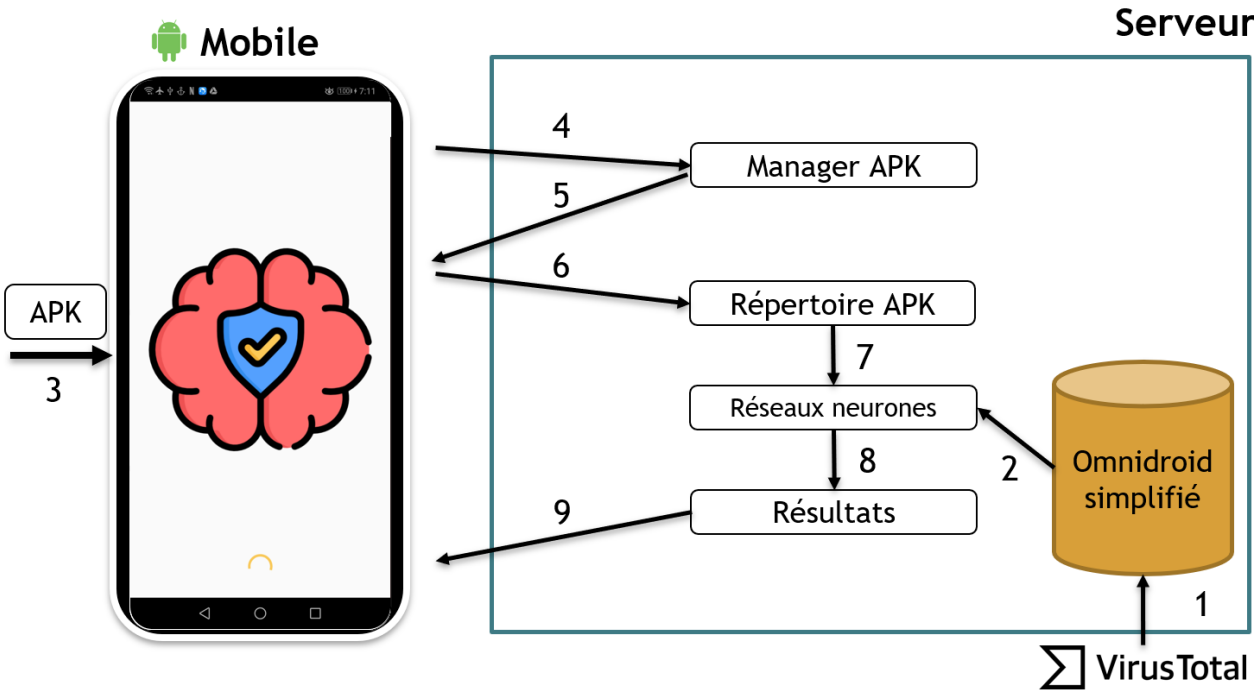


Figure 4.1 Architecture du prototype BrainShield

1. **Étiquetage** : VirusTotal a permis d'attribuer à chaque application sa classification initiale bénigne ou malveillante. Cette étape n'est réalisée qu'une seule fois.
2. **Entraînement** : le réseau de neurones entièrement connecté est entraîné grâce aux techniques d'apprentissage (cf. chap 3, §3.2.2). Les hyperparamètres ont été préalablement réglés (cf. §4.4). Cette étape n'est réalisée qu'une seule fois.
3. **Acquisition APKs** : pour pouvoir faire la prédiction d'applications inconnues, il est nécessaire de se procurer des applications. Nous avons installé, pour le développement, une cinquantaine d'applications provenant du Google Play Store.
4. **Envoi de la demande d'analyse**: le client envoie la liste des applications qu'il souhaite analyser.
5. **Renvoi des APKS absents** : le serveur renvoie au client la liste des applications (les APKs) qu'il ne possède pas.

6. **Envoi des APKs**: les applications qui ne sont pas présentes sur le serveur sont envoyées par requête POST et enregistrer.
7. **Extraction des caractéristiques** : le serveur extrait les caractéristiques des applications.
8. **Prédiction** : le réseau de neurones choisi (statique ou dynamique) entraîné fournit la prédiction de chaque application en fonction du vecteur de caractéristiques d'entrée.
9. **Envoi des prédictions** : le serveur envoie les prédictions au client.

4.2.2 Application du côté du serveur

Le serveur est une application Python développée avec **Flask** qui roule sur un **serveur Amazon**. Il s'agit d'un serveur Ubuntu Server 18.04 LTS (HVM), SSD Volume, type t2.xlarge situé dans la zone us-east-2a avec 4 vCPUs, Architecture x86_64, 16384 MiB de mémoire RAM et 16 Go de mémoire SSD. Le prix est de 0.1856 USD par heure. Le code du serveur est disponible en Annexe J. Le serveur est composé de deux dossiers. L'un se nomme « AndroPyTool » (en rose) et contient tous les scripts et codes nécessaires à l'extraction des caractéristiques statiques et dynamiques. Il s'agit du code en libre accès sur le GitHub. L'autre dossier « Serveur Deploiement » contient les scripts Python pour faire rouler l'application Android, ainsi que la base de données des APK déjà analysées et leurs fichiers json respectifs contenant les caractéristiques extraites.

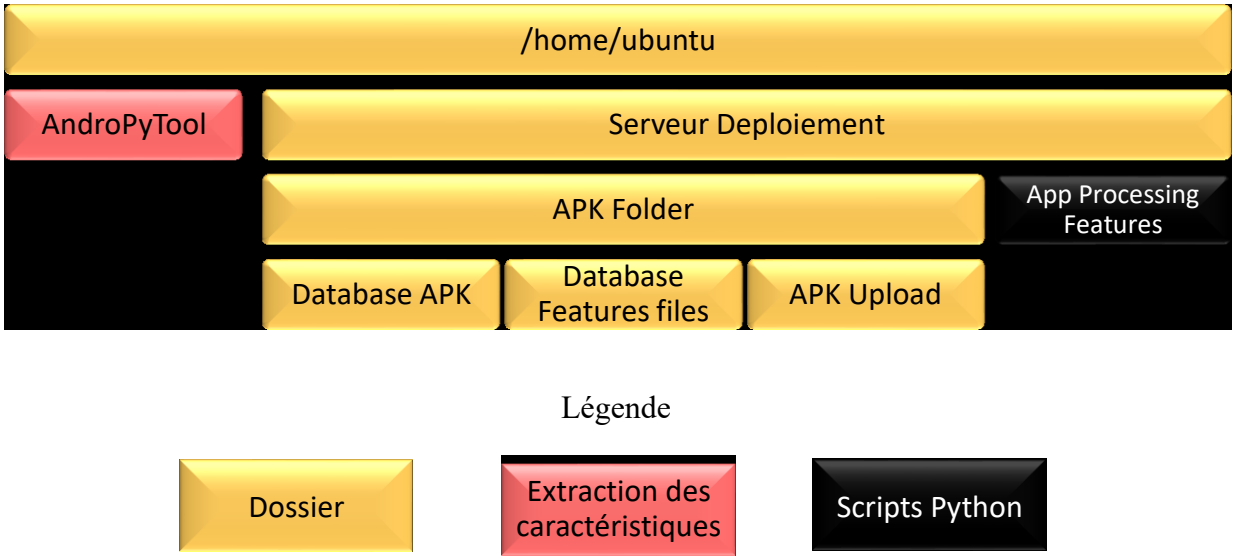


Figure 4.2 Architecture du serveur

Premièrement, le serveur attend que le client communique par requête HTTP puis quand le client envoie la liste des applications Android qu'il souhaite analyser, le serveur renvoie la liste des applications qu'il n'a pas déjà dans sa base de données et copie ce qu'il a déjà dans le dossier de travail « Upload_APK ». Le client envoie finalement les APK qui manquent au serveur. Enfin le serveur, place tous les APK qu'il doit analyser dans le dossier de travail. Il lance la commande « python /home/ubuntu/AndroPyTool/androPyTool.py » en précisant le dossier de travail. Cette commande permet de lancer l'extraction des caractéristiques. Enfin, le modèle de neurones est chargé via la commande « keras.models.load_model('/home/ubuntu/ServeurDep/StaticModel.h5') » avant de faire la prédiction pour chaque application. Il renvoie la réponse au format json que le client traitera.

4.2.3 Application du côté du client

Le client est l'application Android écrite en langage de programmation Kotlin. C'est cette application qui roule sur le téléphone Huawei P20 Lite (ANE-LX3). Le code est disponible en Annexe I. Pour rappel, l'extraction des caractéristiques et la prédiction se déroulent sur le serveur.

Au lancement de l'application, c'est l'activité « SplashScreenActivity.kt » qui se lance. Elle affiche le logo et une icône de chargement pendant 0.3 secondes, puis c'est l'activité principale « MainActivity.kt » qui prend le relais. Nous avons alors un RecyclerView affichant la liste des applications installées. Pour pouvoir afficher ces dernières, l'activité principale fait appel aux classes « AppInfo.kt » et« AppManager.kt ». Il y a aussi un menu en haut à droite de l'écran à l'intérieur de la barre d'outils. Ce menu présente l'option pour sélectionner une application présente dans les fichiers de l'utilisateur.

Tableau 4.3 Liste des classes de l'application Android

Liste des classes Kotlin	Description
SplashScreenActivity.kt	Écran de démarrage avec logo
AppInfo.kt	Contient les informations des applications (Nom, nom de paquet, icone, sélectionné ou non…)
AppManager.kt	Gestion des informations des applications (Tri par ordre alphabétique, application système ou non…)
MainActivity.kt	Activité principale listant les applications installées et les boutons pour interagir
MainActivityAdapter.kt	Chaque activité suivante cochée possède un adapter. Un adapter permet de gérer le visuel dynamique du Recycler View
ResultStaticActivity.kt	Affiche les applications analysées et les prédictions statiques
ResultDynamicActivity.kt	Affiche les applications analysées et les prédictions dynamiques
ResultStorageActivity.kt	Affiche l'application analysée et la prédiction
VirusTotalActivity.kt	Affiche les résultats selon VirusTotal. Correspond au bouton ONLINE sur l'activité principale

Interface client

Les captures d'écrans suivantes permettront de visualiser le fonctionnement de l'application prototype.

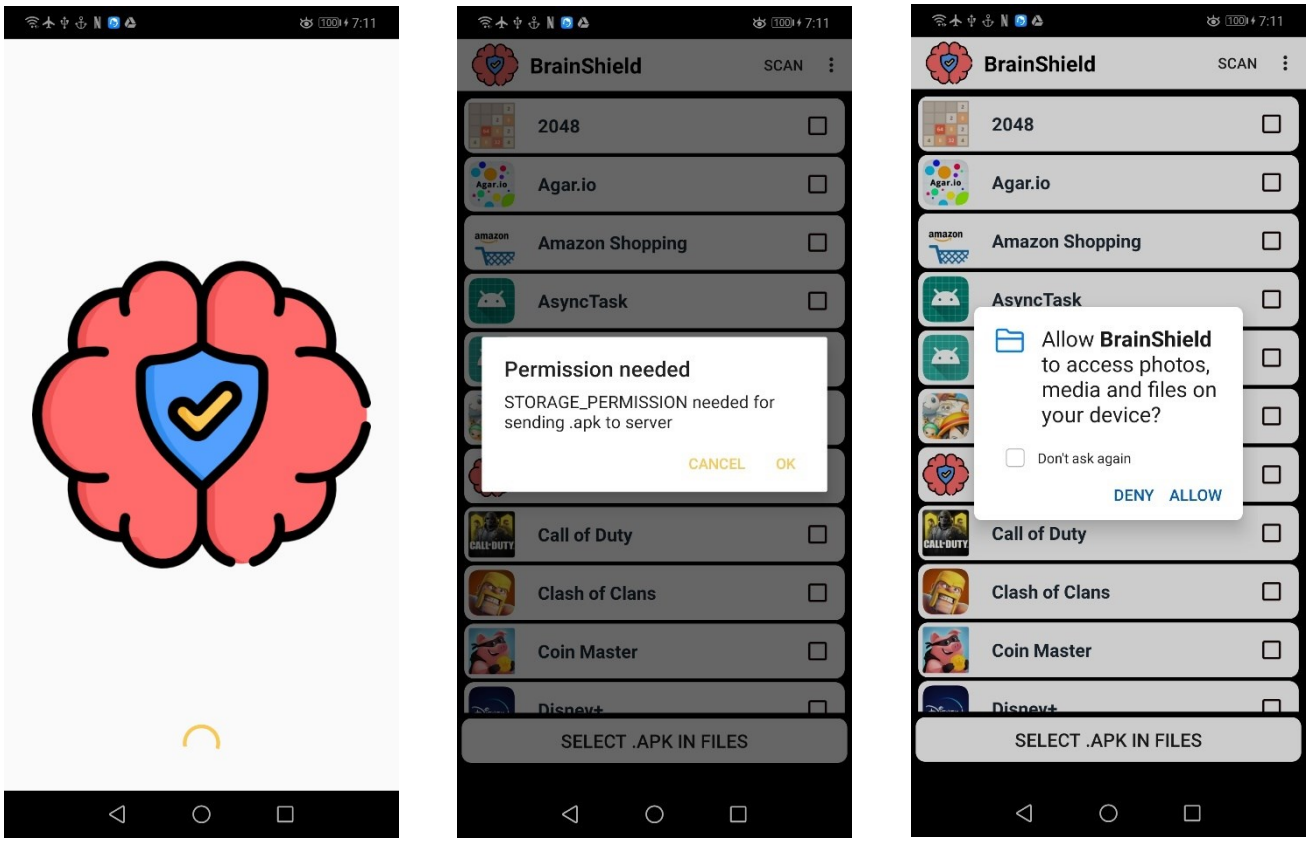


Figure 4.3 Demande d'autorisation de l'application

Sur la Figure 4.3, nous observons tout d'abord, sur la gauche, une capture d'écran représentant le chargement. Le logo représente un cerveau ainsi qu'un bouclier en son centre. Le **cerveau** représente l'utilisation des techniques d'intelligence artificielle, tandis que le **bouclier** représente la sécurité et la protection. C'est pourquoi nous avons décidé de nommer l'application **BrainShield**. Ensuite, les deux prochaines captures d'écran montrent que l'application a besoin d'avoir accès aux médias présents sur le téléphone. Il n'y a pas besoin de cette autorisation pour afficher les applications installées avec BrainShield, mais elle est nécessaire pour envoyer les applications APK sur le serveur.

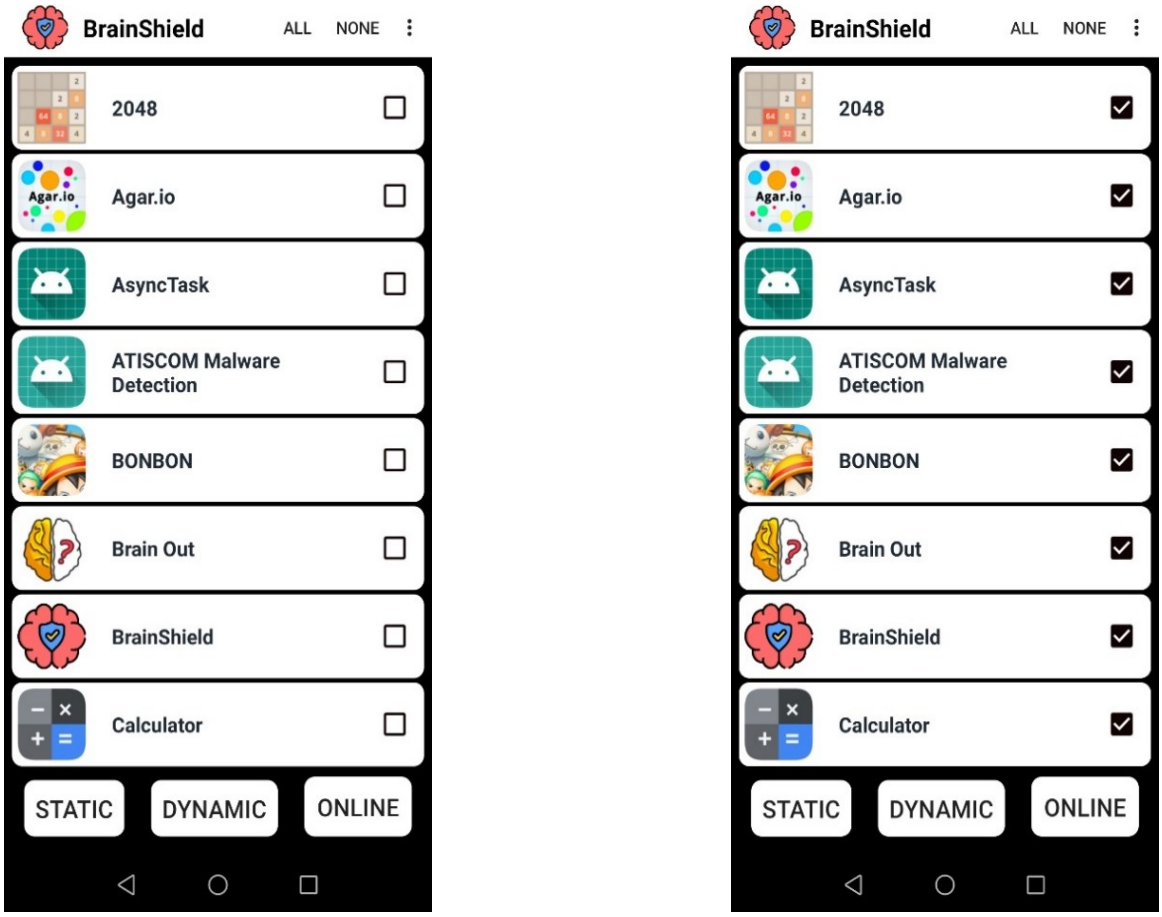


Figure 4.4 Liste des applications installées et sélection

Nous pouvons observons sur la Figure 4.4 qu'une fois l'autorisation accordée par l'utilisateur ou non, l'application affiche la liste des applications installées sur le terminal mobile dans un RecyclerView. Nous laissons le soin à l'utilisateur de choisir la méthode de détection qu'il souhaite grâce aux trois boutons disponibles en bas de l'écran. Il est nécessaire tout d'abord de sélectionner les applications désirées en cliquant dessus. Nous observons les applications sélectionnées sur l'image de droite. En haut à droite, dans le menu, les options « ALL » et « NONE » permettent de cocher ou décocher toutes les applications en un clic.

Nous avons implémenté trois méthodes :

- La **méthode statique** est accessible via le bouton « STATIC » et permet d'envoyer sur les serveurs les applications souhaitées pour faire l'extraction des caractéristiques statiques. L'extraction des caractéristiques est effectuée avec AndroPyTool basé sur AndroGuard [82]. Pour chaque application, un vecteur de caractéristiques statiques est créé. Nous avons donc autant de vecteurs que nous avons d'applications. Chaque vecteur a une dimension égale au nombre de caractéristiques. Pour effectuer la prédiction, le réseau de neurones statique est chargé en mémoire, puis effectue la prédiction pour chaque vecteur. Le serveur renvoie alors au client l'ensemble des prédictions au format json. Une deuxième fonctionnalité est de permettre à l'utilisateur d'envoyer un APK qui n'a pas été installé via le menu en haut à droite.
- La **méthode dynamique** est accessible via le bouton « DYNAMIC ». Il s'agit exactement des mêmes étapes que la méthode statique. En revanche, l'extraction des caractéristiques est effectuée avec DroidBox [91]. Le réseau de neurones est quant à lui entraîné avec les caractéristiques dynamiques.
- La **méthode VirusTotal** est accessible via le bouton « ONLINE ». Pour cette méthode, aucun modèle n'est entraîné puisque nous faisons appel au service en ligne VirusTotal via une clé académique qui nous a été donnée par un administrateur de VirusTotal. Pour détecter si une application est malveillante ou non, nous nous basons sur le nombre d'antivirus détectant cette application comme telle. Le seuil ε est fixé à 1, c'est-à-dire que si un seul antivirus parmi la soixantaine disponible détecte l'application soumise comme malveillante, alors le logo rouge symbolisant le danger sera affiché.

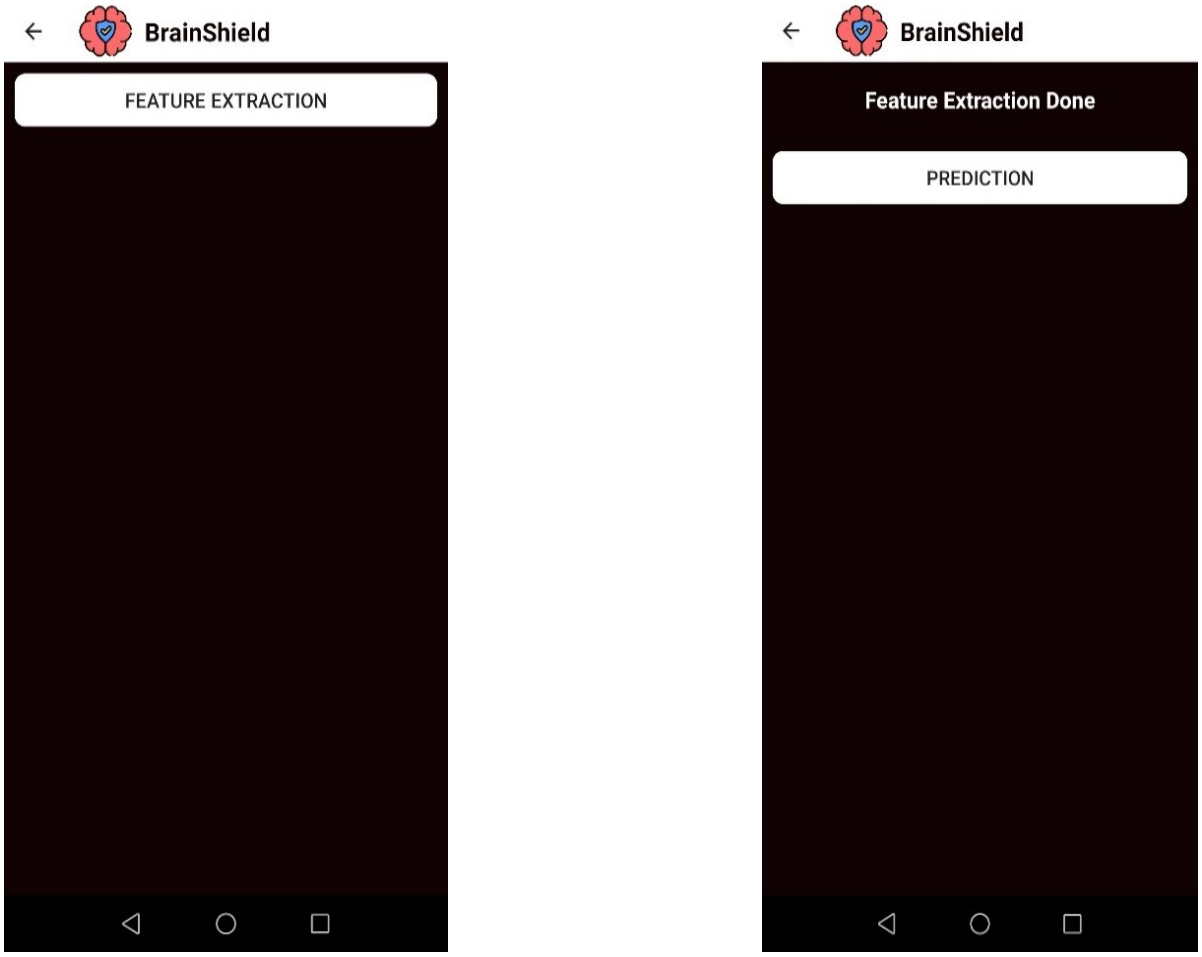


Figure 4.5 Extraction des caractéristiques et prédiction

Sur la Figure 4.5, seuls les boutons permettant de lancer l'extraction des caractéristiques et la prédiction sur le serveur sont visibles à l'utilisateur.

Les boutons « Feature Extraction » et « Prediction » ont le même visuel pour les analyses statique et dynamique mais lancent des commandes correspondantes à la méthode choisie par l'utilisateur. C'est-à-dire que le bouton « Feature Extraction » statique lance l'extraction des caractéristiques statiques. Le bouton « Feature Extraction » dynamique lance l'extraction des caractéristiques dynamiques.

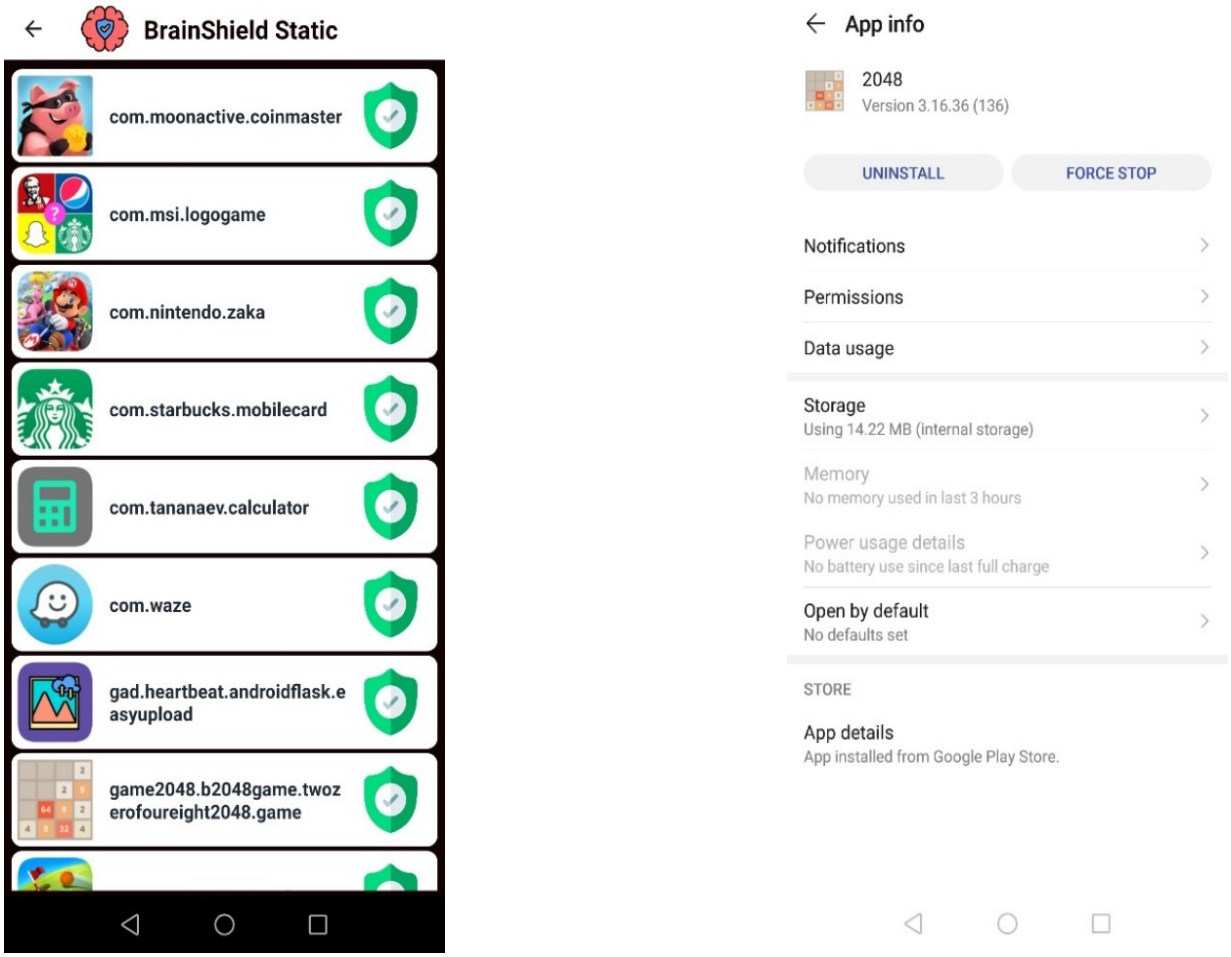


Figure 4.6 Résultats de la méthode statique et clic sur une application

Sur la Figure 4.6, nous pouvons voir les résultats statiques sur la gauche. Pour chaque application, un logo est disposé : un logo vert pour une application bénigne, un logo rouge pour une application malveillante et un logo orange pour une application à risque (prédiction comprise entre 0.4 et 0.6). Si l'utilisateur désire supprimer une application, il peut le voir à partir de cette interface, en cliquant sur un endroit compris dans le rectangle blanc de l'application correspondante. Nous voyons alors les paramètres de l'application sélectionnée sur l'image de droite. Il s'agit de l'écran des paramètres de l'application propre au terminal mobile qui est lancée via une intention.

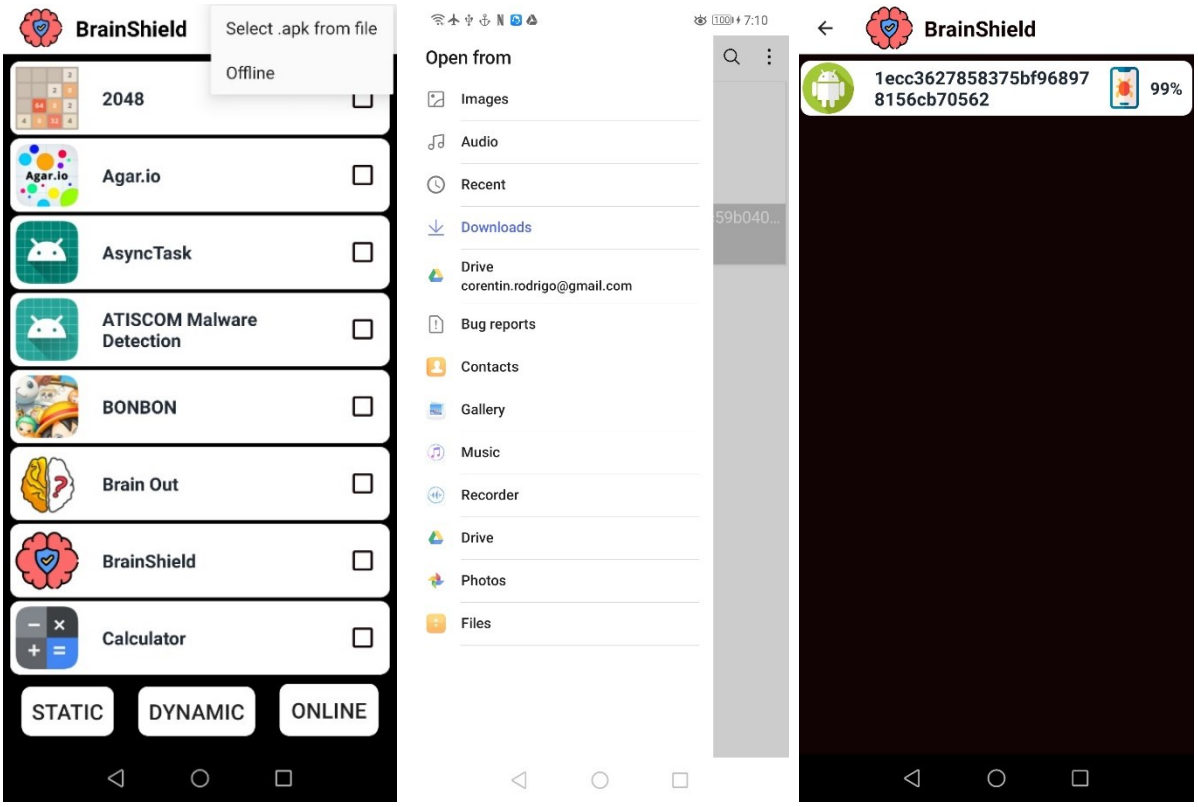


Figure 4.7 Cas pour une application non installée

La Figure 4.7 permet de montrer l'utilisation de la fonctionnalité du menu « Select .apk from file ». Après avoir cliqué sur ce bouton du menu, nous pouvons naviguer dans les médias du terminal mobile via une intention afin de sélectionner un fichier .apk qui n'ait pas forcément installé. Cela fait, nous pouvons reprendre strictement les mêmes étapes que la méthode statique via les boutons pour extraire les caractéristiques et effectuer la prédiction. Enfin, sur l'image de droite, on observe le résultat statique de l'application sélectionnée.

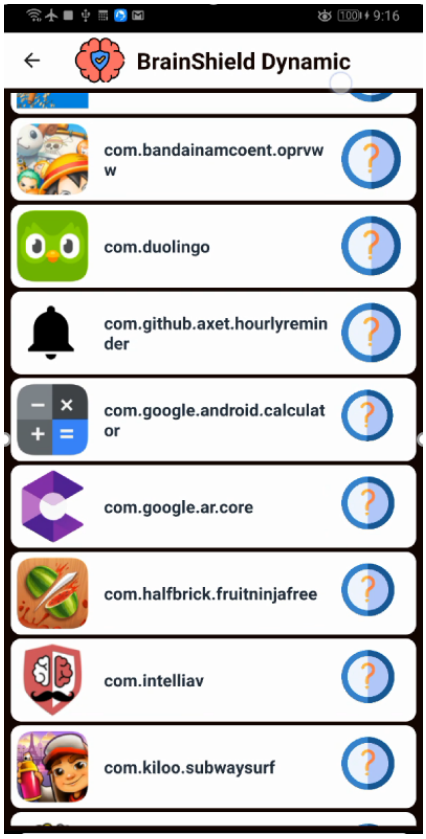


Figure 4.8 Résultats de la méthode dynamique

La Figure 4.8 montre les résultats de l'approche dynamique. Comme pour l'approche statique nous affichons les logos verts, oranges et rouges pour les prédictions correspondantes. Nous avons aussi ajouté un nouveau logo qui est représenté à l'écran : un point d'interrogation. Nous avons lancé l'outil d'extraction des caractéristiques pendant cinq minutes mais on peut s'apercevoir que les caractéristiques extraites ne sont pas en nombre suffisants pour effectuer une prédiction. Nous rediscuterons des causes probables de ce phénomène dans les limitations et travaux futurs.

4.3 Implémentation des réseaux de neurones

Le code Python suffisant permettant d'entraîner un réseau de neurones avec **Tensorflow et Keras** est présenté à la Figure 4.9 suivante. Le code complet pour entraîner les réseaux de neurones proposés est disponible en Annexe H. Il est nécessaire de charger au préalable la base de données sous le nom « omnidroid_dataframe ». Nous commençons tout d'abord par créer le modèle de la ligne 2 à 5. Ensuite, nous définissons les ensembles d'entraînement et de test en découpant la base de données Omnidroid de la ligne 8 à 11. Par la suite, nous définissions la fonction de perte et l'optimiseur à la ligne 14, puis nous lançons l'entraînement à la ligne 15 en précisant la répartition de l'ensemble d'entraînement et l'ensemble de validation. Le taux choisi de 0.1765 permet d'obtenir un entraînement sur 15400 applications et une validation sur 3300 applications. Ce qui laisse 3300 applications pour l'ensemble de test. Nous respectons ainsi les pourcentages suivants :

- 70% : 15,400 applications pour l'ensemble d'entraînement
- 15% : 3,300 applications pour l'ensemble de validation
- 15% : 3,300 applications pour l'ensemble de test

```
1.  // Création du réseau de neurones
2.  Number_of_features = len(preprocess_features(omnidroid_dataframe))-1
3.  model = Sequential()
4.  model.add(Dense(Number_of_features,input_dim = Number_of_features))
5.  model.add(Dense(1))
6.
7.  // Définition des ensembles d'entraînements et de tests
8.  X_train = omnidroid_dataframe.head(18700).values[:,1:]
9.  Y_train = omnidroid_dataframe.head(18700).values[:,0]
10. X_test =   omnidroid_dataframe.tail(3300).values[:,1:]
11. Y_test =   omnidroid_dataframe.tail(3300).values[:,0]
12.
13. // Entraînement du modèle. C'est ici que nous définissons la répartition entraînement/validation
14. model.compile(optimizer = keras.optimizers.Adamax(learning_rate=0.002), loss='binary_crossentropy')
15. model.fit(X_train,Y_Train,validation_split = 0.1765)
16.
17. // Évaluation du modèle
18. model.evaluate(X_test,Y_test)
```

Figure 4.9 Code Python pour entraîner un réseau de neurones avec Tensorflow/Keras

4.4 Réglage des hyperparamètres

Cette section expose le réglage des hyperparamètres d'entraînements des réseaux de neurones qui détectent les logiciels malveillants.

4.4.1 Conditions d'entraînement

Pour trouver les valeurs des hyperparamètres, nous allons mener divers entraînements sur la base de données statiques de 3,359 caractéristiques. Pour rappel, les hyperparamètres à régler sont les suivants : le nombre d'itérations, le taux d'abandon, le taux d'apprentissage, le nombre de neurones et la fonction d'activation.

Le Tableau 4.4 contient les valeurs initiales pour les entraînements. Parmi ce réglage initial, le taux d'apprentissage, le taux d'abandon et la fonction d'activation sont par défaut. Le nombre d'itérations de 50 et le nombre de neurones sur la couche L1 de 1,119 sont choisis de manière suffisamment grande pour avoir des résultats fidèles et pour mener à bien des centaines d'entraînements dans un temps convenable.

Tableau 4.4 Valeurs initiales pour le réglage des hyperparamètres statiques et dynamiques

Nombre d'itérations	50	
Taux d'apprentissage	0.002	
Taux d'abandon	0.3	
Nombres de neurones sur L1	Statiques : 1,119	Dynamiques : 1,240
Fonction d'activation	Relu	

4.4.2 Nombre d'itérations

Afin de choisir le nombre d'itérations permettant d'obtenir les meilleurs résultats, nous faisons varier le nombre d'itérations et comparons les différents entraînements avec les métriques d'évaluation préalablement présentés (cf. chap 3, §3.2.3).

Les résultats sont obtenus en faisant une moyenne statistique sur dix entraînements pour chaque valeur de la variable étudiée.

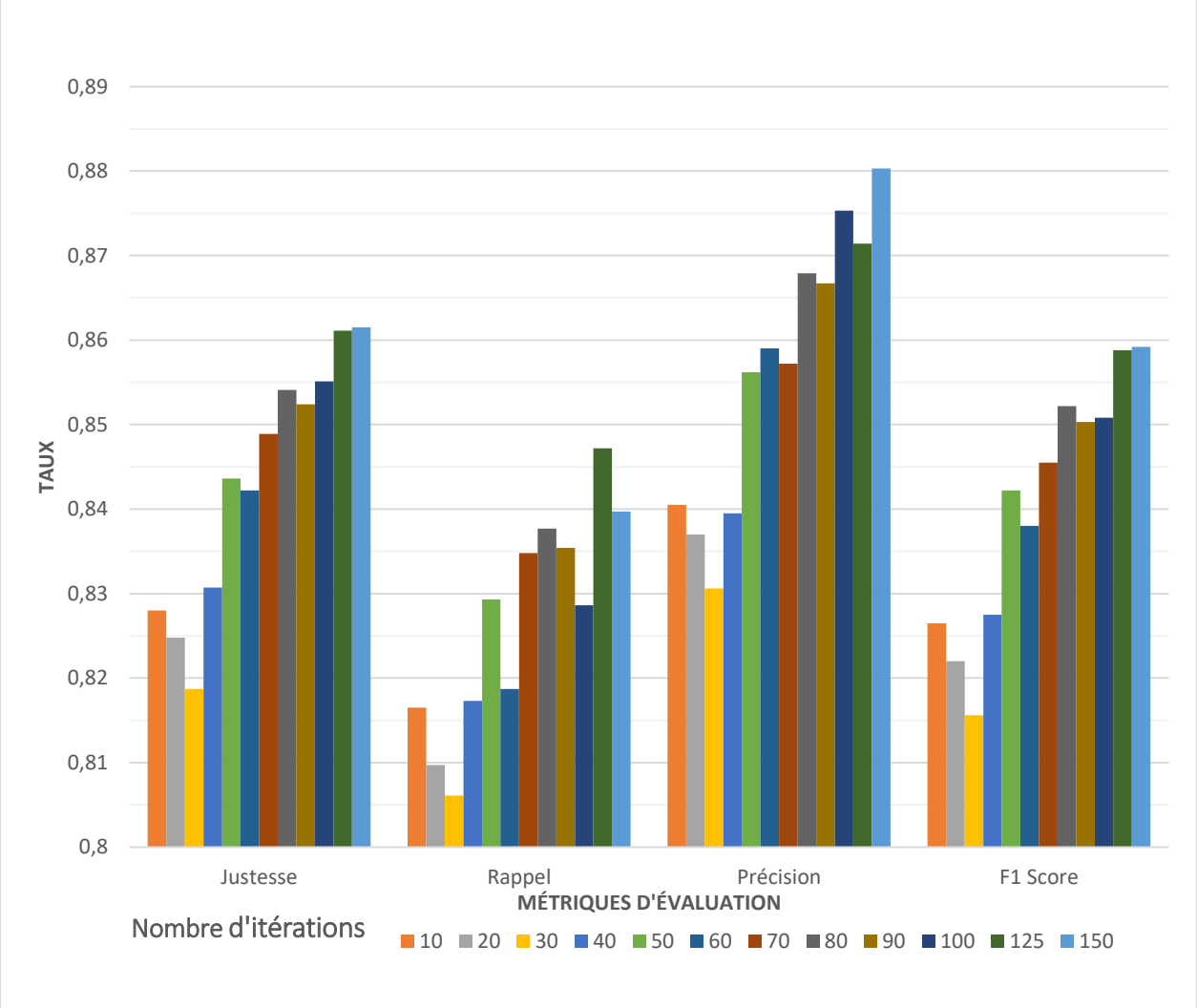


Figure 4.10 Impact du nombre d'itérations sur les métriques d'évaluation statique

Nous pouvons observer à la Figure 4.10 que, de manière générale, plus le nombre d'itérations d'apprentissage est grand, plus les métriques d'évaluation montrent de meilleurs scores. Nous

notons cependant la baisse de résultats pour le nombre d'itérations 20 et 30 comparé aux réseaux de 10 itérations. Le réseau de neurones présente un comportement imprévisible quand le nombre d'itérations est faible, comme c'est le cas entre 10 et 30. Les résultats ne cessent d'augmenter à partir de 40 itérations et le maximum choisi de 150 itérations ne suffit pas à trouver la valeur maximale souhaitée. Nous choisirons donc un nombre d'itérations plus élevé prochainement (120 entraînements d'une durée totale de 5,610 secondes soit 93 minutes ont été réalisés).

Nous remarquons à la Figure 4.11 que le temps d'entraînement est proportionnel au nombre d'itérations.

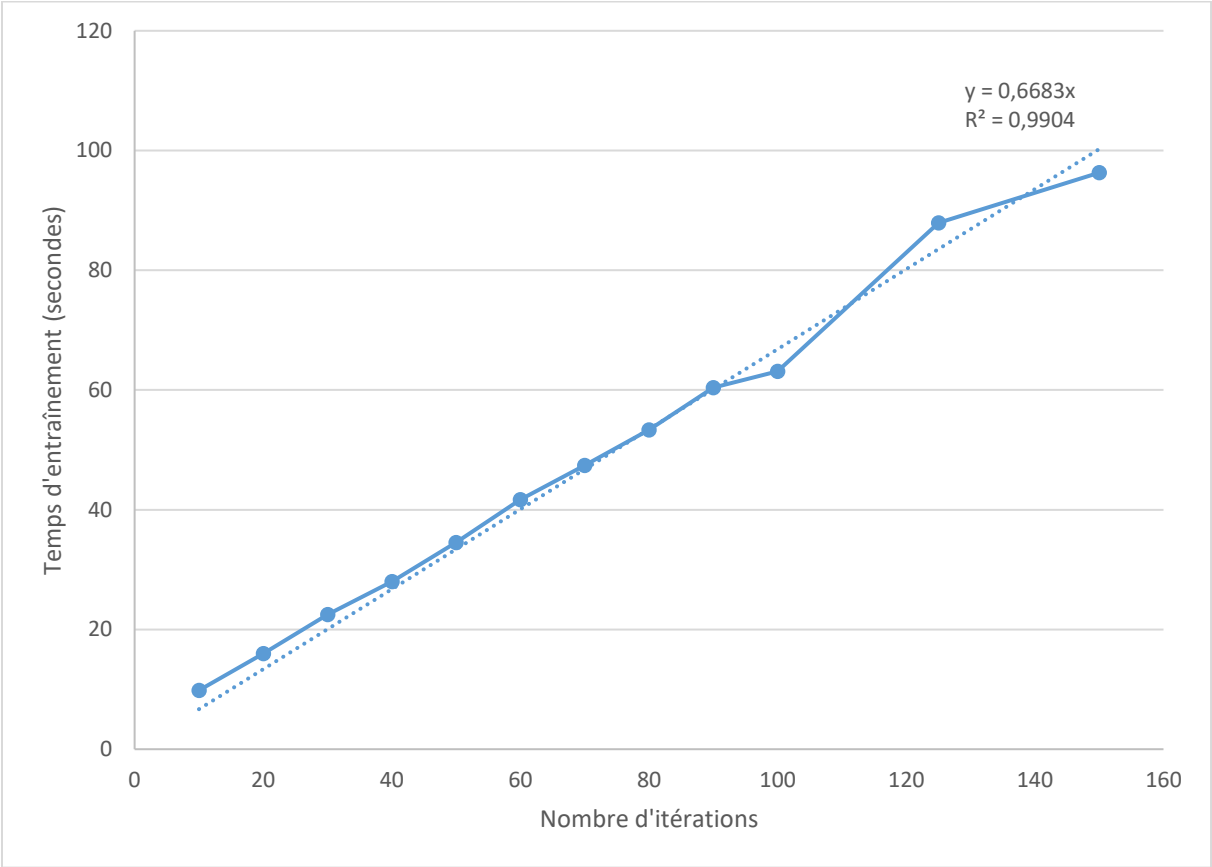


Figure 4.11 Relation entre le temps d'entraînement et le nombre d'itérations

Nous allons explorer de manière plus générale le nombre d'itérations sur la base de données dynamique afin de s'assurer que ceux-ci conviennent également sur la base de données dynamique.

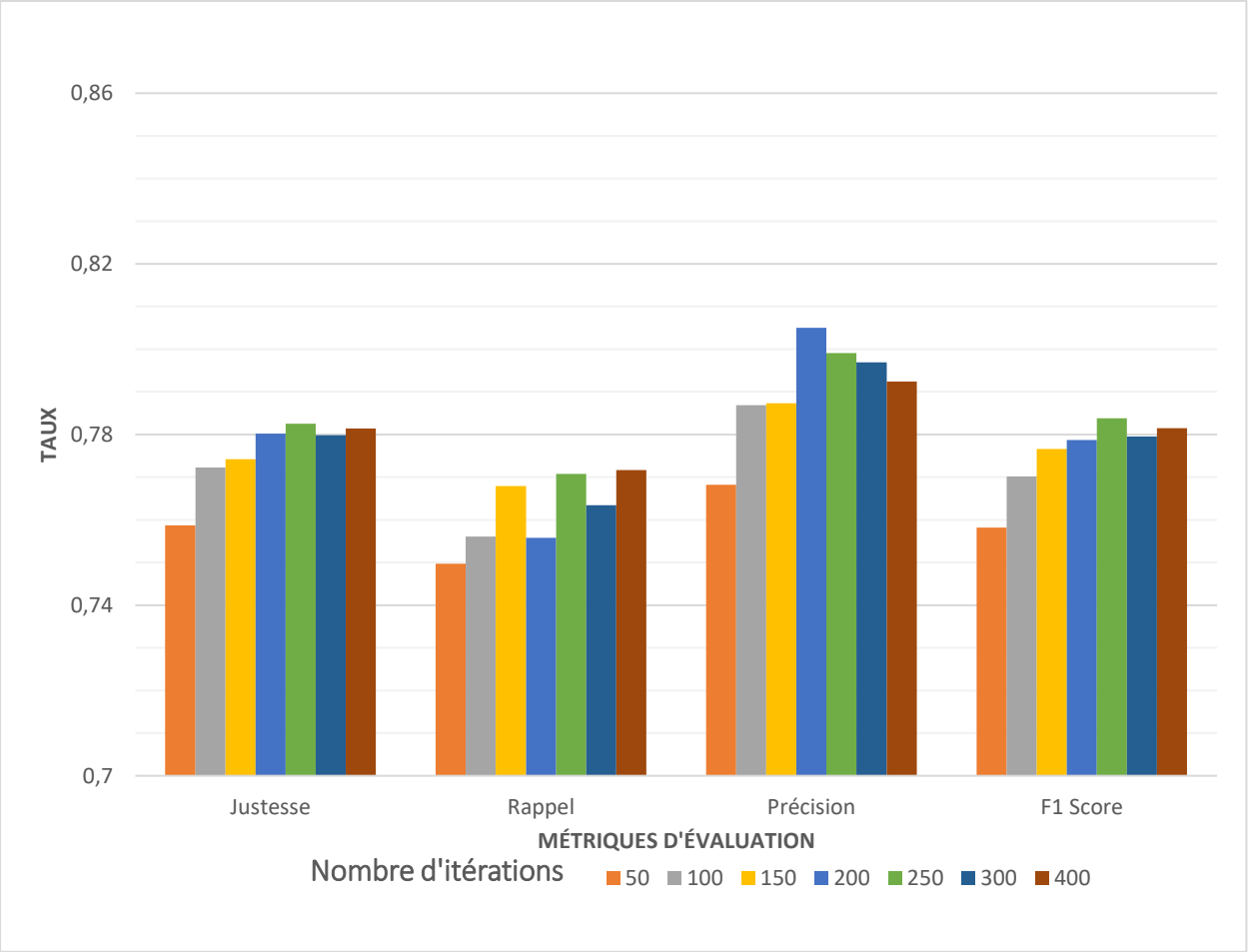


Figure 4.12 Impact du nombre d'itérations sur les métriques d'évaluation dynamique

Nous observons à la Figure 4.12 une augmentation des résultats progressifs jusqu'à 250 itérations. Par la suite, l'augmentation du nombre d'itérations ne suffit plus à augmenter les résultats, pire nous observons une légère baisse. Pour les résultats finaux dynamiques, nous choisissons de garder un nombre d'itérations égal à 250 (70 entraînements d'une durée totale de 9,437 secondes soit 157 minutes ont été réalisés)

4.4.3 Taux d'abandon

De même, afin de choisir le taux d'abandon permettant d'obtenir les meilleurs scores, nous faisons varier ce dernier de 0 (aucun abandon) à 0.9 (nous oublions 90% de l'apprentissage entre chaque apprentissage).

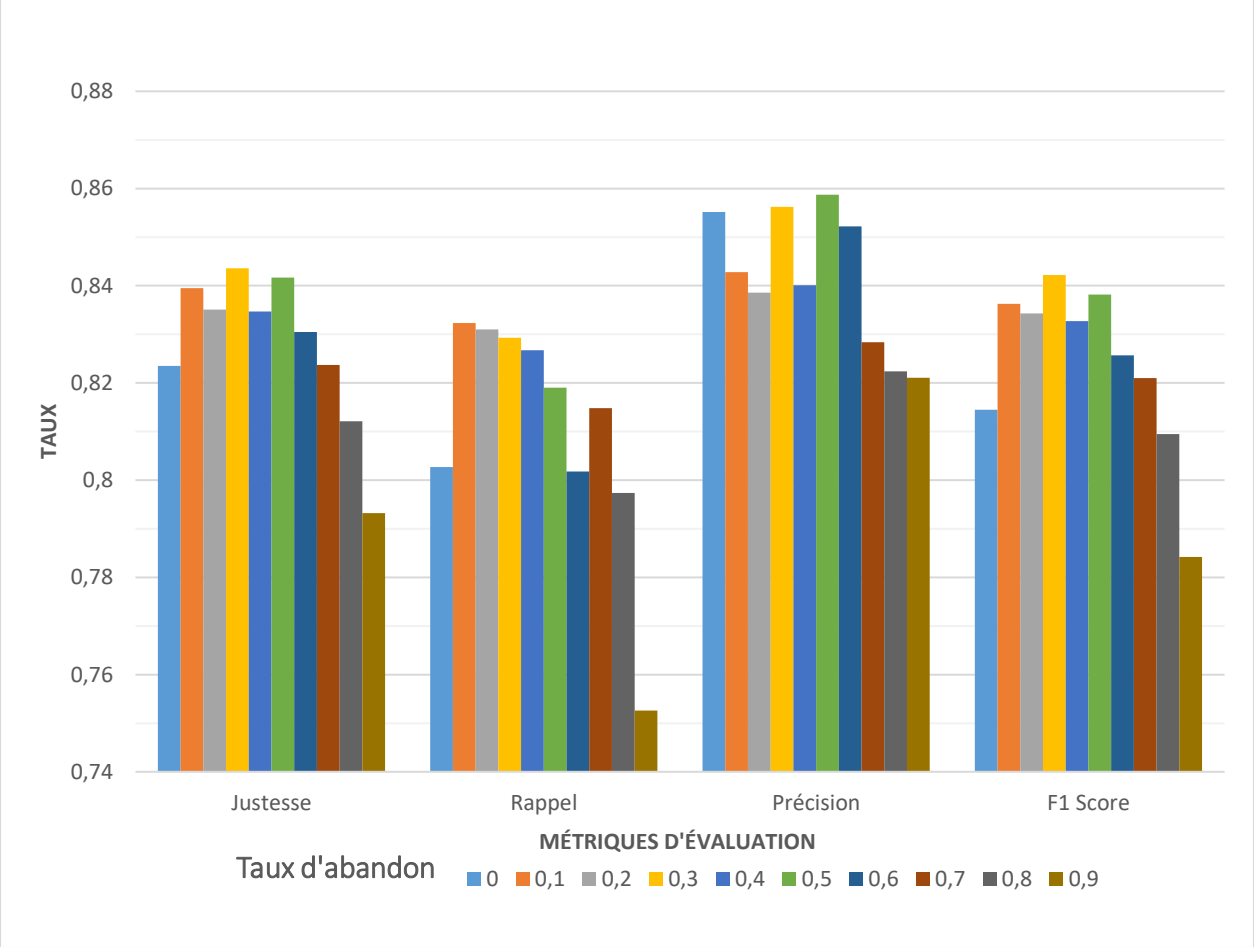


Figure 4.13 Impact du taux d'abandon sur les métriques d'évaluation

Pour l'ensemble des métriques d'évaluation à la Figure 4.13, nous notons que la valeur d'abandon égale à 0.3 permet d'obtenir la meilleure justesse ainsi que le meilleur F1 score. La valeur par défaut de 0.3 est donc bien la valeur à choisir pour un entraînement juste **et** précis (100 entraînements d'une durée totale de 3,340 secondes)

4.4.4 Taux d'apprentissage

Un taux d'apprentissage trop élevé et le minimum de la fonction de perte peut être dépassé, tandis qu'un taux d'apprentissage trop faible entraîne un apprentissage inutilement trop long [118]. Afin de choisir le taux d'apprentissage adéquat permettant d'obtenir les meilleurs scores, nous faisons varier ce dernier de 0.00002 à 0.2. Le paramètre par défaut est 0.002 (bâtonnet de couleur verte sur la Figure 4.14). Les facteurs choisis correspondent en fait à la valeur par défaut 0.002, multipliée ou divisée par les facteurs: 5, 10, 50, 100.

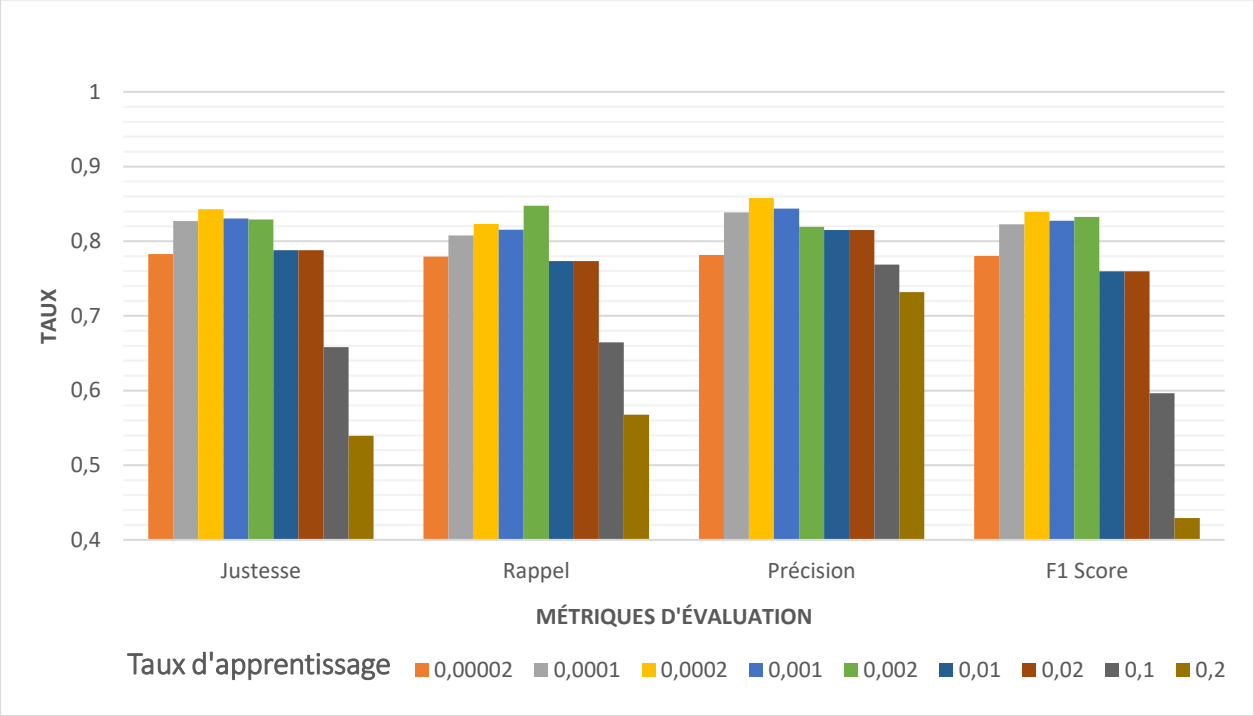


Figure 4.14 Impact du taux d'apprentissage sur les métriques d'évaluation

Nous notons à la Figure 4.14 que les réseaux de neurones ayant la valeur par défaut proposée de 0.002 ainsi que la valeur 0.0002 obtiennent les meilleurs résultats, en termes de F1 Score. Nous choisissons de garder la valeur par défaut de **0.002** proposée par Keras pour la suite des entraînements. En effet, celle-ci obtient un meilleur rappel qui prend la détection des faux négatifs (applications malveillantes non détectées) qui est la base même de la détection de logiciels malveillants (90 entraînements d'une durée totale de 3,055 secondes).

4.4.5 Nombre de Neurones

Afin de choisir le nombre de neurones permettant d'obtenir les meilleurs scores, nous faisons varier ce paramètre de 10 neurones à 4359 sur la première couche L1. Pour choisir ces bornes, nous avons commencé par choisir le nombre de neurones égal au nombre de caractéristiques, soit 3,359, puis nous avons augmenté et diminué ce paramètre. Le pas est de 250, puis nous avons resserré le pas en dessous de 100.

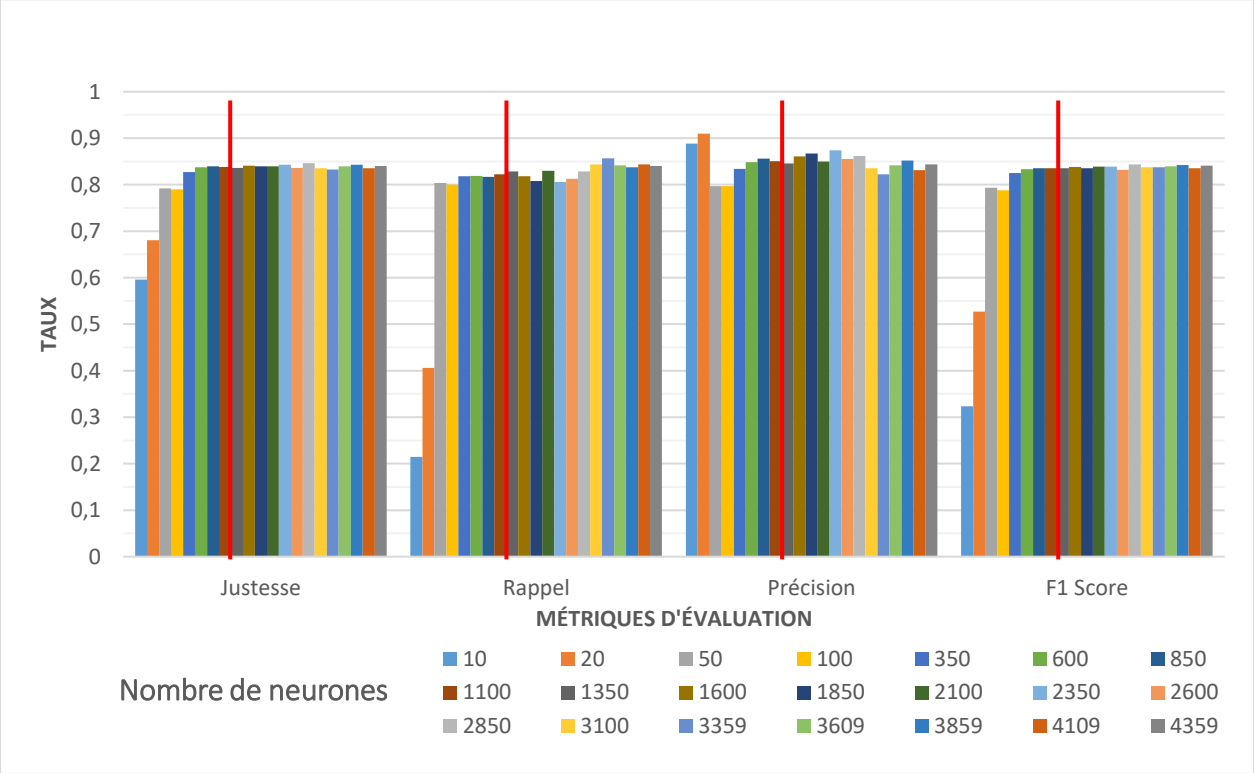


Figure 4.15 Impact du nombre de neurones sur L1 sur les métriques d'évaluation

Nous observons sur la Figure 4.15 qu'augmenter le nombre de neurones au-dessus du nombre de caractéristiques ne donne pas de meilleurs résultats. Nous pouvons surtout remarquer que les résultats sont sensiblement les mêmes de 3,359 neurones jusqu'à 350 neurones. Au-delà de ce seuil, les résultats se détériorent. À la lumière de ces résultats, nous estimons que le nombre minimum de neurones doit être au moins égal à 10% du nombre de caractéristiques pour conserver les mêmes résultats. Afin de mener à bien l'ensemble des entraînements qui représentent une quantité non négligeable, nous décidons de choisir un **nombre de neurones égal au nombre de caractéristiques divisé par trois**. En effet, nous garantissons ainsi les **mêmes résultats** en

diminuant le temps d'entraînement (210 entraînements d'une durée totale de 8,066 secondes, soit 134 minutes ont été réalisés).

4.4.6 Fonction d'activation

Afin de choisir la fonction d'activation permettant d'obtenir les meilleurs scores, nous choisissons de comparer toutes les fonctions d'activation que propose Keras [119]. Ces représentations sont illustrées dans le Tableau 4.5. Chaque fonction d'activation a ses avantages et ses inconvénients[18].

Tableau 4.5 Listes et formules des fonctions d'activation dans L1

Nom de la fonction d'activation	Équation de la fonction d'activation
$\mathbf{sigmoid}(x)$	$\frac{1}{1+\exp(-x)}$
$\mathbf{elu}(x)$	$\begin{cases} x, & x > 0 \\ \frac{1}{\exp(x)-1}, & x < 0 \end{cases}$
$\mathbf{softmax}(x_j)$	$\frac{exp(x_j)}{\sum_{k=1}^{K}\exp(x_k)}$
$\mathbf{selu}(x)$	$scale * elu(x, alpha)$
$\mathbf{softplus}(x)$	$\log(\exp(x)+1)$
$\mathbf{softsign}(x)$	$\frac{x}{(abs(x)+1)}$
$\mathbf{relu}(x)$	$\max(x, 0)$
$\mathbf{tanh}(x)$	$\frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$
$\mathbf{hard}_{\mathbf{sigmoid}}(x)$	$\begin{cases} 0, & x < -2.5 \\ 1, & x > 2.5 \\ 0.2 * x + 0.5, & -2.5 \leq x \leq 2.5 \end{cases}$
$\mathbf{linear}(x)$	x

[18] Machine Learning Cheatsheet (2017), Documentation, https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html, (Consulté en Juin 2019)

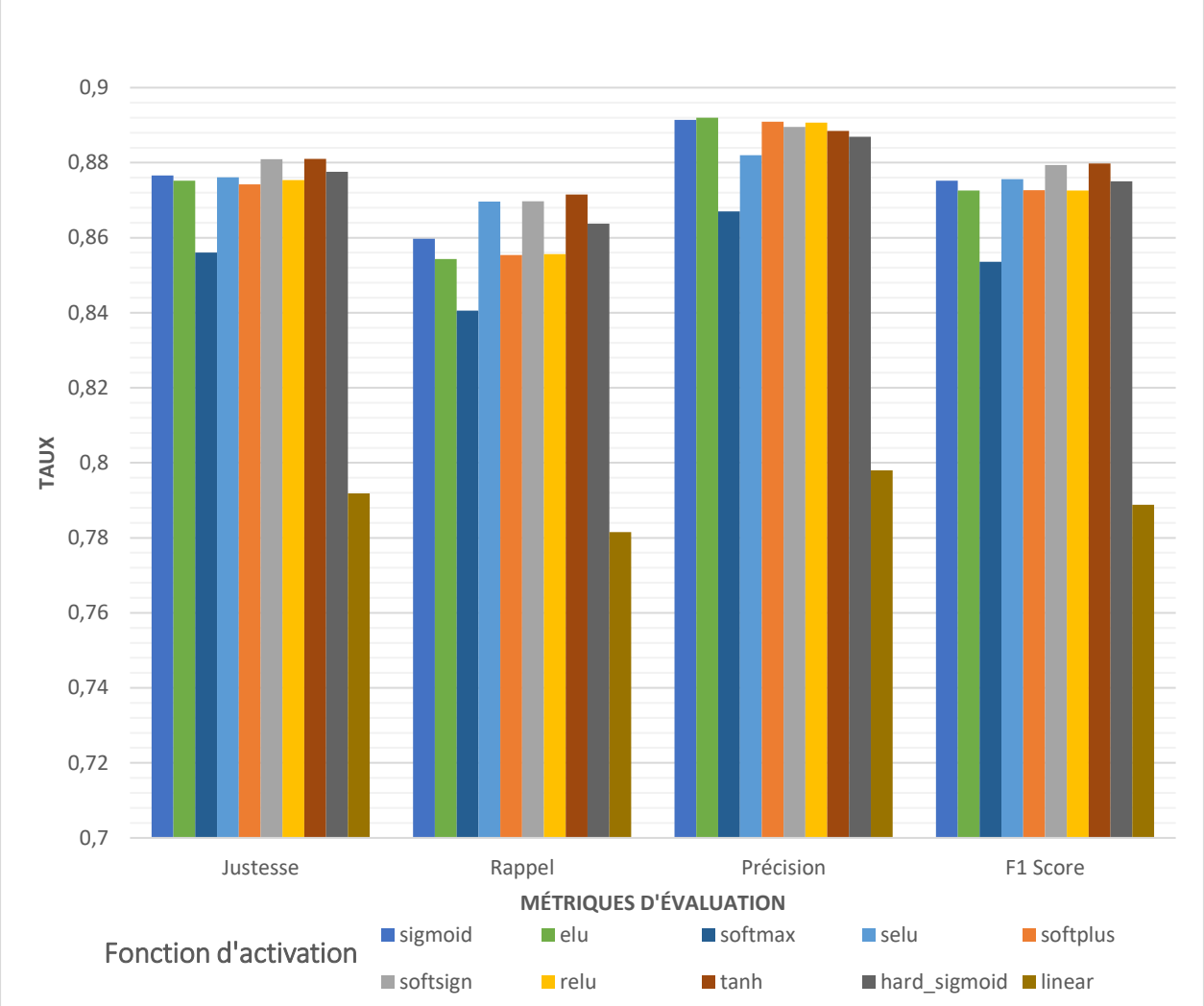


Figure 4.16 Impact de la fonction d'activation sur L1 sur les métriques d'évaluation statique

Nous remarquons à la Figure 4.16 que les résultats sont sensiblement les mêmes quelle que soit la fonction d'activation à l'exception de softmax et surtout de linear dont les résultats sont inférieurs aux autres. Nous avons observé une stabilité des résultats avec la fonction d'activation **Relu** lors de différents entraînements préliminaires. Cela justifie que nous l'ayons choisi pour réaliser les entraînements (100 entraînements d'une durée totale de 4,870 secondes, soit 81 minutes, ont été réalisés).

4.5 Sélection des caractéristiques statiques

Cette section présente les méthodes que nous proposons pour sélectionner les caractéristiques statiques pertinentes qui représentent les logiciels malveillants afin de mieux les détecter.

4.5.1 Sélection de la base de données statiques

Omnidroid est composée initialement de 25,999 caractéristiques statiques. Dans le Tableau 4.6, nous présentons trois répartitions de caractéristiques différentes. La répartition initiale est celle d'Omnidroid à laquelle aucun filtre n'est appliqué. Les deux autres répartitions sont le fruit des méthodes de sélections que nous proposons.

Tableau 4.6 Répartition des caractéristiques statiques après sélection

	Initiale		Étape 1		Étape 2	
Caractéristiques	Nombre	Pourcentage	Nombre	Pourcentage	Nombre	Pourcentage
Receivers	6415	24.7%	216	6.4%	171	8.7%
Activités	6089	23.4%	27	0.8%	1	0.1%
Autorisations	5501	21.2%	710	21.1%	82	4.2%
Services	4365	16.8%	82	2.4%	3	0.2%
API calls	2129	8.2%	1914	57.0%	1369	69.4%
FlowDroid	961	3.7%	95	2.8%	83	4.2%
Opcodes	224	0.9%	224	6.7%	221	11.2%
API Packages	212	0.8%	0	0.0%	0	0.0%
Commandes	103	0.4%	91	2.7%	43	2.2%
Total	**25,999**	**100%**	**3,359**	**100%**	**1973**	**100%**

La première étape (Tableau 4.6, **Étape 1**) proposée est d'enlever les caractéristiques **vides** ainsi que d'enlever les caractéristiques pour lesquelles **une seule application comporte un 1**. Nous passons ainsi de 25,999 à 3,359 caractéristiques. La Figure 4.18 montre que nous préservons les résultats et la Figure 4.19 démontre qu'on peut le faire en gagnant un temps d'entraînement considérable. La première étape proposée est donc pertinente.

La deuxième étape (Tableau 4.6, **Étape 2**) consiste à supprimer les caractéristiques dont la **somme ne dépasse pas 220** pour les autorisations, les opcodes, les appels API, les commandes systèmes et les activités; et qui **ne dépasse pas 22** pour les services, les Receivers, les packages API et pour FlowDroid passant ainsi de 3,359 à 1973. Ces valeurs correspondent aux 10% des valeurs les moins pertinentes, c'est-à-dire les 10% dont les valeurs se ressemblent le plus.

L'objectif visé est de permettre de réduire la taille de la base de données pour permettre un **entraînement plus rapide**, ainsi qu'une plus grande simplicité lors du chargement de la base de données en mémoire RAM. En effet, le chargement de la base de données de 25,999 caractéristiques en mémoire RAM est de 240 secondes, tandis qu'il est seulement de 6 secondes pour la base de données de 3,359 caractéristiques. Il s'agit donc d'une réduction du temps de chargement de **96.8%.**

- 25,999 caractéristiques : 1,126,722 Ko (1,1 Go)
- 3,359 caractéristiques : 150,802 Ko (150 Mo)
- 1,973 caractéristiques : 93,182 Ko (93 Mo)

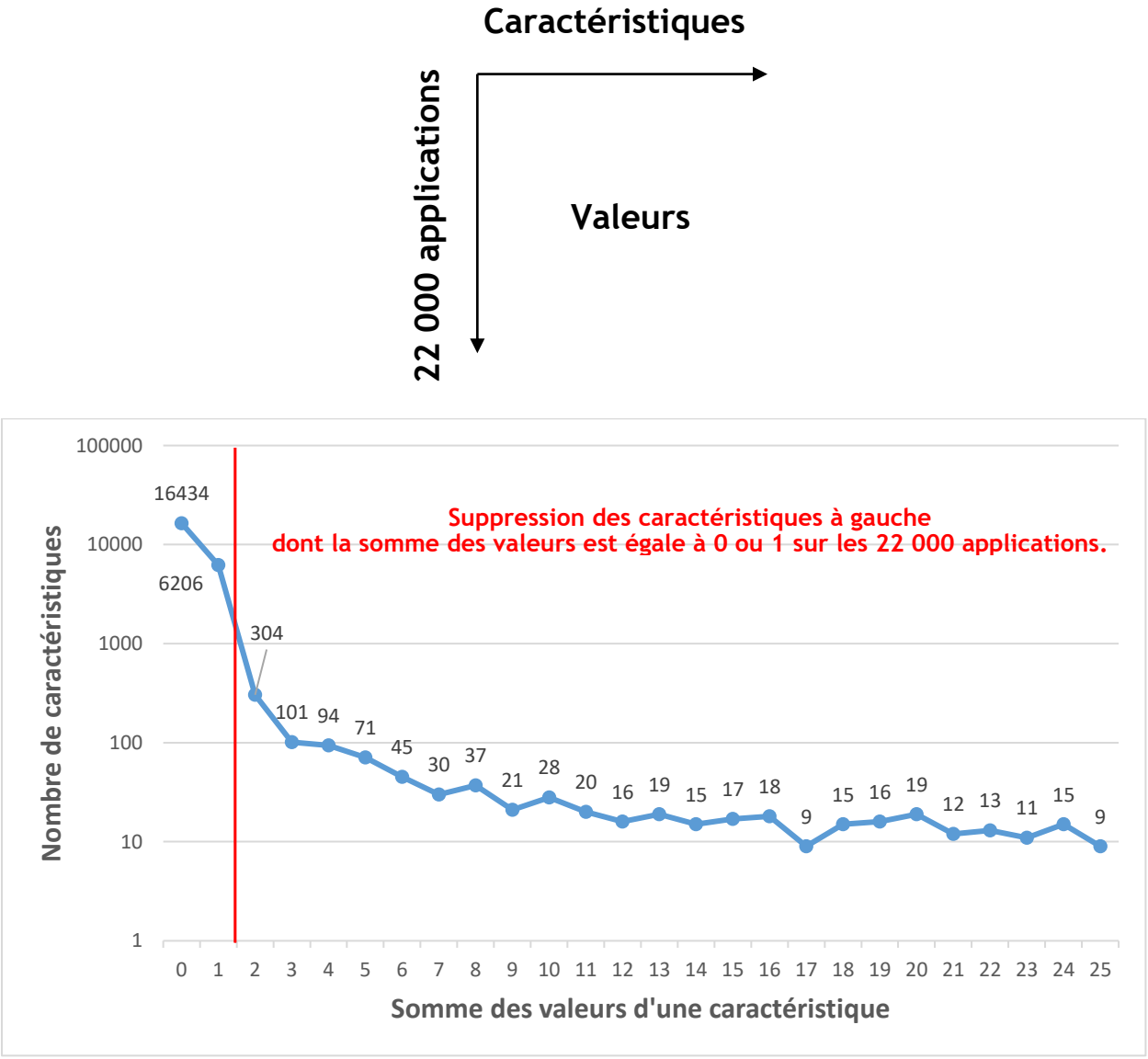


Figure 4.17 Sélection des caractéristiques Statiques (Étape 1)

À la Figure 4.17, nous montrons que **16434 caractéristiques statiques d'Omnidroid** sont **vides** et que **6206 caractéristiques statiques** sont **presque vides**. **Seule une application comporte la valeur un**. Nous verrons dans la Figure 4.18 les résultats de ces mesures.

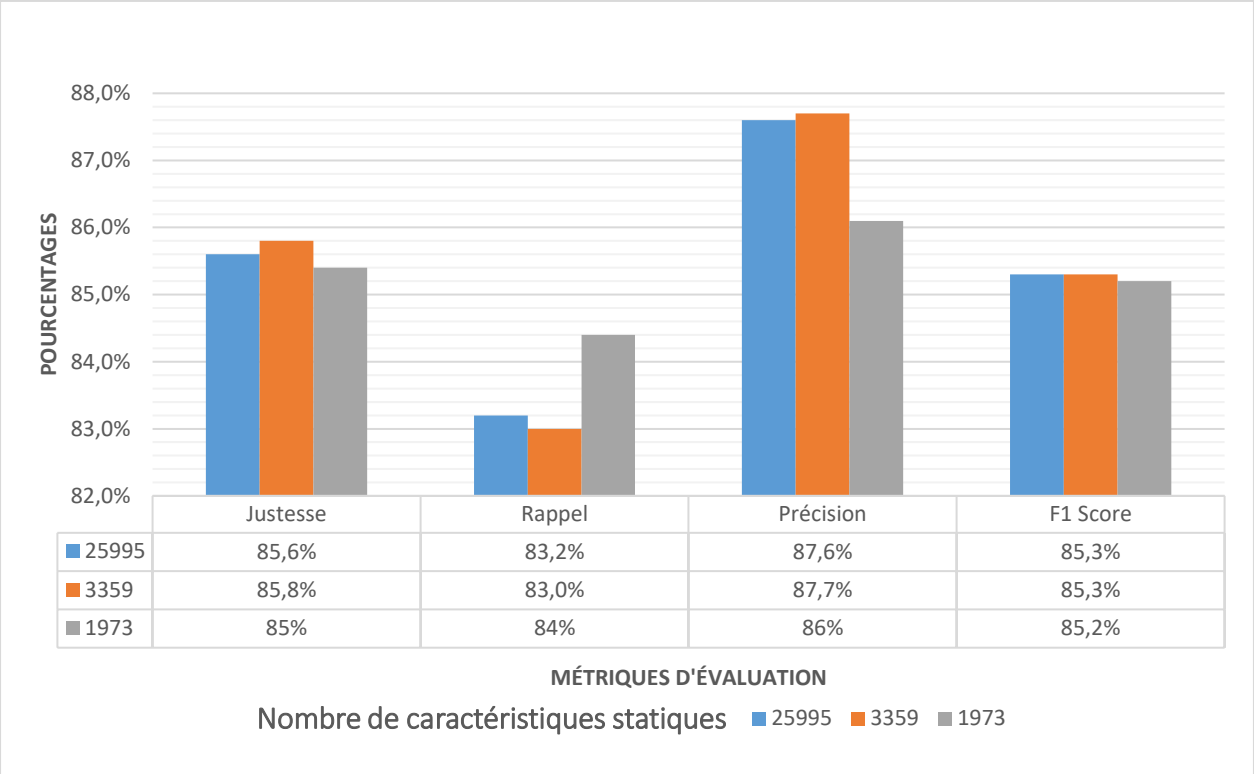


Figure 4.18 Comparaison des bases de données statiques

Bien que les résultats pour les métriques d'évaluation du rappel et de la précision soient différents à la Figure 4.18, le F1 score montre que les résultats sont très ressemblants pour la base de données de 25,999 et 3,359 caractéristiques et nous perdons 0,1% de F1 score pour celle de 1973 caractéristiques. Les caractéristiques bien que nombreuses initialement comportent des colonnes littéralement vides ou bien très peu diversifiées. Ainsi, nous déduisons que les colonnes vides et que les colonnes dont les informations sont presque vides ne permettent pas au réseau de neurones d'améliorer les résultats de détection. Elles ne permettent pas au modèle de détection de faire la différence entre les applications bénignes et les applications malveillantes, puisque quelle que soit l'application il n'y a pas de différence. Ces caractéristiques bien que non utiles pour l'apprentissage ralentissent le temps d'apprentissage et augmentent considérablement les ressources allouées. En effet, nous pouvons observer les différences des temps d'entraînement sur la Figure 4.19. Pour l'obtenir, dix entraînements pour chaque base de données ont été réalisés. La valeur est donc la moyenne sur ces dix entraînements (30 entraînements ont été réalisés pour une durée totale de 10,856 secondes, soit environ trois heures).

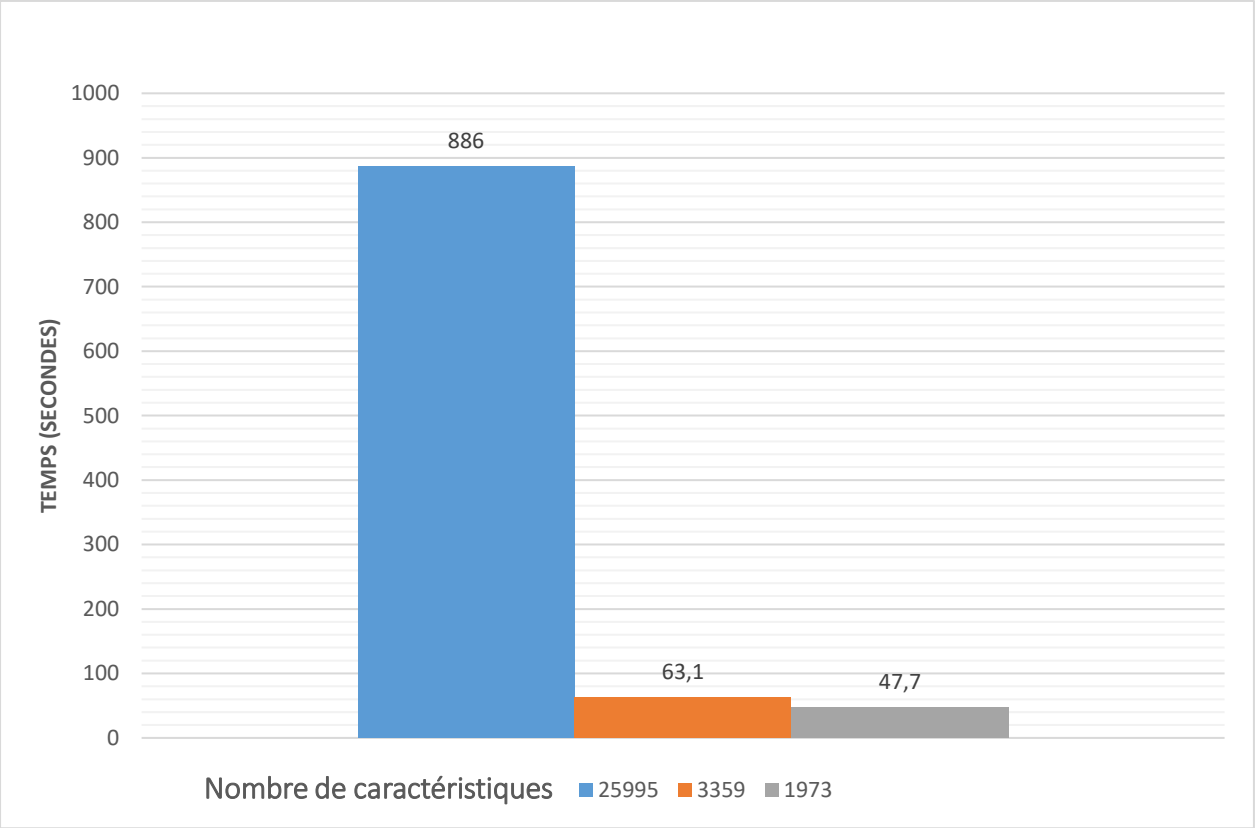


Figure 4.19 Comparaison du temps d'entraînement des bases de données statiques

Nous observons à la Figure 4.19 une baisse significative du temps d'entraînement passant de 886 secondes à 63,1 secondes en entraînant le modèle de détection avec 3,359 caractéristiques au lieu de 25,999. En plus de cela, nous réduisons la mémoire RAM nécessaire. En effet, nous avons observé un pic à 10 Go de mémoire RAM nécessaire pour seulement charger la base de données en mémoire. Nous ajoutons à ces 10 Go de mémoire RAM, la RAM nécessaire pour faire tourner le système d'exploitation et les autres logiciels, ainsi 16 Go de mémoire RAM est le strict minimum pour envisager d'entraîner un réseau de neurones avec Omnidroid. Pour la suite des entraînements, nous décidons de **garder la base de données composée de 3,359 caractéristiques** statiques, afin de pouvoir mener à bien le plan d'expérience dans un temps raisonnable.

4.5.2 Sélection des caractéristiques statiques les plus pertinentes

La sélection des caractéristiques est un processus qui consiste à sélectionner les caractéristiques de la base de données qui contribuent le plus à une bonne prédiction. La présence de caractéristiques non pertinentes peut réduire les performances en termes de rapidité d'entraînement mais peut aussi réduire les résultats, en termes de métriques d'évaluation.

Un modèle Extra Tree peut être utilisé pour estimer l'importance des caractéristiques en les triant par ordre d'importance [120]. Nous avons, dans un premier temps, entraîné 10 réseaux de neurones sur 10 Extra Tree différents pour un total de 100 entraînements. Chaque Extra Tree propose une liste de caractéristiques triées par ordre d'importance différent. Nous choisirons l'arbre qui trouvera le meilleur tri par ordre d'importance parmi les 3,359 caractéristiques.

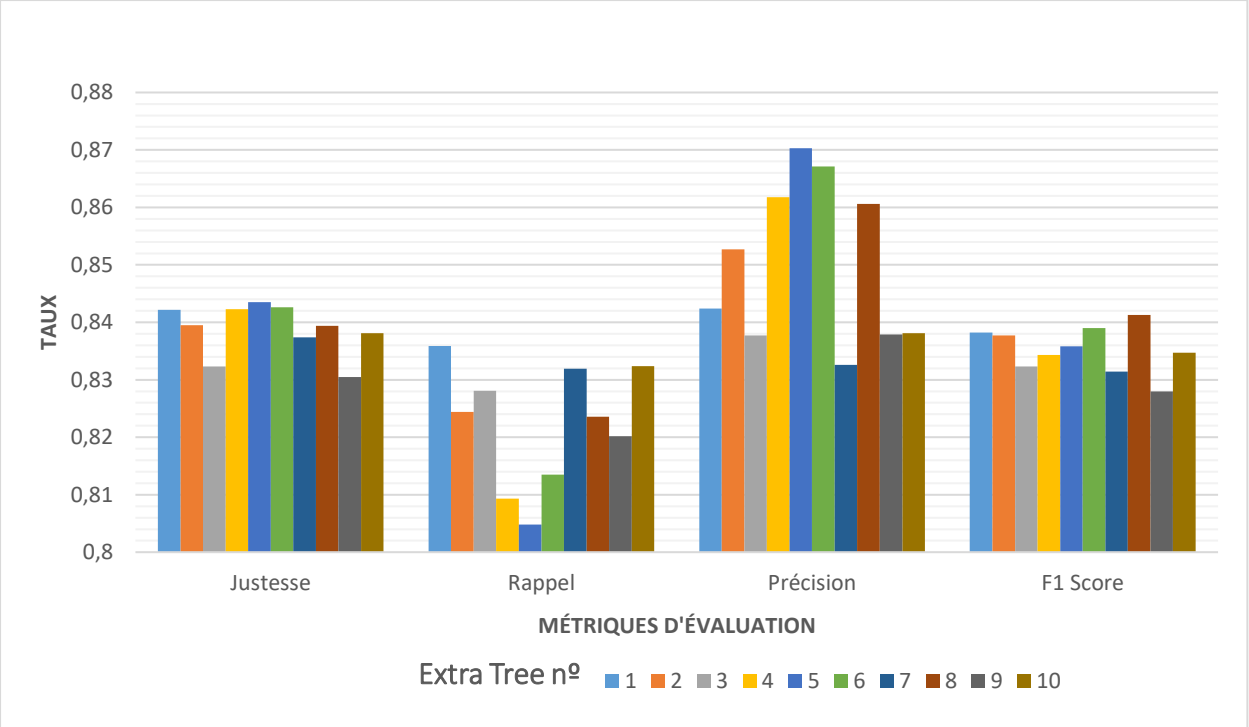


Figure 4.20 Sélection des caractéristiques statiques : choix de l'arbre

Tous les arbres de la Figure 4.20 ont 1,680 caractéristiques. Nous choisissons d'étudier l'arbre dont les résultats sont les plus prometteurs, soit l'Extra Tree nº8. En effet, c'est l'arbre dont le F1 score est le plus élevé. Les résultats des entraînements de l'Extra Tree nº8 sont dans la Figure 4.21.

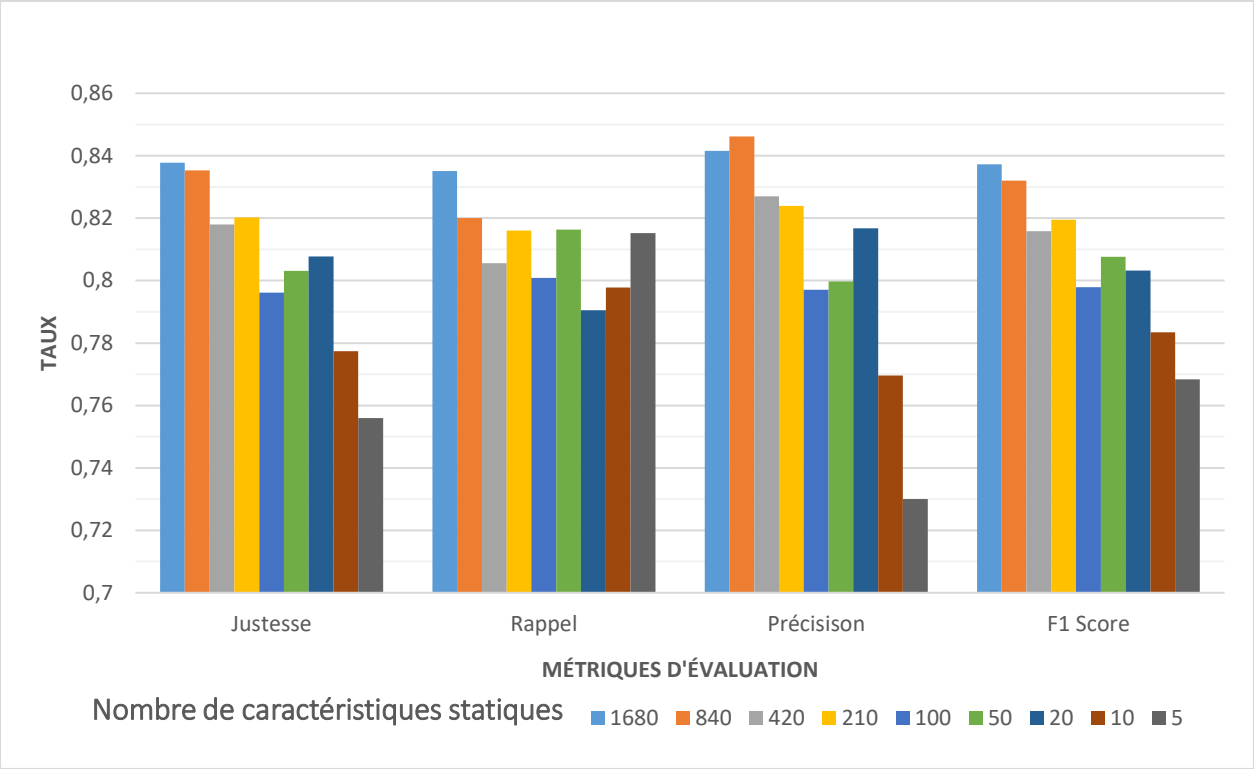


Figure 4.21 Sélection des caractéristiques statiques: ET nº8 (marron)

Nous remarquons à la Figure 4.21 que les résultats ne s'améliorent pas en diminuant le nombre de caractéristiques. Cependant, nous pouvons remarquer un résultat particulièrement intéressant : vingt caractéristiques suffisent pour avoir une justesse de 80,7% et une précision de 81,7% avec seulement 50 itérations.

Nous essayons maintenant une sélection des caractéristiques statiques par une autre méthode : la corrélation de Pearson [121]. Elle nous permet de sélectionner les caractéristiques dont la corrélation est la plus haute entre la caractéristique étudiée et l'étiquette application malveillante ou bénignes.

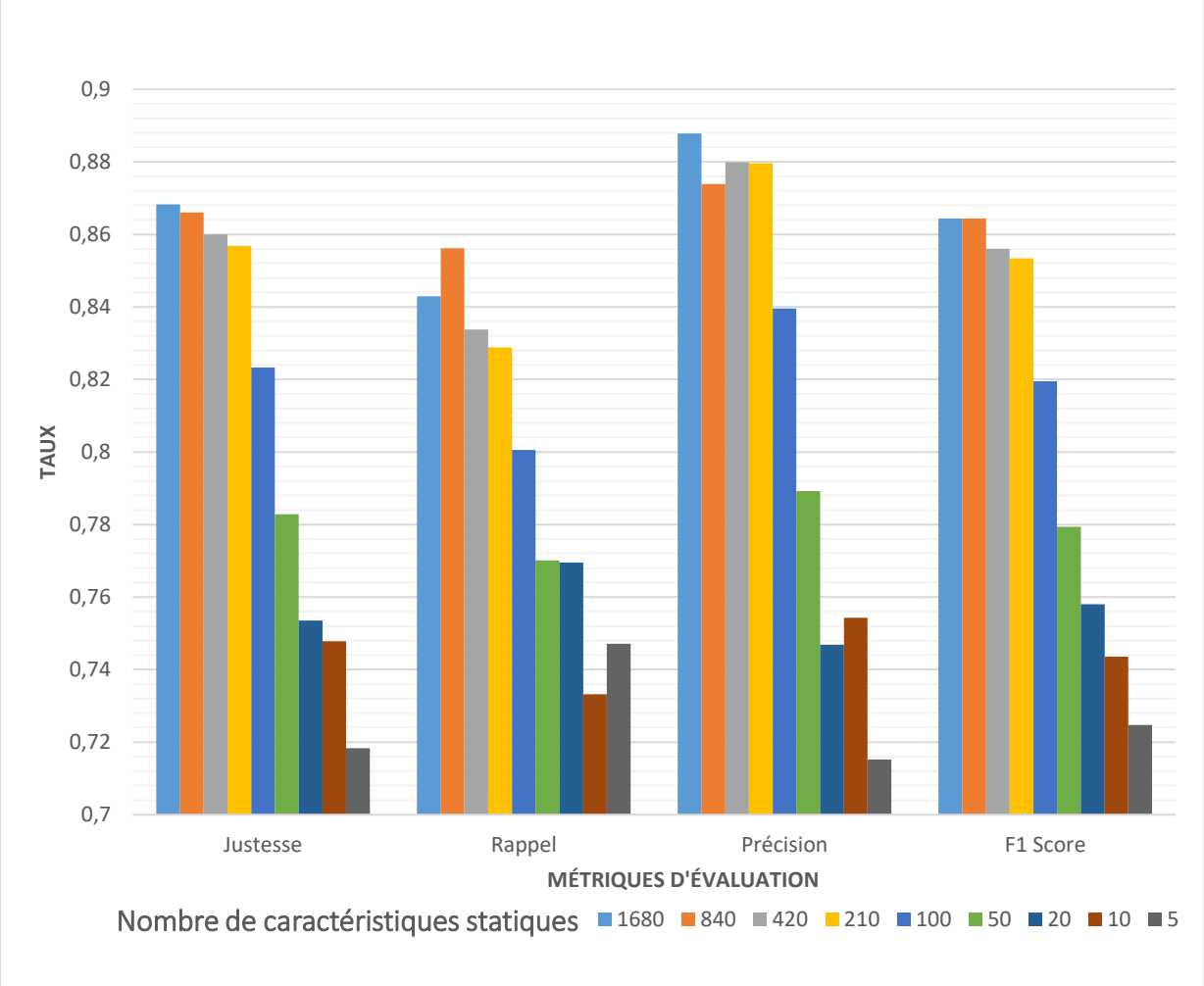
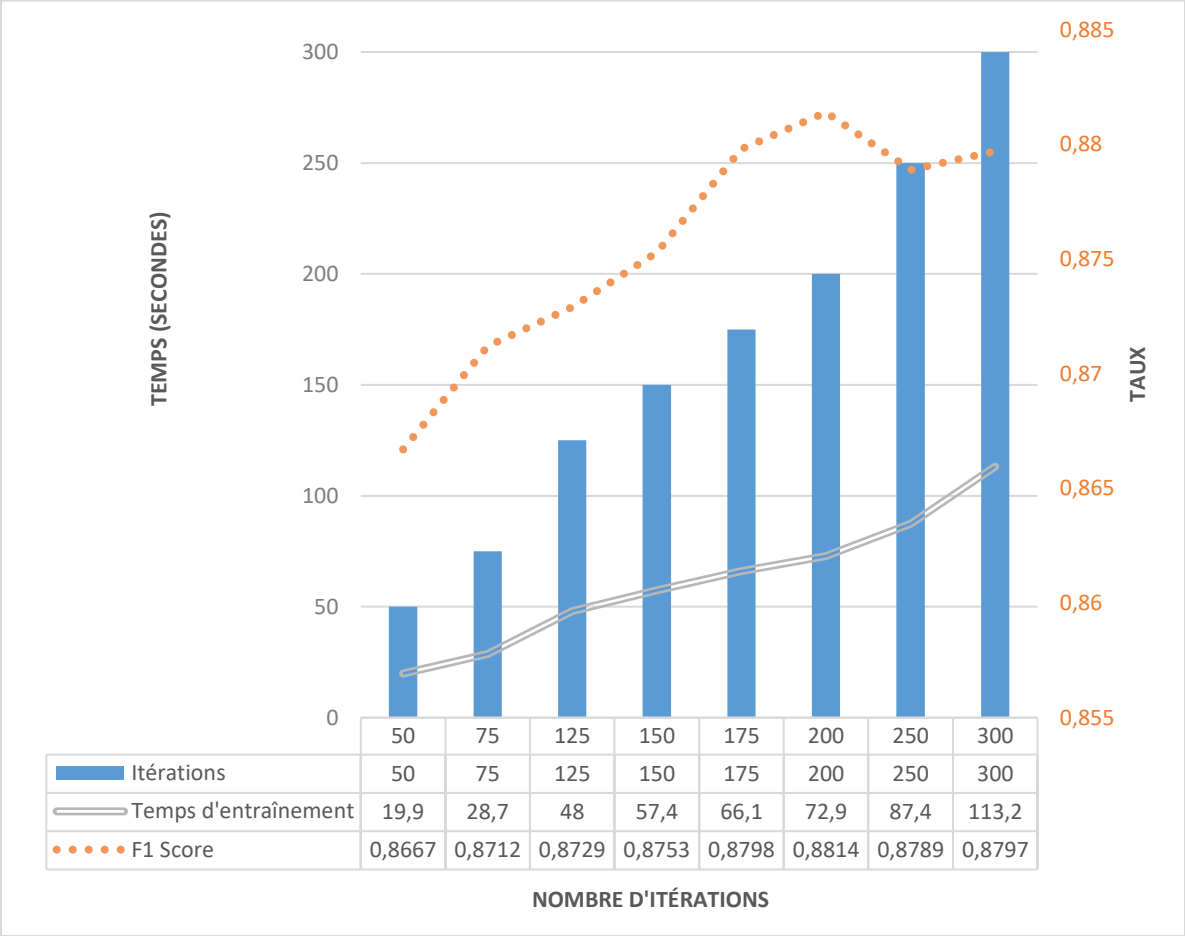


Figure 4.22 Sélection des caractéristiques statiques: Corrélation de Pearson

Nous observons à la Figure 4.22 que la sélection de caractéristiques avec la corrélation de Pearson obtient de meilleurs résultats que la méthode basée sur les ExtraTree. En effet, les réseaux de neurones de 1680 et 840 caractéristiques obtiennent 86,44% de F1 score comparé à 83,7% de F1 score pour la méthode basée sur les ExtraTree. Nous exclurons donc la méthode des ExtraTree pour la sélection des caractéristiques statiques. À partir de ces résultats, nous décidons d'étudier plus en détails ces deux réseaux de neurones.

Nous faisons varier le nombre d'itérations pour trouver la valeur du nombre d'itérations nous permettant d'obtenir les meilleurs résultats pour le réseau de neurones de 840 caractéristiques (cf. Figure 4.22). Pour rappel nous avions trouvé 250 itérations pour le réseau dynamique (cf. §4.4.2).



	50	75	125	150	175	200	250	300
Itérations	50	75	125	150	175	200	250	300
Temps d'entraînement	19,9	28,7	48	57,4	66,1	72,9	87,4	113,2
F1 Score	0,8667	0,8712	0,8729	0,8753	0,8798	0,8814	0,8789	0,8797

Figure 4.23 Impact du nombre d'itérations sur F1 Score (Pearson 840 statiques)

D'après les résultats de la Figure 4.23, nous déduisons que le nombre d'itérations permettant d'obtenir les meilleurs scores avec 840 caractéristiques est de **200 itérations**. Le réseau de neurones obtient un F1 score de 88,14 %. À partir de 200 itérations, nous n'observons pas d'amélioration des résultats, donc nous choisissons 200 itérations pour le réseau de neurones de 840 caractéristiques.

Dans la même optique que les entraînements ayant 840 caractéristiques, nous cherchons à dépasser les résultats obtenus en choisissant cette fois-ci 1680 caractéristiques.

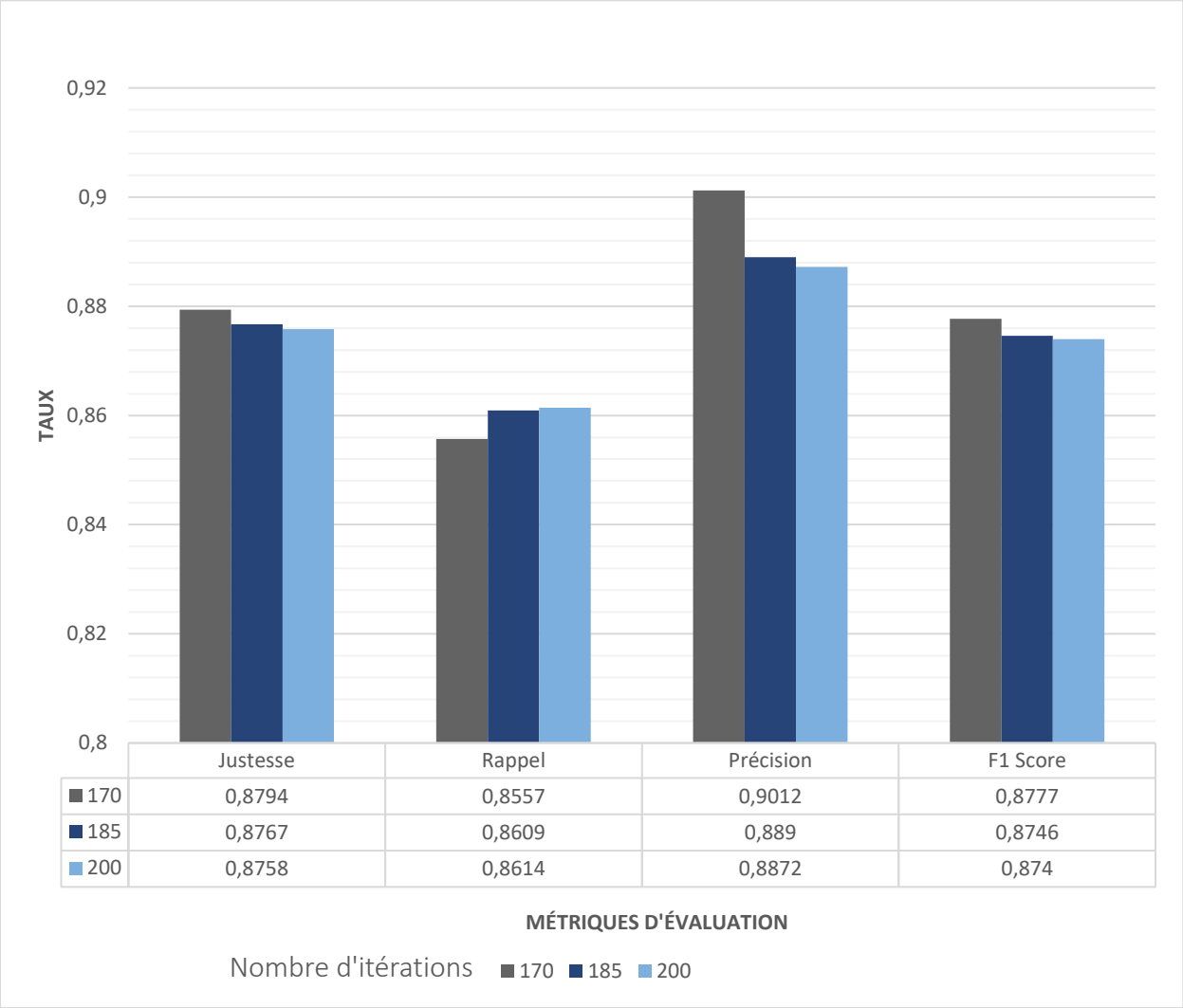


Figure 4.24 Impact du nombre d'itérations (ExtraTree 1680 Statiques)

Les résultats à la Figure 4.24 sont inférieurs de 0.5% aux modèles avec 840 caractéristiques. Ainsi, les modèles de 1680 caractéristiques seront écartés.

Tous les résultats précédents ont été effectués avec une seule couche L1 de neurones. Nous envisageons maintenant d'ajouter une seconde couche L2 (hidden layer) afin d'améliorer les résultats. Ces résultats sont présentés aux Figures 4.25 et 4.30.

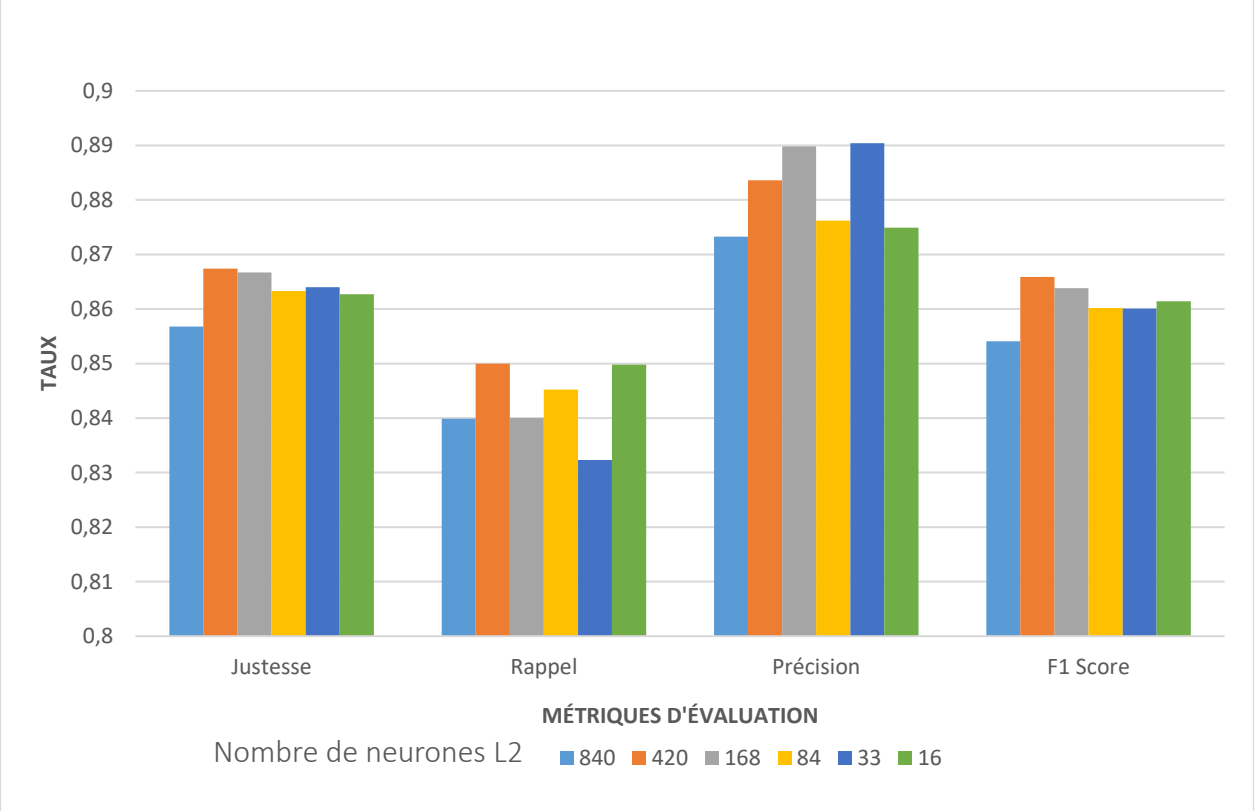


Figure 4.25 Impact du nombre de neurones de L2 sur les métriques d'évaluation

Le nombre de neurones de la deuxième couche ne permet pas d'améliorer les résultats **au-delà de 420 neurones,** d'après la Figure 4.25. Nous choisirons donc 420 neurones sur la couche L2.

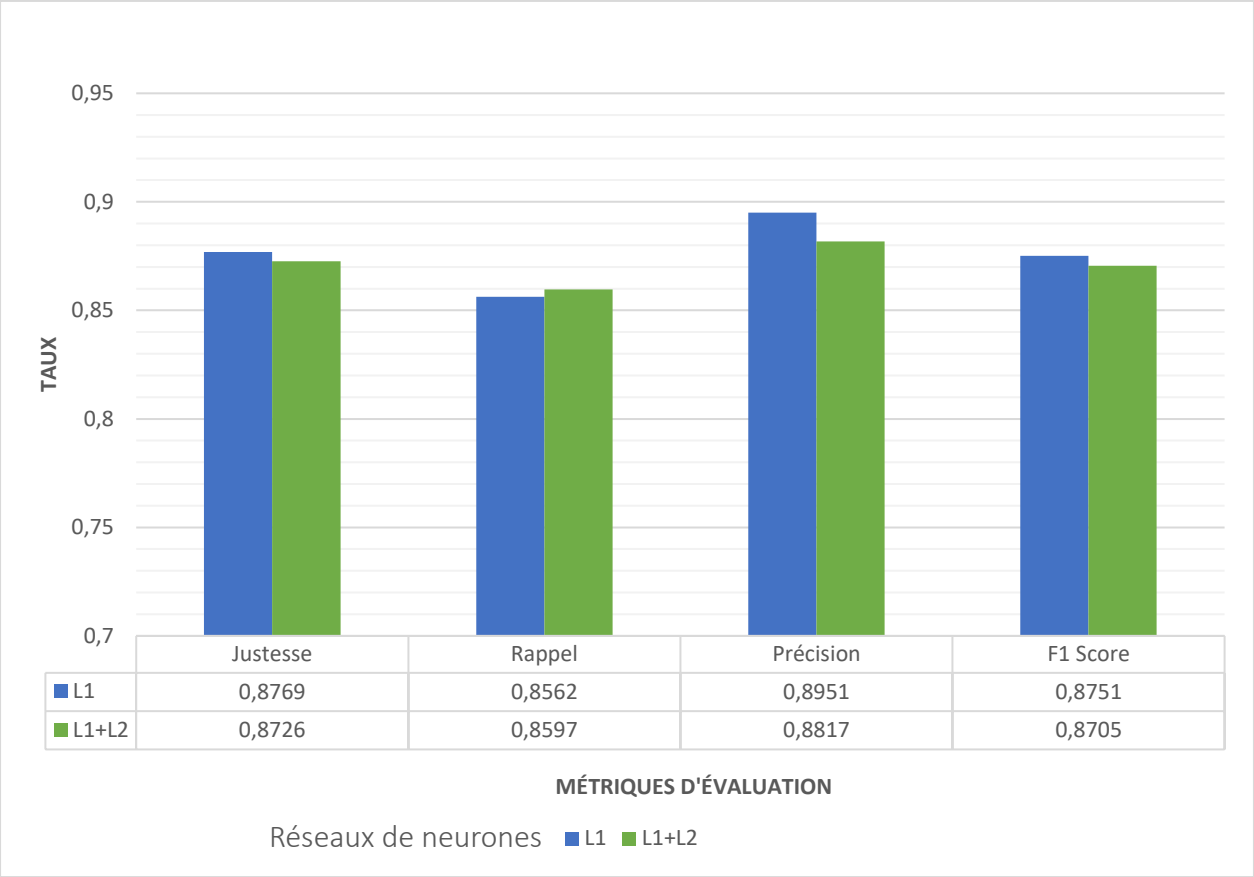


Figure 4.26 Impact de l'ajout d'une 2e couche de neurones sur les métriques d'évaluation

Nous observons à la Figure 4.26 qu'ajouter une deuxième couche L2 n'améliore pas les résultats. Nous exclurons donc l'ajout d'une deuxième couche de neurones.

4.6 Sélection des caractéristiques dynamiques

Cette section présente les méthodes que nous proposons pour sélectionner les caractéristiques dynamiques pertinentes qui représentent les logiciels malveillants afin de mieux les détecter.

4.6.1 Sélection de la base de données dynamiques

De la même manière que pour la sélection des caractéristiques statiques, nous effectuons une sélection des caractéristiques dynamiques par la même approche que nous proposons. Nous enlevons donc les caractéristiques dont les colonnes sont vides ou bien égales à 1 passant ainsi de 5,932 à 3,722. Le chargement de la base de données de 5,932 caractéristiques en mémoire RAM est de 16 secondes tandis qu'il est de 6 secondes pour la base de données de 3,722 caractéristiques.

- 5,932 : 255,703 Ko (225 Mo)
- 3,722 : 160,514 Ko (160 Mo)
- 310 : 13,437 Ko (13 Mo)

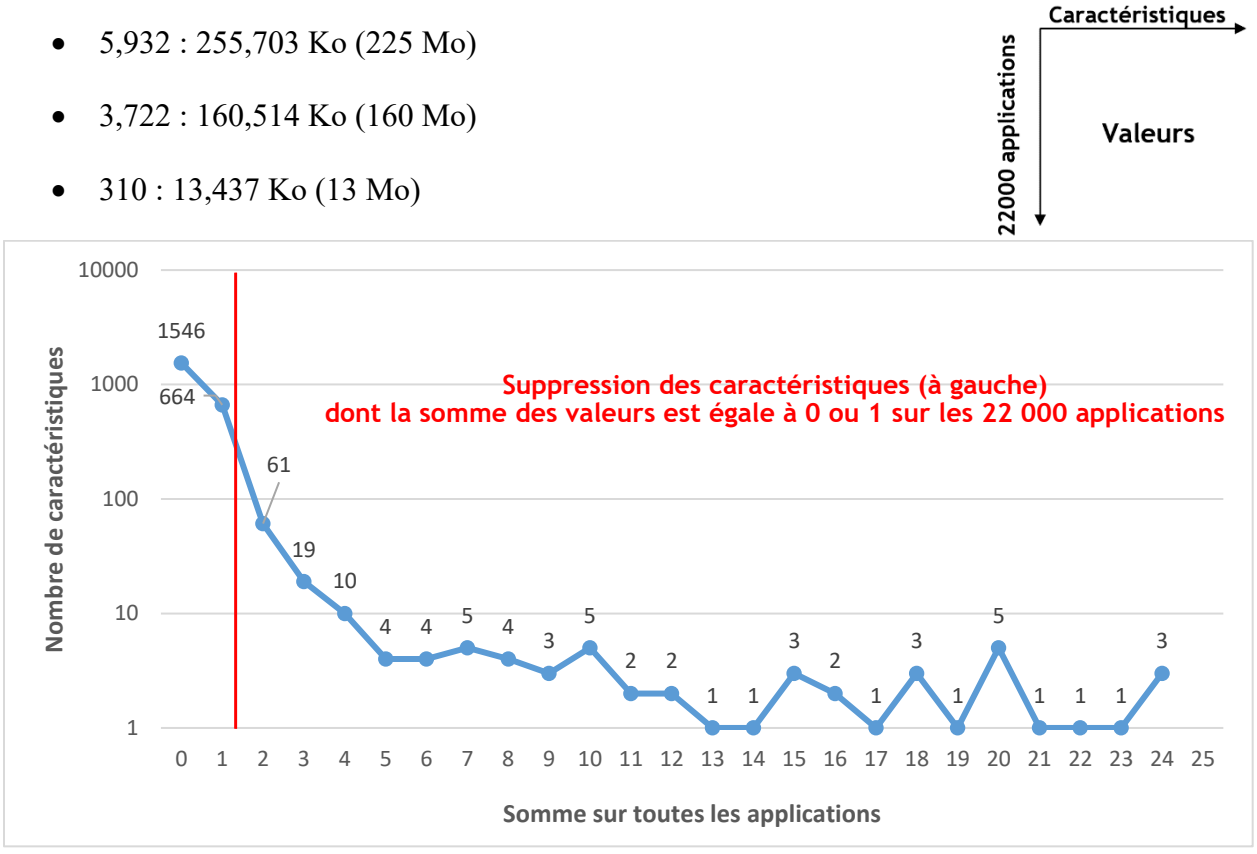


Figure 4.27 Sélection des caractéristiques Dynamiques (Étape 1)

Nous remarquons à la Figure 4.27 que 1546 caractéristiques dynamiques d'Omnidroid sont vides et que 664 caractéristiques dynamiques sont presque vides. Seule une application comporte la valeur un. Il s'agit d'un total de 2210 caractéristiques apparemment inutiles. Pour obtenir la base

de données des 310 caractéristiques les plus diversifiées nous avons supprimé toutes les caractéristiques dont la somme était inférieure ou égale à 20.

Tableau 4.7 Répartition des caractéristiques dynamiques après sélection

	Initiale		**Étape 1**		**Étape 2**	
Caractéristiques	Nombre	Pourcentage	Nombre	Pourcentage	Nombre	Pourcentage
Opennet	1483	25.0%	862	23.2%	0	0.0%
Sendnet	1186	20.0%	718	19.3%	0	0.0%
Fdaccess	937	15.8%	630	16.9%	0	0.0%
Dataleaks	867	14.6%	509	13.7%	134	43.2%
Recvnet	837	14.1%	540	14.5%	0	0.0%
Cryptousage	488	8.2%	369	9.9%	156	50.3%
Dexclass	134	2.3%	94	2.5%	20	6.5%
Total	**5,932**	**100%**	**3,722**	**100%**	**310**	**100%**

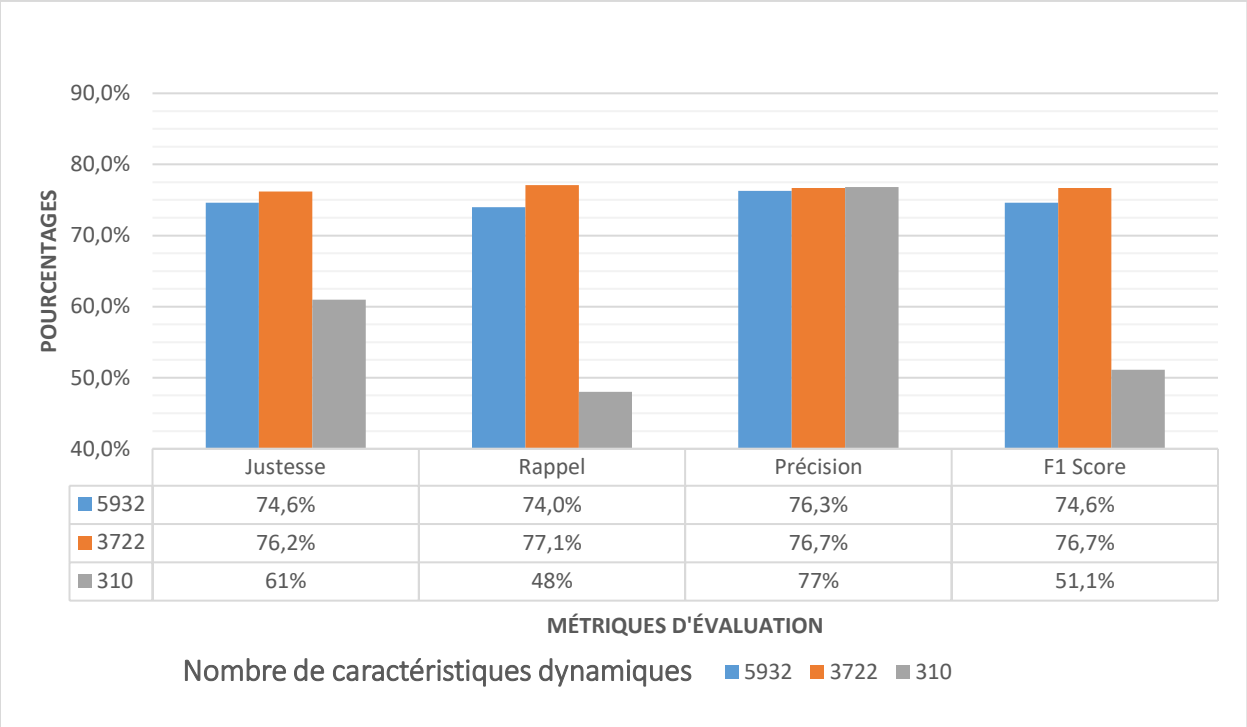


Figure 4.28 Comparaison des bases de données dynamiques

Nous observons à la Figure 4.28 une hausse des résultats avec un nombre de caractéristiques égal à 3,722. Cependant, ce n'est pas le cas pour un nombre égal à 310 ; cette chute considérable nous permet d'exclure cette base de données. Nous avons choisi de garder 3,722 caractéristiques dynamiques, puisque toutes les métriques d'évaluation sont plus élevées qu'initialement (300 entraînements d'une durée totale de 2,455 secondes soit 41 minutes ont été réalisés).

4.6.2 Sélection des caractéristiques dynamiques les plus pertinentes

Avec la corrélation de Pearson

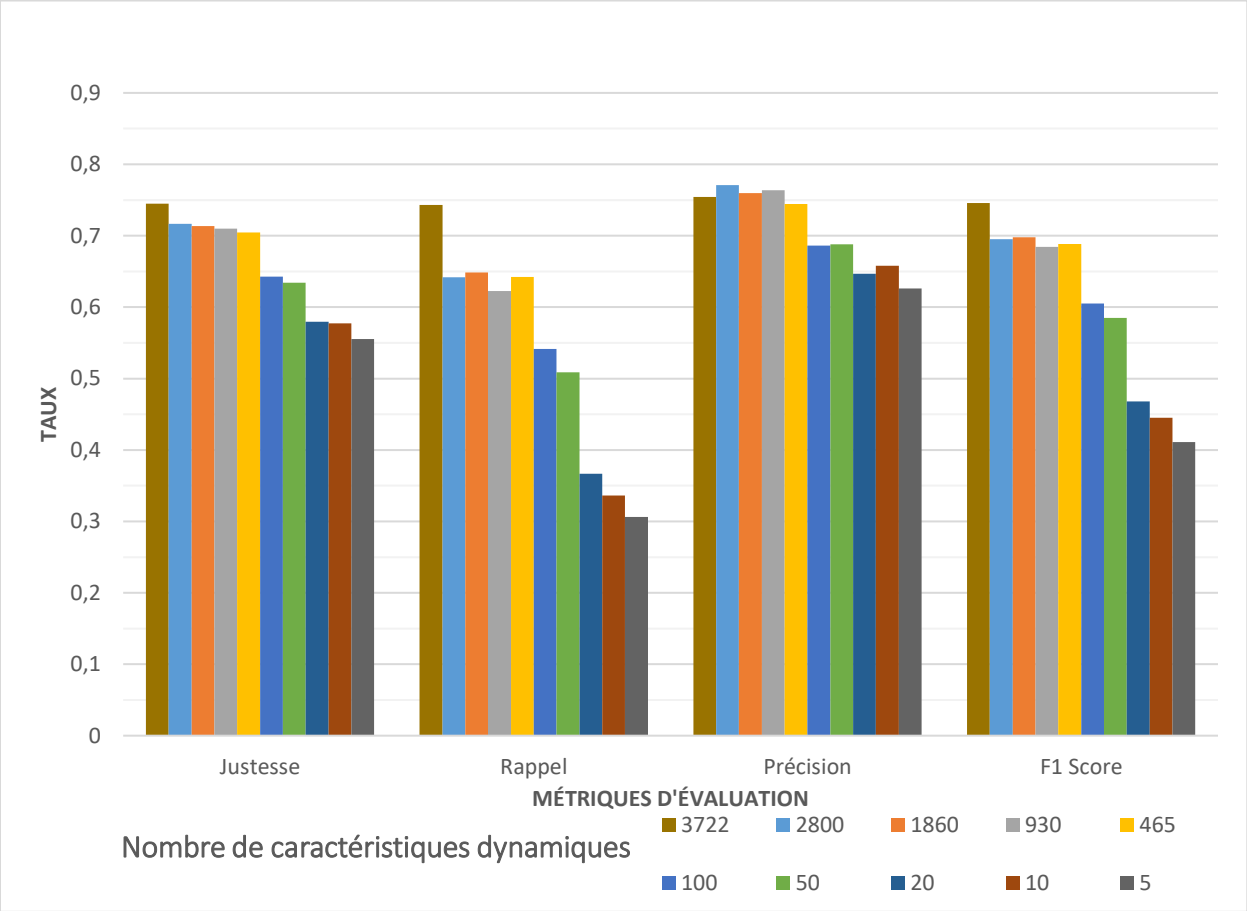


Figure 4.29 Sélection des caractéristiques dynamiques: Corrélation de Pearson

Nous observons à la Figure 4.29 que la sélection de caractéristiques dynamiques entraîne une baisse des résultats sur la base de données de 3722 caractéristiques. Cette base de données sera donc écartée, puisqu'elle ne remplit pas l'objectif qui est d'augmenter les scores (100 entraînements d'une durée totale de 1,764 secondes soit 30 minutes ont été réalisés).

Avec les Extra Tree

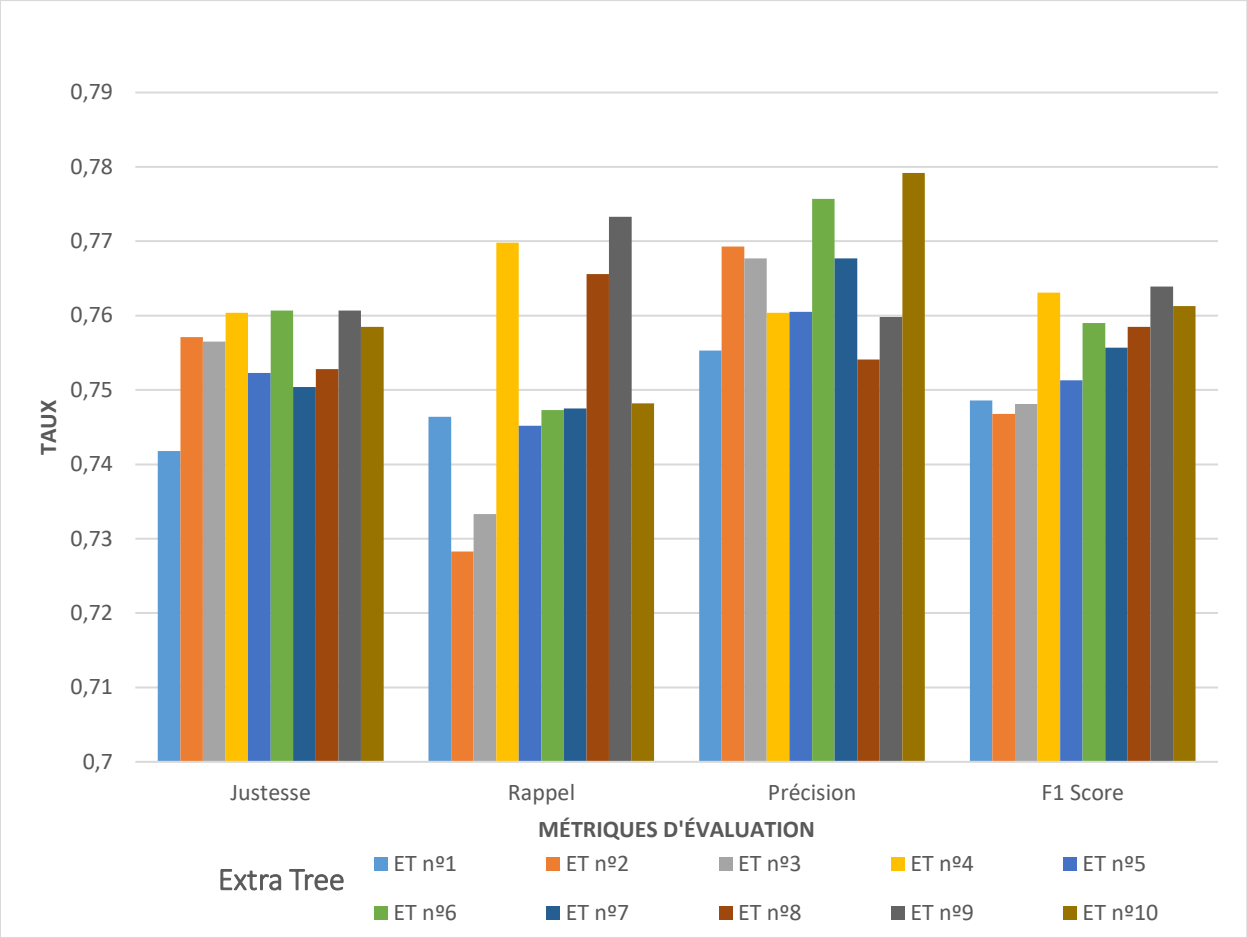


Figure 4.30 Sélection des caractéristiques dynamiques : choix du modèle ExtraTree

Nous observons à la Figure 4.30 une variation en fonction des différents tris des arbres. L'arbre nº4 et l'arbre nº9 présentent tous les deux des résultats légèrement au-dessus des autres (100 entraînements d'une durée totale de 1,839 secondes soit 30 minutes ont été réalisés).

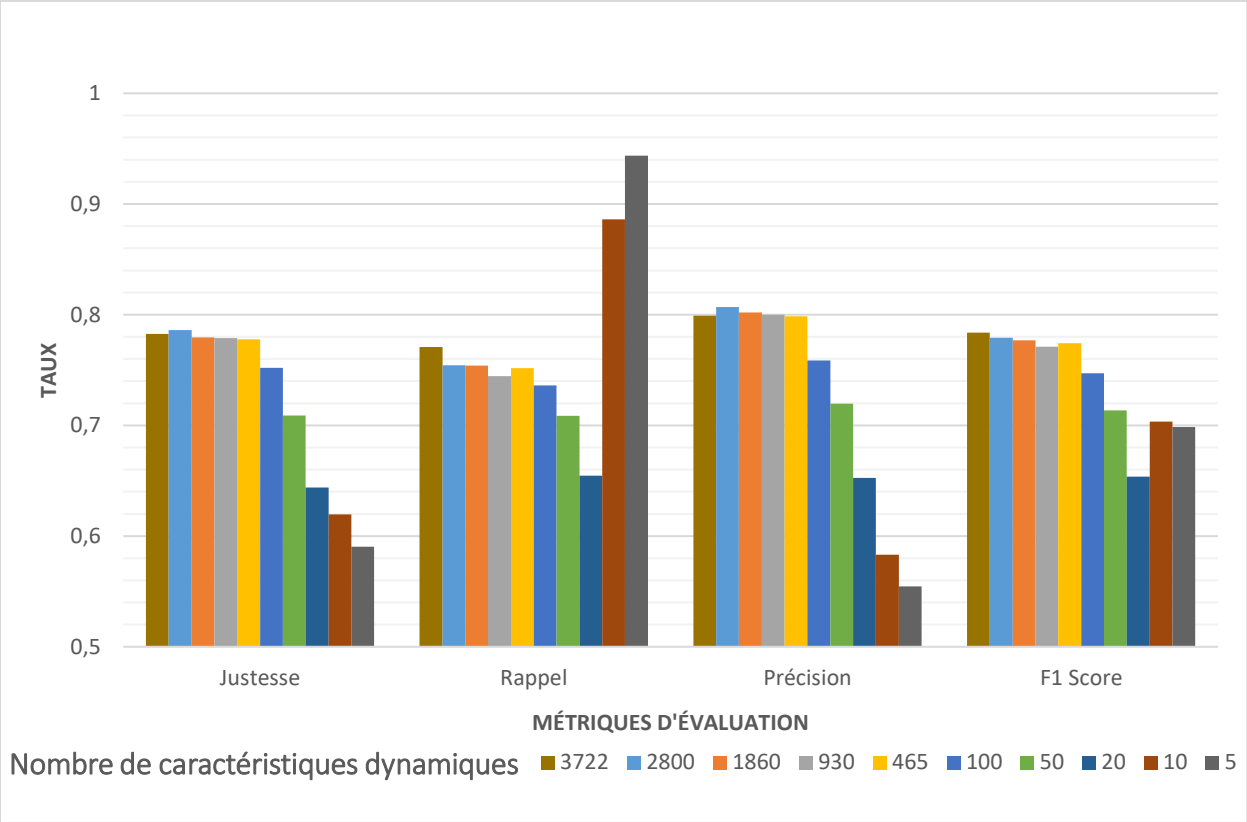


Figure 4.31 Sélection des caractéristiques Dynamiques: ExtraTree ET nº4 (jaune)

Nous observons à la Figure 4.31 de manière générale que moins il y a de caractéristiques, plus les résultats décroissent. Les résultats sont sensiblement les mêmes pour 3,722, 2800, 1860, 930 et 465 caractéristiques. Ensuite, nous notons une baisse non négligeable. Sur la Figure 4.31 présentée, c'est la moyenne de 10 entraînements qui est représentée. Il est donc tout à fait possible que le meilleur entraînement du modèle à 465 caractéristiques présente de meilleurs résultats que le pire des modèles avec 3,722. Il s'agit là d'une tendance générale. Nous avons cependant noté que le modèle présentant les meilleurs résultats parmi tous est celui ayant 3,722 caractéristiques suivi de très près par le meilleur des modèles ayant 2,800 caractéristiques. Nous restons donc avec **3,722** caractéristiques dynamiques (100 entraînements d'une durée totale de 9,264 secondes soit 154 minutes ont été réalisés avec 250 itérations par entraînement ce qui explique que les métriques d'évaluation soient plus élevées qu'avec 50 itérations).

4.7 Entraînement proposé

Dans cette section, nous montrons les courbes représentant les métriques d'évaluation pour visualiser les différences lors de l'entraînement d'un modèle entraîné avec 3,359 caractéristiques et un modèle entraîné avec 840 caractéristiques statiques. Tous les hyperparamètres sont les mêmes que précédemment à l'exception du nombre d'itérations qui a été augmenté à 1000.

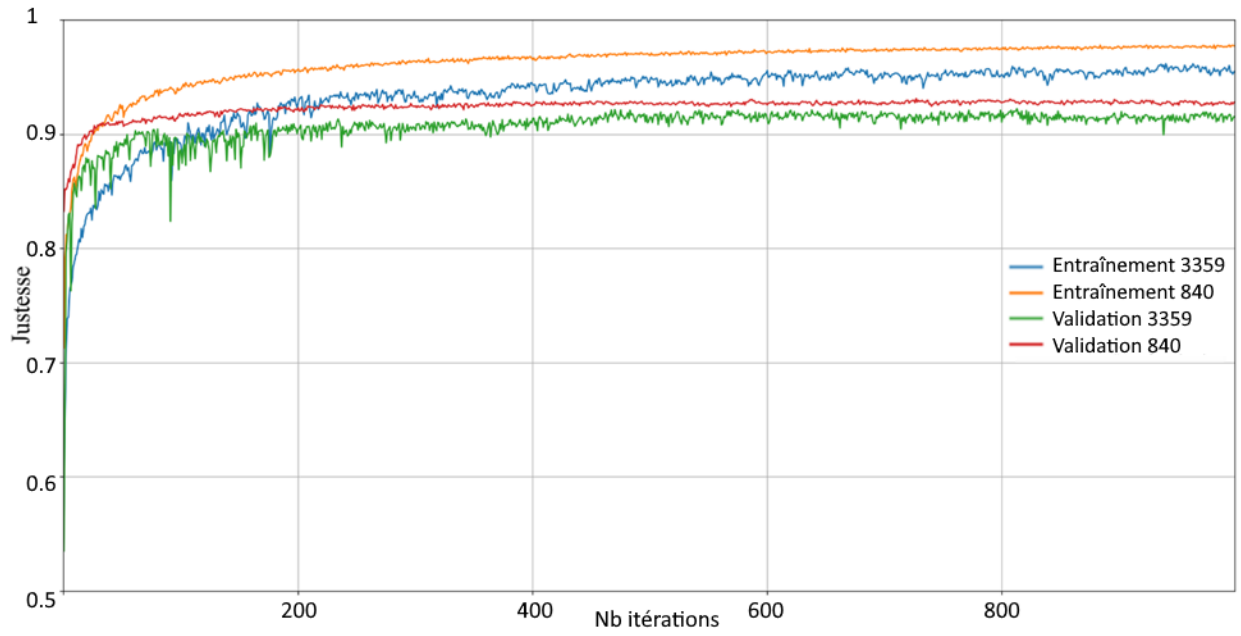


Figure 4.32 Courbe d'apprentissage : justesse

Nous observons à la Figure 4.32 que la réduction du nombre de caractéristiques permet d'augmenter la justesse sur l'ensemble d'entraînement et sur l'ensemble de validation. Nous retrouvons bien l'avantage nº2 énoncé au chap, 3 §3.1.2.3, à savoir améliorer la précision et la justesse. En effet, la courbe d'entraînement du modèle avec 840 caractéristiques est au-dessus de la courbe d'entraînement du modèle avec 3,359 caractéristiques. Il est en de même que pour les courbes de validation.

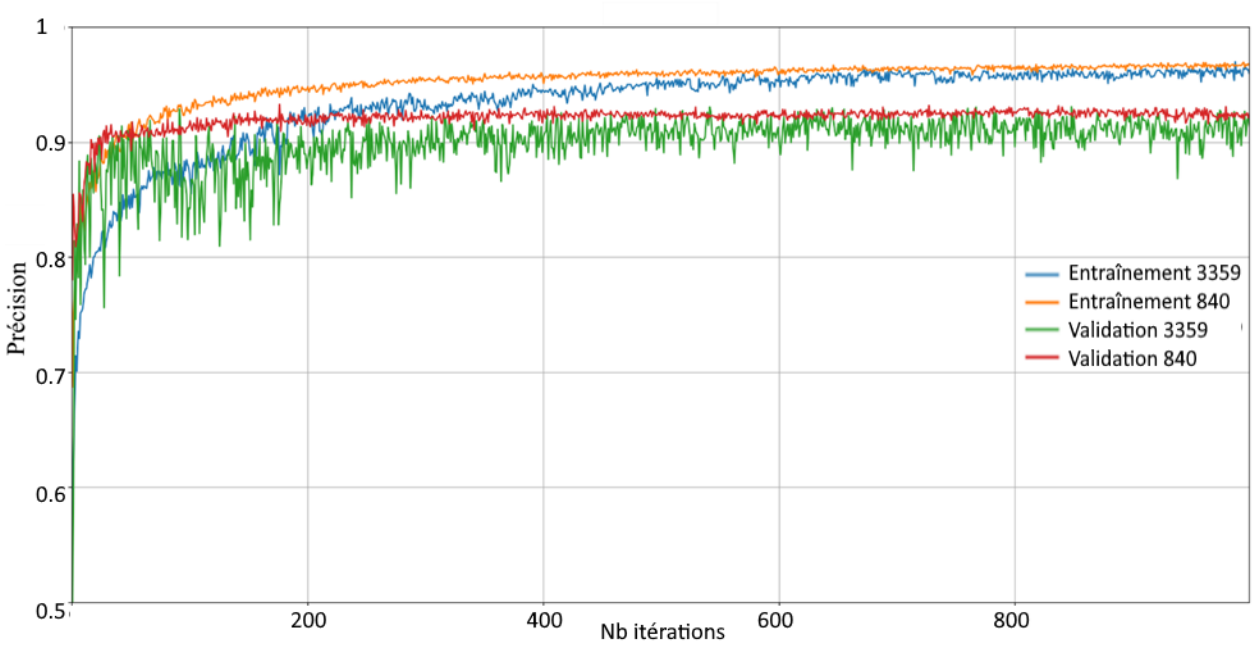


Figure 4.33 Courbe d'apprentissage : précision

Nous observons à la Figure 4.33 que la réduction du nombre de caractéristiques permet d'augmenter la précision sur l'ensemble d'entraînement et sur l'ensemble de validation. Nous retrouvons également l'avantage nº2 énoncé au chap, 3 §3.1.2.3, à savoir améliorer la précision et la justesse. Nous observons aussi une réduction du bruit sur la courbe rouge. Nous pouvons déduire que nous réduisons le sur-apprentissage, d'après l'avantage nº3 énoncé au chap, 3 §3.1.2.3.

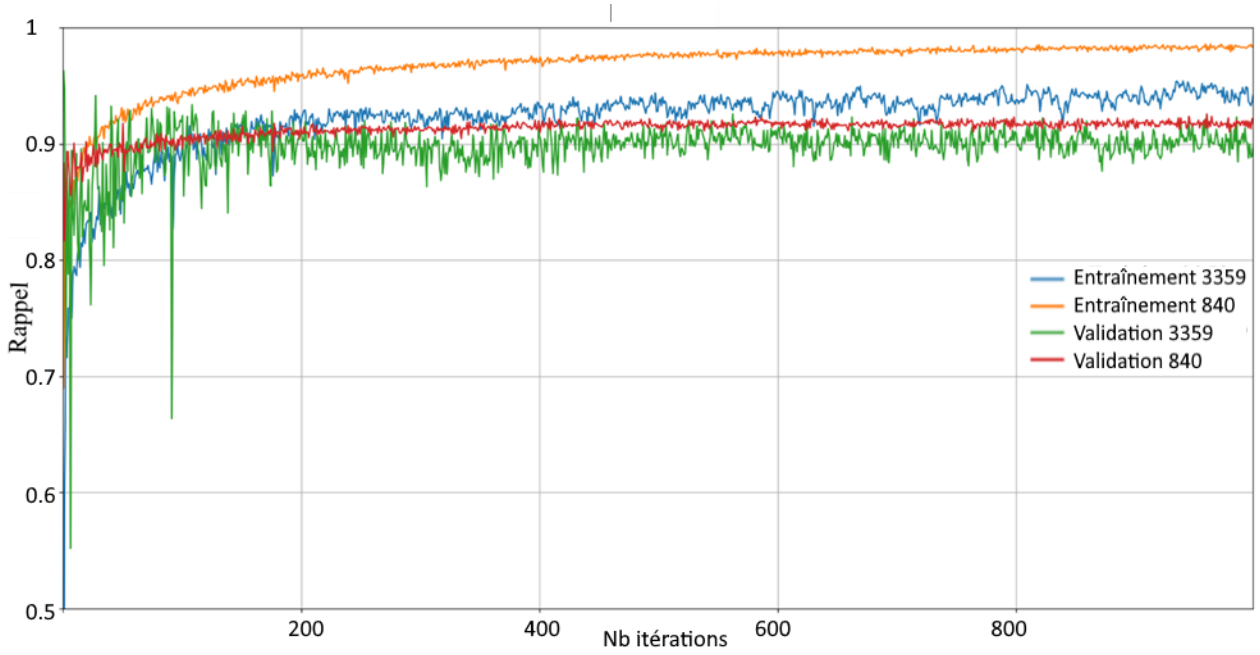


Figure 4.34 Courbe d'apprentissage : rappel

Nous observons à la Figure 4.34 une légère augmentation du rappel sur l'ensemble de validation ainsi qu'une diminution du bruit, mais nous observons surtout une nette augmentation sur l'ensemble d'entraînement du modèle de 840 caractéristiques. Bien que l'écart entre les courbes d'entraînement soit grand, cela n'implique pas le même écart entre les courbes sur les ensembles de validation. Ainsi, nous en déduisons qu'il ne sert à rien d'approfondir l'apprentissage sur l'ensemble d'entraînement si nous n'observons pas d'amélioration sur l'ensemble de validation. À la lumière de ces résultats, nous confirmons que 200 itérations sont suffisantes pour avoir les meilleurs résultats.

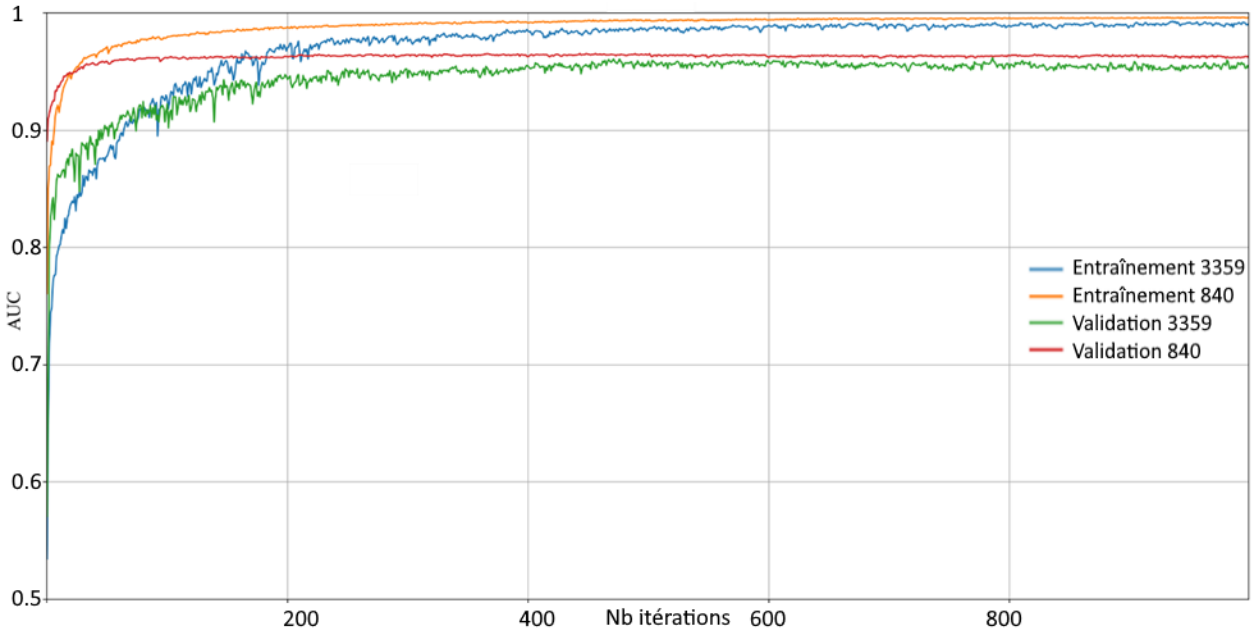


Figure 4.35 Courbe d'apprentissage : AUC

Nous observons à la Figure 4.35 une convergence plus rapide pour le modèle 840 en comparaison au modèle 3,359. L'AUC (qui, pour rappel, prend en compte les faux négatifs et les faux positifs) du modèle 840 est supérieure sur les deux ensembles à celle du modèle 3,359. Nous en déduisons que le modèle 840 apprend mieux que le modèle 3,359. Nous remarquons aussi qu'il apprend plus vite.

4.8 Ré-étiquetage

Cette section présente les résultats concernant le ré-étiquetage des bases de données.

4.8.1 Ré-étiquetage statique

Nous avons mentionné lors de la description d'obtention de la base de données Omnidroid, que le seuil Ɛ était égal à un. Ce seuil Ɛ représente le nombre d'antivirus qui détecte une application comme malveillante. Ainsi, un seuil Ɛ réglé à 1 signifie que si un seul antivirus parmi la soixantaine présente sur VirusTotal détecte une application comme malveillante, alors cette application sera étiquetée comme telle dans Omnidroid. Il est possible qu'à un temps t donné, un antivirus puisse se tromper faussant ainsi l'étiquetage. Il aurait peut-être été plus judicieux d'augmenter ce seuil Ɛ. Intuitivement, la probabilité que plusieurs antivirus se trompent à un temps donné t sur la même application est beaucoup plus basse qu'un seul antivirus ne se trompe permettant de mieux étiqueter. Afin de s'en assurer, il est nécessaire de faire une série d'entraînements afin de trouver le seuil Ɛ maximisant les métriques d'évaluation. C'est pourquoi nous traçons le nombre d'applications détectées par les antivirus en fonction du nombre d'antivirus. Il s'agit des 22 000 applications de la base de données Omnidroid.

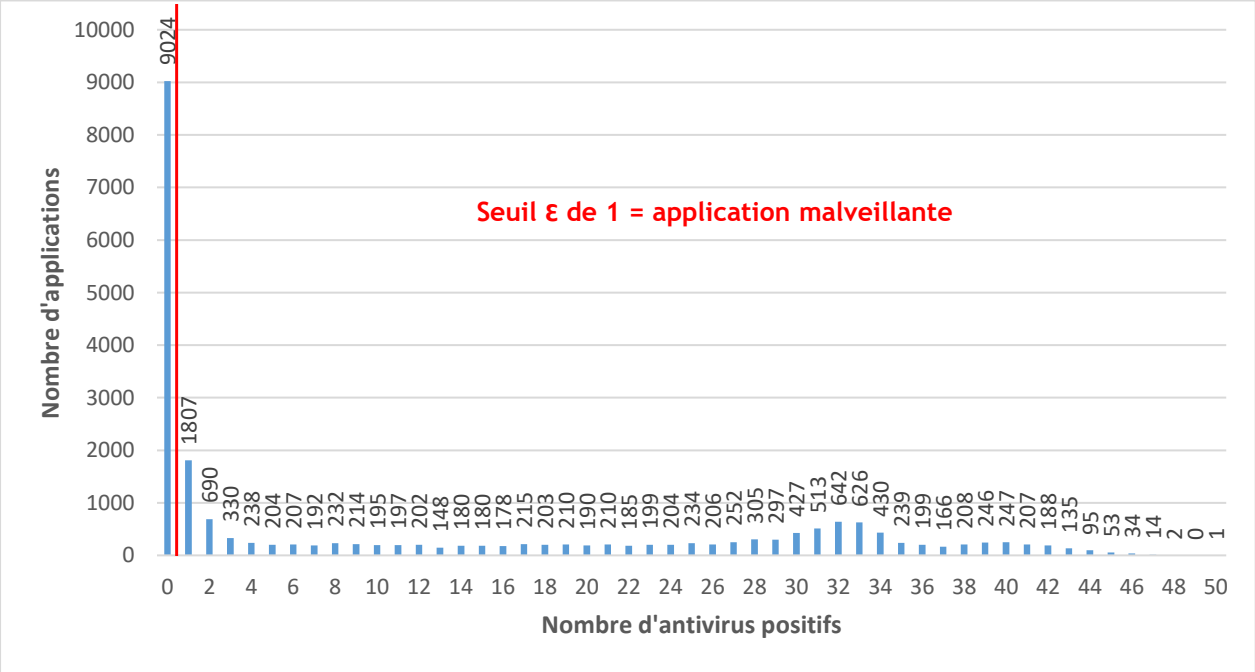


Figure 4.36 Nombre d'applications en fonction du nombre d'antivirus (en septembre 2019)

Les résultats à la Figure 4.36 sont valides à la date de septembre 2019. Nous avons eu recours au service VirusTotal [66] grâce une API académique qui comprend l'accès à 20 000 demandes d'API par jour afin d'obtenir un rapport pour chaque application (identifiée par son hash). Ainsi, deux jours ont été nécessaires à la collecte des rapports. D'après la date de modification des fichiers dans la base de données Omnidroid qui est du mois de juillet 2018, ces résultats datent donc d'un peu plus d'un an et demi après les résultats initiaux de A. Martín, R. Lara-Cabrera et D. Camacho [100]. Nous remarquons donc qu'un an et demi plus tard seulement 9,024 applications sont détectées comme bénignes avec un seuil Ɛ égal à 1 ce qui correspond à 0 antivirus sur l'axe des abscisses. Nous remarquons aussi que 1,807 applications sont classifiées malveillantes alors qu'un seul antivirus les a détectées comme malveillantes. Un seuil Ɛ égal à 2 aurait suffi pour les classifier comme bégnines. Or 1,807 représentent 8.2% des 22 000 applications, cela a un impact sur l'entraînement du réseau de neurones et donc sur les métriques du réseau de neurones. Dans le cas idéal, nous ajouterions ces 8,2% à notre justesse statique pour obtenir 97.8% et pour obtenir 87.8 % pour les résultats dynamiques. Nous avons effectué les graphiques suivants avec l'aide des 22 000 rapports d'applications datant de septembre 2019 qui ont été collectés à l'aide d'un script python et d'une clé académique VirusTotal. Nous avons réétiqueté Omnidroid pour les entraînements statiques en faisant **varier le seuil** Ɛ de 1 à 4 pour les rapports de 2018 et de 1 à 10 pour les rapports de 2019. Les résultats sont consignés sur la Figure 4.37. **Le nombre d'itérations est de 50 itérations.**

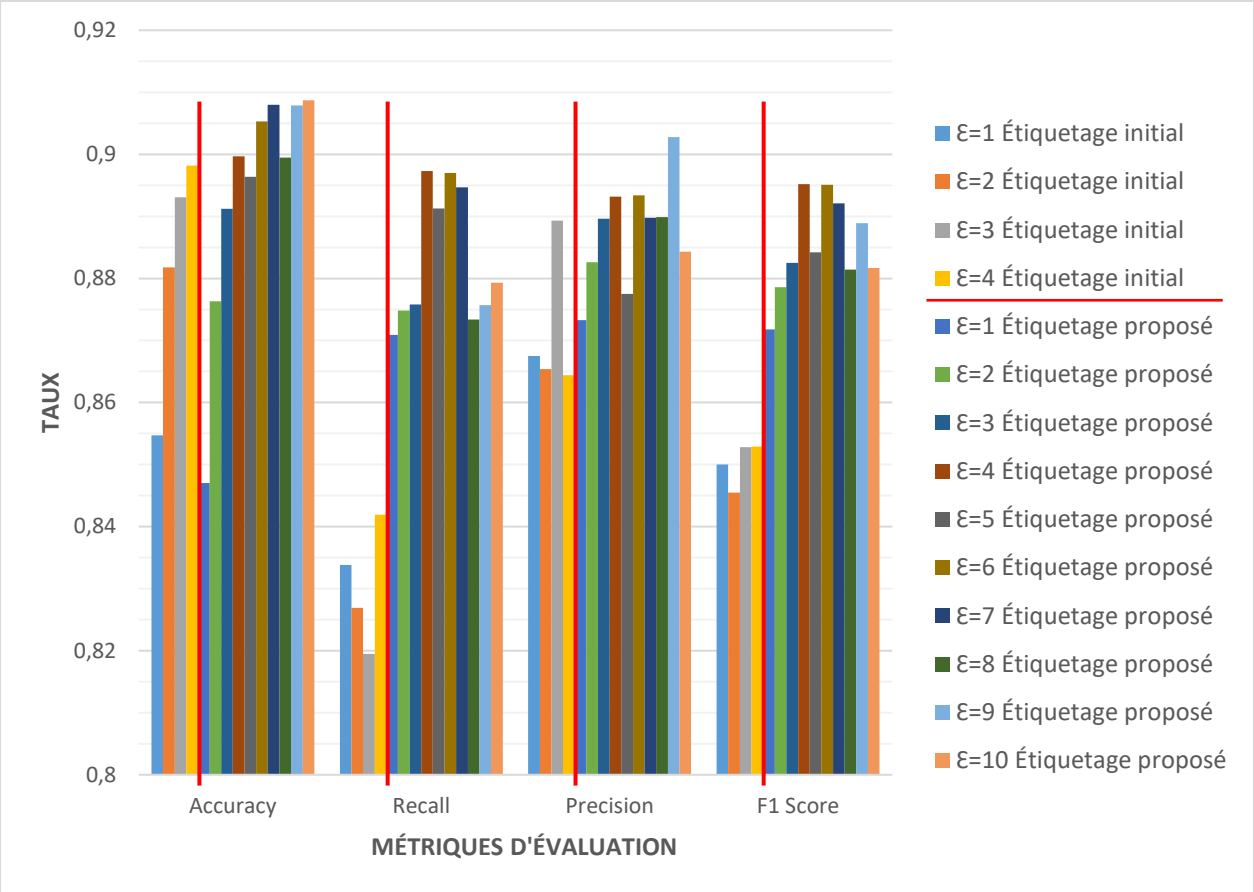


Figure 4.37 Impact du ré-étiquetage statique

En effet, nous remarquons à la Figure 4.37 bien une amélioration des résultats à la suite du ré-étiquetage. Nous en déduisons qu'il y a eu de nouvelles analyses des antivirus parmi VirusTotal en un an et demi et que cela a un impact sur la détection d'applications malveillantes.

4.8.2 Ré-étiquetage dynamique

Nous avons réétiqueté également Omnidroid pour les entraînements dynamiques, mais cette fois-ci en faisant **varier le seuil** Ɛ de 1 à 6 pour les rapports. Les résultats sont consignés sur la Figure 4.38.

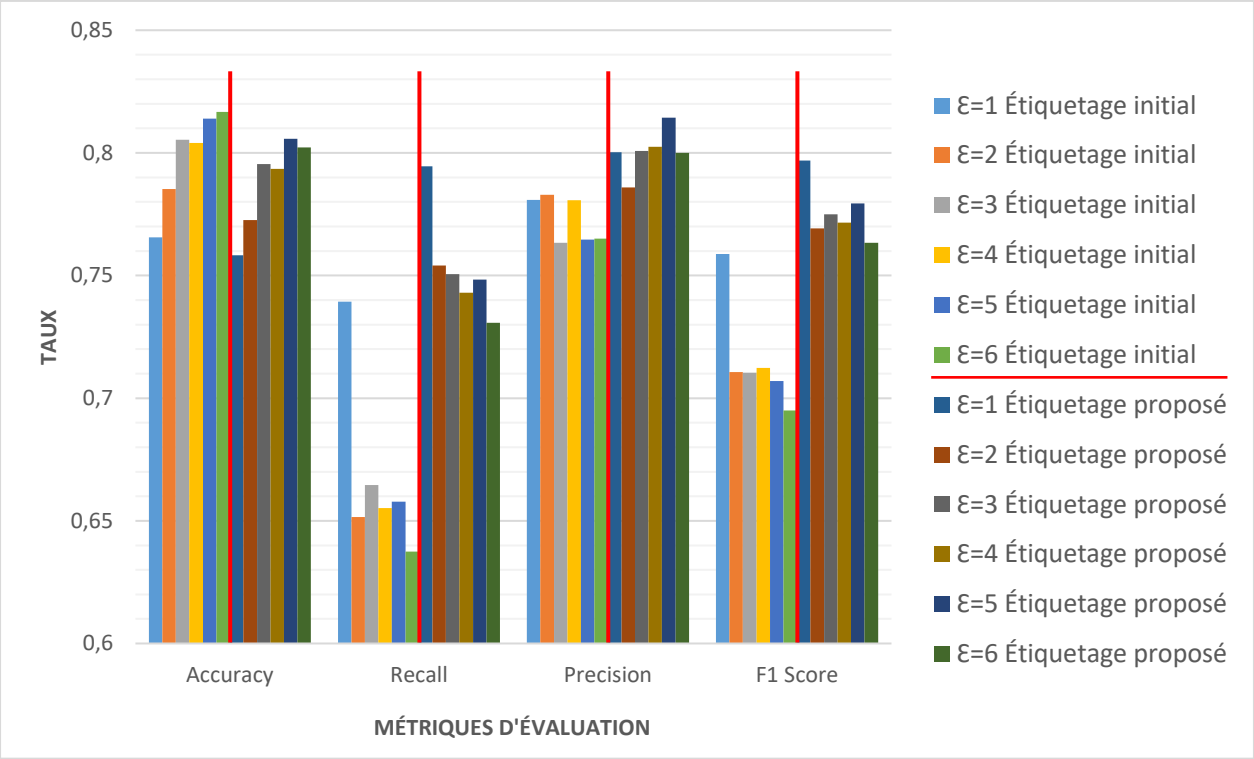


Figure 4.38 Impact du ré-étiquetage dynamique

Comme pour le ré-étiquetage statique, nous remarquons à la Figure 4.38 une amélioration des résultats à la suite du ré-étiquetage dynamique. Le rappel est la métrique d'évaluation ayant le plus augmenté. C'est-à-dire que le ré-étiquetage a permis de pouvoir détecter plus d'applications malveillantes qui étaient auparavant non détectés.

4.9 Résultats finaux des modèles

À la suite du ré-étiquetage, nous avons fixé le nombre d'itérations à la valeur qui nous permettent d'obtenir les meilleurs scores pour chaque type de réseau de neurones. Le Tableau 4.8 montre les valeurs finales des paramètres d'entraînements.

Tableau 4.8 Paramètres pour les entraînements finaux statiques, dynamiques et hybrides

Nombre d'itérations	Statique 200	Dynamique 250	Hybride 200
Taux d'apprentissage	0.002		
Taux d'abandon	0.3		
Taille de batch	512		
Nombre de caractéristiques	Statiques : 840	Dynamiques : 2800	
Nombres de neurones sur L1	Statiques : 280	Dynamiques : 933	
Fonction de perte	Binary Crossentropy		
Fonction d'activation L1	Relu		
Fonction d'activation L2	Sigmoïde		

4.9.1 Résultats du réseau statique de neurones

La réduction des caractéristiques a été réalisée avec la Corrélation de Pearson. Le Tableau 3.9 présente les résultats du réseau de neurones statiques sur **l'ensemble de test**.

Tableau 4.9 Comparaison du réseau statique proposé avec [100]

	Justesse	Précision	#Caractéristiques
[100]	89.3%	89.3%	25,999
Modèle proposé	92.9%	92.1%	840
Gain	3.6%	2.8%	96.8%

La justesse et la précision sont les seules métriques auxquelles nous pouvons nous comparer. Toutes les métriques du meilleur réseau statique sur **l'ensemble de test** sont dans le Tableau 4.10.

Tableau 4.10 Métriques d'évaluation du réseau statique proposé

Perte	0.3058	
Justesse	92.9%	
Rappel	92.5%	
Précision	92.1%	
VN	1,662	50.4%
VP	1,398	42.4%
FP	123	3.6%
FN	117	3.4%
Total	3,300	100%
AUC	96.9%	
F1 Score	92.3%	

4.9.2 Résultats du réseau dynamique de neurones

La réduction des caractéristiques a été réalisée avec une méthode manuelle consistant à enlever les caractéristiques vides ou dont la somme est égale à 1. Le Tableau 3.11 présente les résultats du réseau de neurones dynamiques sur **l'ensemble de test**.

Tableau 4.11 Comparaison du réseau dynamique proposé avec [100]

	Justesse	Précision	#Caractéristiques
[100]	78.6%	78.6%	5,932
Modèle proposé	81.1%	83.4%	3,722
Gain	2.5%	4.8%	37.3%

De la même manière que pour le réseau statique, les métriques d'évaluation du réseau dynamique proposé sur **l'ensemble de test** sont présentées dans le Tableau 4.12.

Tableau 4.12 Métriques d'évaluation du réseau dynamique proposé

Perte	0.9033	
Justesse	81.1%	
Rappel	73.4%	
Précision	83.4%	
VN	1,566	47.5%
VP	1,110	33.6%
FP	221	6.7%
FN	403	12.2%
Total	3,300	100 %
AUC	87.1%	
F1 Score	80%	

4.9.3 Résultats du réseau hybride de neurones

Le Tableau 3.13 présente les résultats du réseau de neurones hybride sur **l'ensemble de test**.

Tableau 4.13 Comparaison du réseau hybride proposé avec [100]

	Justesse	Précision	#Caractéristiques
[100]	89.7%	89.7%	25,999 + 5,932
Modèle proposé	91.1%	91%	3,359 + 3,722
Gain	1.4%	1.3%	77.8%

De la même manière que pour les méthodes précédentes les métriques d'évaluation du réseau hybride proposé sur **l'ensemble de test** sont présentées dans le Tableau 4.14.

Tableau 4.14 Métriques d'évaluation du réseau hybride proposé

Perte	2.4066	
Justesse	91.1%	
Rappel	89.6%	
Précision	91%	
VN	1,647	50%
VP	1,361	41.2%
FP	134	4.1%
FN	158	4.8%
Total	3,300	100 %
AUC	94.1%	
F1 Score	90.3%	

4.9.4 Comparaison des résultats des méthodes statique/dynamique/hybride

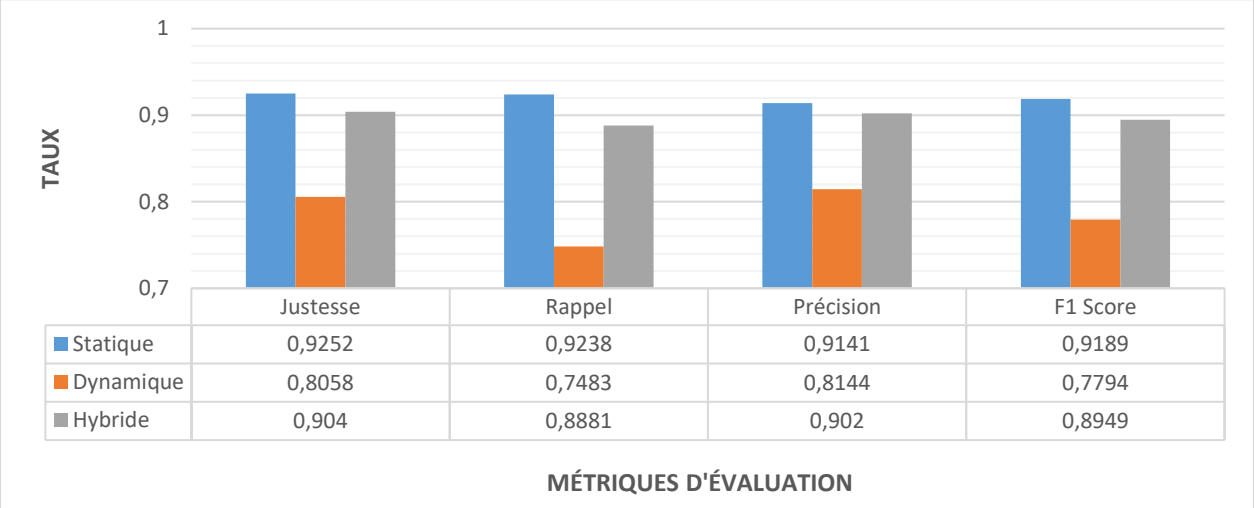


Figure 4.39 Comparaison des résultats des différentes méthodes

Les résultats affichés à la Figure 4.39 sont les moyennes d'une série de dix entraînements pour chaque méthode. Les meilleurs réseaux de neurones ont été sélectionnés parmi ces séries. Les résultats statiques surpassent les résultats dynamiques, mais aussi les résultats hybrides. Tout d'abord, il faut remarquer que les résultats statiques sont, sans techniques d'apprentissages, meilleurs que les résultats dynamiques. Ainsi, il n'est pas étonnant qu'en rajoutant des techniques comme la sélection de caractéristiques ou la régularisation qui permet d'augmenter tous les résultats, nous gardons des meilleurs résultats statiques. Maintenant, nous pensions obtenir de meilleurs résultats hybrides puisqu'elles combinent les deux méthodes complémentaires, mais en réalité ce n'est pas le cas. Bien que les méthodes statique et dynamique soient théoriquement complémentaires, nous pensons que ce n'est pas le cas et que les deux méthodes détectent en fait les mêmes applications malveillantes **sur cette base de données**. Ainsi, la méthode dynamique viendrait perturber l'apprentissage statique. En effet, nous n'aurions pas eu ces résultats si les méthodes étaient effectivement complémentaires (c'est-à-dire qu'elles détectent des applications malveillantes différentes).

Une autre explication provient des outils d'extraction des caractéristiques. Plutôt aisé de manière statique, il s'agit d'une étape longue et compliqué pour la méthode dynamique et il se peut qu'il y ait des travaux de recherches supplémentaires à effectuer.

CHAPITRE 5 CONCLUSION

Après avoir présenté l'ensemble des résultats des réseaux de neurones pour la détection de logiciels malveillants ainsi que leur implémentation BrainShield, il nous revient maintenant de conclure le mémoire. La première section est une synthèse des travaux. La deuxième section de cette conclusion présente les limitations de nos travaux. Enfin, la troisième section se concentre sur les travaux futurs.

5.1 Synthèse des travaux

Dans un premier temps, nous avons fait la revue de littérature dans le domaine de la détection de logiciels malveillants sur Android. Ce parcours comprend tout d'abord les solutions proposées par le milieu industriel présent sur le Google Play Store. Ensuite, nous avons vu plusieurs outils disponibles en libre accès sur GitHub, ainsi que plusieurs articles du milieu académique. Ces derniers proposent des outils et des modèles statiques, dynamiques ou bien hybrides. Cette vue d'ensemble nous a permis de nous diriger vers le modèle proposé.

Dans un second temps, nous avons proposé un modèle de détection de logiciels malveillants pour terminaux mobiles. Ce modèle est composé de trois modules. Le premier module est constitué de la base de données Omnidroid, choisie pour entraîner les réseaux de neurones. C'est une base de données hybride composé de caractéristiques statiques et dynamiques qui comprend 22 000 applications. Le deuxième module est constitué des techniques d'apprentissage que nous avons mis en place, ainsi que les métriques permettant d'évaluer les réseaux de neurones proposés. Le troisième module est l'architecture client/serveur du modèle proposé.

Nous avons alors implémenté les réseaux de neurones dans un prototype d'application se nommant BrainShield. Cette solution complète propose de téléverser les applications du terminal Android sur le serveur, afin d'y réaliser l'extraction des caractéristiques avec AndroPyTool ainsi que la prédiction. Le serveur renvoie à l'application mobile l'ensemble des prédictions au format json. Ce format laisse la possibilité de développer davantage de solutions, comme une application web. De plus, nous avons présenté de nombreux résultats concernant le paramétrage des hyperparamètres nécessaires lors de l'entraînement des réseaux de neurones. Ces résultats permettent de saisir

l'impact de chaque paramètre sur l'entraînement des réseaux de neurones et peuvent être utiles aux pairs.

Nous avons proposé aussi deux méthodes de détection d'applications malveillantes. Ces deux méthodes s'appuient sur des réseaux de neurones entièrement connectés entraînés par les librairies Tensorflow/Keras. La première méthode est statique et consiste en un premier réseau **statique** atteignant une justesse de **92.9%** et une précision de **91.1%,** entraîné sur **840** caractéristiques statiques (static features). La deuxième méthode de détection consiste en un deuxième réseau de neurones dynamique atteignant une justesse (accuracy) de **81.1%** et une précision (precision) de **83.4%,** entrainé sur **3,722** caractéristiques dynamiques (dynamic features). Nous avons proposé également un réseau de neurones hybride atteignant une justesse de **91.1%** et une précision de **91.0%,** entrainé sur **7081** caractéristiques (3,359 statiques + 3,722 dynamiques). Des techniques de sélection de caractéristiques sont utilisées, comme la corrélation de Pearson ainsi que des forêts d'arbres décisionnels et une méthode manuelle. En outre, nous avons montré que 22,636 caractéristiques statiques ainsi que 2,210 caractéristiques dynamiques de la base de données d'Omnidroid sont vides, soit un total de 24,846 sur 31,931 (77.81%).

5.2 Limitations

La première limitation est que nous nous sommes cantonnés au système d'exploitation Android. En effet, il s'agit du système d'exploitation le plus utilisé dans le monde avec en particulier plus de 80% du marché mobile [122]. Il s'agit donc d'une cible prometteuse pour les programmeurs de logiciels malveillants.

Ensuite, les résultats présentés sont valables à la date des échantillons de la base de données. Nous rappelons que la collection des applications malveillantes présentes dans Omnidroid date de 2012 à 2018. Nous avons bien vu l'impact de la date des rapports VirusTotal sur les métriques d'évaluation. On peut alors se demander ce qu'il adviendra de la justesse et de la précision des réseaux de neurones dans plusieurs années. En effet, les applications pourraient évoluer ainsi que leurs analyses présentes sur le service VirusTotal qui nous a permis d'étiqueter les applications Android.

De plus, nous n'avons pas pu développer nos propres outils de détection. Il s'agit d'un travail conséquent et fastidieux. Il se pourrait que les applications aient trop évolué pour pouvoir les faire tourner sur un émulateur d'API 16 qui est celui de DroidBox [90], l'outil d'extraction des caractéristiques dynamiques. Une API de 16 qui correspond à une date officielle de sortie d'été 2012, nous permet de faire rouler des applications relativement anciennes pour la période de l'été 2020. Néanmoins, il est possible que les développeurs d'applications estiment que cette API soit trop ancienne et mettent une API minimale de 19 par exemple. Ainsi, il serait impossible de pouvoir faire rouler leur application sur l'émulateur. Cependant, il peut s'agir d'un avantage puisque nous pouvons analyser des applications anciennes. Par ailleurs, l'extraction des caractéristiques dynamiques peut se compliquer dans les applications nécessitant une identification (applications de type Facebook, Instagram, WhatsApp, Messenger…). En effet, l'outil d'extraction des caractéristiques automatique serait bloqué à l'affichage de l'identification et ne pourrait explorer l'ensemble des fonctionnalités de l'application. Pour rappel, l'émulateur dispose d'une fonctionnalité appelée Monkey qui permet de cliquer de manière aléatoire sur l'écran afin de simuler les clics de l'utilisateur. Les caractéristiques extraites seraient, soit inexistantes, soit en quantité trop peu représentative pour pouvoir faire une prédiction. Il s'agit là d'une limitation intrinsèque à l'analyse dynamique.

5.3 Travaux futurs

Pour répondre à la première limitation de nos travaux, l'exercice pourrait être généralisé à d'autres systèmes d'exploitation comme iOS. En effet, de nombreux autres terminaux mobiles s'exécutent sur le système d'exploitation iOS et puisque iOS représente 20% du marché mobile [122]. Il faudrait à ce moment-là développer de nouveaux outils d'extraction de caractéristiques statiques et dynamiques. Grâce à ces nouveaux outils, nous pourrions construire une nouvelle base de données que l'on viendrait étiqueter à l'aide du même service utilisé : VirusTotal. En outre, toutes les techniques d'apprentissages, les métriques d'évaluation et le paramétrage des hyperparamètres pourront être réutilisés avec ce nouveau système d'exploitation.

Il est aussi nécessaire de mettre à jour la base de données avec les applications malveillantes les plus récentes ainsi que les analyses les plus récentes, de VirusTotal ou d'autres services de détection de logiciels malveillants pour étiqueter ces applications. La mise à jour de la base de

données pourrait être effectuée manuellement dans quelques années par des pairs ou par moi-même lors d'un éventuel doctorat. Nous pourrions aussi envisager le développement d'un outil d'automatisation de mise à jour des réseaux neuronaux.

Pour répondre aux limitations concernant l'extraction des caractéristiques dynamiques, une des solutions serait de développer un outil d'extraction des caractéristiques ayant une déclinaison pour chaque API afin de faire tourner toutes les applications. Une solution pour contrecarrer le manque de caractéristiques dynamiques pour des applications nécessitant de se connecter serait de pouvoir passer en mode manuel. En effet, une intervention humaine pourrait pallier cette limitation. Il serait alors nécessaire de développer un outil comprenant une option automatique et une option manuelle. Nous verrons alors apparaître une autre problématique : l'analyse dynamique d'une grande quantité d'applications. En effet, dès lors que l'humain doit intervenir, il devient difficile d'imaginer une analyse manuelle lorsque des milliers d'applications voire des millions d'applications sont à analyser.

RÉFÉRENCES

[1] S. R. Department, "Nombre d'utilisateurs de smartphone dans le monde entre 2014 et 2020(en milliards)," 2016. [En ligne]. Disponible: https://fr.statista.com/statistiques/574542/utilisateurs-de-smartphone-dans-le-monde--2019/

[2] I. D. C. (IDC), "Worldwide Tablet Shipments Return to Growth in Q3 2019, Fueled by New Product Launches," 2020. [En ligne]. Disponible: https://www.idc.com/getdoc.jsp?containerId=prUS45620419

[3] I. D. C. (IDC), "Worldwide Wearables Shipments Surge 94.6% in 3Q 2019 Led by Expanding Hearables Market," 2019. [En ligne]. Disponible: https://www.idc.com/getdoc.jsp?containerId=prUS45712619

[4] H. Torii, "More Than Half of All Households in Japan, US and Europe Will Have Smart TVs by 2019," 2016. [En ligne]. Disponible: https://technology.informa.com/571765/more-than-half-of-all-households-in-japan-us-and-europe-will-have-smart-tvs-by-2019

[5] MarketsAndMarkets, "Connected Car Market," 2019. [En ligne]. Disponible: https://www.marketsandmarkets.com/Market-Reports/connected-car-market-102580117.html

[6] N. Woolf. (2016) DDoS attack that disrupted internet was largest of its kind in history, experts say. [En ligne]. Disponible: https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet

[7] J. Fruhlinger. (2018) What is WannaCry ransomware, how does it infect, and who was responsible? [En ligne]. Disponible: https://www.csoonline.com/article/3227906/what-is-wannacry-ransomware-how-does-it-infect-and-who-was-responsible.html

[8] MalwareBytes. (2020) Malware. [En ligne]. Disponible: https://fr.malwarebytes.com/malware/

[9] W. Tounsi, *Cybervigilance et confiance numérique: La cybersécurité à l'ère du Cloud et des objets connectés. ISTE Group*, 2019.

[10] E. F. Ian Witten, Mark Hall and Chris Pal, "Data Mining - Practical Machine Learning Tools and Techniques," 2016. [En ligne]. Disponible: https://www.cs.waikato.ac.nz/ml/weka/book.html

[11] G. Inc. (2020) Machine Learning Crash Course. [En ligne]. Disponible: https://developers.google.com/machine-learning/crash-course/classification/

[12] Microsoft, "Évaluation des performances d'un modèle dans Azure Machine Learning Studio (classique)," 2020. [En ligne]. Disponible: https://docs.microsoft.com/fr-fr/azure/machine-learning/studio/evaluate-model-performance

[13] G. Inc, "Android web page," 2020. [En ligne]. Disponible: https://www.android.com/intl/fr_fr/

[14] Android Open Source Project. [En ligne]. Disponible: https://source.android.com

[15] FileInfo, ".APK File Extension," 2020. [En ligne]. Disponible: https://fileinfo.com/extension/apk

[16] L. R. Clubic, "Logiciels malveillants contre antivirus : l'histoire d'une lutte sans fin," 2018. [En ligne]. Disponible: https://www.clubic.com/antivirus-securite-informatique/logiciel-antivirus/article-844896-1-antivirus-malwares-histoire-poursuite.html

[17] IDC. (2019) IDC Forecasts Worldwide Smartphone Market Will Face Another Challenging Year in 2019 with a Return to Growth on the Horizon. [En ligne]. Disponible: https://www.idc.com/getdoc.jsp?containerId=prUS45115119

[18] Statcounter. (2019) Mobile Operating System Market Share Worldwide. [En ligne]. Disponible: http://gs.statcounter.com/os-market-share/mobile/worldwide

[19] Kaspersky. Smartphone Security: Android vs. iPhone vs. BlackBerry vs. Windows Phone - Top Mobile Threats. [En ligne]. Disponible: https://usa.kaspersky.com/resource-center/threats/android-vs-iphone-mobile-security

[20] I. Mohamed et D. Patel, "Android vs iOS security: A comparative study," communication présentée à 2015 12th International Conference on Information Technology-New Generations, 2015, p. 725-730.

[21] A. Martín *et al.*, "ADROIT: Android malware detection using meta-information," communication présentée à 2016 IEEE Symposium Series on Computational Intelligence (SSCI), 2016, p. 1-8.

[22] S. Feldman, D. Stadther et B. Wang, "Manilyzer: automated android malware detection through manifest analysis," communication présentée à 2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems, 2014, p. 767-772.

[23] A. Sadeghi *et al.*, "A temporal permission analysis and enforcement framework for android," communication présentée à 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018, p. 846-857.

[24] V. Afonso *et al.*, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," communication présentée à The Network and Distributed System Security Symposium, 2016, p. 1-15.

[25] M. Sun et G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," communication présentée à Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks, 2014, p. 165-176.

[26] S. Alam *et al.*, "DroidNative: Automating and optimizing detection of Android native code malware variants," vol. 65, p. 230-246, 2017.

[27] L. Glanz *et al.*, "CodeMatch: obfuscation won't conceal your repackaged app," communication présentée à Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, p. 638-648.

[28] J. Akram *et al.*, "Droidcc: A scalable clone detection approach for android applications to detect similarity at source code level," communication présentée à 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 2018, p. 100-105.

[29] X. Su *et al.*, "AndroGenerator: An automated and configurable android app network traffic generation system," vol. 8, n°. 18, p. 4273-4288, 2015.

[30] Q. Wang *et al.*, "I know what you did on your smartphone: Inferring app usage over encrypted data traffic," communication présentée à 2015 IEEE Conference on Communications and Network Security (CNS), 2015, p. 433-441.

[31] M. Li *et al.*, "LibD: scalable and precise third-party library detection in android markets," communication présentée à 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, p. 335-346.

[32] D. Octeau *et al.*, "Combining static analysis with probabilistic models to enable market-scale android inter-component analysis," communication présentée à ACM SIGPLAN Notices, 2016, p. 469-484.

[33] Y. Chen *et al.*, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," communication présentée à 2018 Network and Distributed System Security Symposium (NDSS'18), 2018.

[34] (2016) Bulletin de sécurité Android. [En ligne]. Disponible: https://source.android.com/security/bulletin/2016-12-01

[35] D.-J. Wu *et al.*, "Droidmat: Android malware detection through manifest and api calls tracing," communication présentée à 2012 Seventh Asia Joint Conference on Information Security, 2012, p. 62-69.

[36] Y. Aafer, W. Du et H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," communication présentée à International conference on security and privacy in communication systems, 2013, p. 86-103.

[37] C. Yang *et al.*, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," communication présentée à European symposium on research in computer security, 2014, p. 163-182.

[38] M. Zhang *et al.*, "Semantics-aware android malware classification using weighted contextual api dependency graphs," communication présentée à Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, 2014, p. 1105-1116.

[39] M. Zheng, M. Sun et J. C. Lui, "Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware," communication présentée à 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 2013, p. 163-171.

[40] G. Suarez-Tangil *et al.*, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," vol. 41, n°. 4, p. 1104-1117, 2014.

[41] L. Deshotels, V. Notani et A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," communication présentée à Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014, 2014, p. 3.

[42] JesusFreke. (2019) Smali/backsmali. [En ligne]. Disponible: https://github.com/JesusFreke/smali

[43] C. Tumbleson et R. Wiśniewski. (2019) Apktool. [En ligne]. Disponible: https://ibotpeaches.github.io/Apktool/

[44] pxb1988. (2018) dex2jar. [En ligne]. Disponible: https://github.com/pxb1988/dex2jar

[45] (2019) jd-gui. [En ligne]. Disponible: https://github.com/java-decompiler/jd-gui/

[46] Google Inc. (2019) Rapport de transparence Google. [En ligne]. Disponible: https://transparencyreport.google.com/android-security/overview

[47] B. d. d. Android. (2018) How we fought bad apps and malicious developers in 2017. [En ligne]. Disponible: https://android-developers.googleblog.com/2018/01/how-we-fought-bad-apps-and-malicious.html

[48] G. M. Blog. (2012) Annonce Google Bouncer. [En ligne]. Disponible: https://googlemobile.blogspot.com/2012/02/android-and-security.html

[49] J. O. e. C. Miller. (2012) Dissecting Android's Bouncer. [En ligne]. Disponible: https://duo.com/blog/dissecting-androids-bouncer

[50] O. Hou. (2012) A Look at Google Bouncer. [En ligne]. Disponible: https://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/

[51] L. Stefanko. (2015) Android trojan drops in, despite Google's Bouncer. [En ligne]. Disponible: https://www.welivesecurity.com/2015/09/22/android-trojan-drops-in-despite-googles-bouncer/

[52] S. Young. (2018) Google Play removes driving apps that installed Android malware. [En ligne]. Disponible: https://www.consumeraffairs.com/news/google-play-removes-driving-apps-that-installed-android-malware-112118.html

[53] E. Then. (2019) BeiTaAd adware discovered in 238 Google Play Store apps. [En ligne]. Disponible: https://www.slashgear.com/beitaad-adware-discovered-in-238-google-play-store-apps-05579281/

[54] B. Businessweek. (2018) Espionnage industriel – la chine prise la main dans le sac. [En ligne]. Disponible: http://plus.lapresse.ca/screens/b4f862e1-3f02-4b9f-ad08-5b0ff5cbf1df__7C___0.html

[55] Kaspersky. (2016) Triada: organized crime on Android. [En ligne]. Disponible: https://www.kaspersky.com/blog/triada-trojan/11481/

[56] G. S. Blog. (2019) PHA Family Highlights: Triada. [En ligne]. Disponible: https://security.googleblog.com/2019/06/pha-family-highlights-triada.html

[57] Android. (2019) Google Play Protect. [En ligne]. Disponible: https://www.android.com/intl/en_ca/play-protect/

[58] A. Bhatia. (2019) Android Security Awesome. [En ligne]. Disponible: https://github.com/ashishb/android-security-awesome

[59] A. Sadeghi *et al.*, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," vol. 43, n°. 6, p. 492-530, 2016.

[60] M. Talal *et al.*, "Comprehensive review and analysis of anti-malware apps for smartphones," p. 1-53, 2019.

[61] A. Souri et R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," vol. 8, n°. 1, p. 3, 2018.

[62] Y. Zhou et X. Jiang, "Dissecting android malware: Characterization and evolution," communication présentée à 2012 IEEE symposium on security and privacy, 2012, p. 95-109.

[63] D. Arp *et al.*, "Drebin: Effective and explainable detection of android malware in your pocket," communication présentée à Ndss, 2014, p. 23-26.

[64] Contagio. [En ligne]. Disponible: https://contagiodump.blogspot.com/

[65] F. Wei *et al.*, "Deep ground truth analysis of current android malware," communication présentée à International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2017, p. 252-276.

[66] VirusTotal. [En ligne]. Disponible: https://support.virustotal.com

[67] Koodous. [En ligne]. Disponible: https://koodous.com/

[68] K. Allix *et al.*, "Androzoo: Collecting millions of android apps for the research community," communication présentée à 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), 2016, p. 468-471.

[69] F.-X. Geiger et I. Malavolta, "Datasets of Android Applications: a Literature Review," 2018.

[70] A. Martín, Lara-Cabrera, R., & Camacho, D., "OmniDroid: A comprehensive benchmark dataset of dynamic and static features from Android applications," 2018. [En ligne]. Disponible: https://aida.ii.uam.es/datasets/

[71] SandDroid. [En ligne]. Disponible: http://sanddroid.xjtu.edu.cn:8080/#home

[72] A. Fournier, "Détection de programmes malveillants dédiée aux appareils mobiles," École Polytechnique de Montréal, 2018.

[73] E. Frank, M. A. Hall et I. H. Witten. (2016) The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques". [En ligne]. Disponible: https://www.cs.waikato.ac.nz/ml/weka/index.html

[74] M. Ahmadi, A. Sotgiu et G. Giacinto, "IntelliAV: Building an Effective On-Device Android Malware Detector," 2018.

[75] W. Enck, M. Ongtang et P. McDaniel, "On lightweight mobile phone application certification," communication présentée à Proceedings of the 16th ACM conference on Computer and communications security, 2009, p. 235-245.

[76] E. Mariconti *et al.*, "Mamadroid: Detecting android malware by building markov chains of behavioral models," 2016.

[77] G. Suarez-Tangil *et al.*, "Droidsieve: Fast and accurate classification of obfuscated android malware," communication présentée à Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, 2017, p. 309-320.

[78] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," communication présentée à Acm Sigplan Notices, 2014, p. 259-269.

[79] J. Garcia, M. Hammad et S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," vol. 26, nº. 3, p. 11, 2018.

[80] Z. Ma *et al.*, "A Combination Method for Android Malware Detection Based on Control Flow Graphs and Machine Learning Algorithms," vol. 7, p. 21235-21245, 2019.

[81] E. B. Karbab *et al.*, "MalDozer: Automatic framework for android malware detection using deep learning," vol. 24, p. S48-S59, 2018.

[82] A. Desnos, "Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!)," éd, 2015.

[83] W. Enck *et al.*, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," vol. 32, nº. 2, p. 5, 2014.

[84] L. K. Yan et H. Yin, "DroidScope: Seamlessly Reconstructing the {OS} and Dalvik Semantic Views for Dynamic Android Malware Analysis," communication présentée à Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12), 2012, p. 569-584.

[85] V. Rastogi, Y. Chen et W. Enck, "AppsPlayground: automatic security analysis of smartphone applications," communication présentée à Proceedings of the third ACM conference on Data and application security and privacy, 2013, p. 209-220.

[86] K. Tam *et al.*, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," communication présentée à Ndss, 2015.

[87] Z.-G. Chen *et al.*, "Automatic ransomware detection and analysis based on dynamic API calls flow graph," communication présentée à Proceedings of the International Conference on Research in Adaptive and Convergent Systems, 2017, p. 196-201.

[88] S. K. Dash *et al.*, "Droidscribe: Classifying android malware based on runtime behavior," communication présentée à 2016 IEEE Security and Privacy Workshops (SPW), 2016, p. 252-261.

[89] M. K. Alzaylaee, S. Y. Yerima et S. Sezer, "Emulator vs real phone: Android malware detection using machine learning," communication présentée à Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics, 2017, p. 65-72.

[90] H. Cai *et al.*, "Droidcat: Effective android malware detection and categorization via app-level profiling," vol. 14, nº. 6, p. 1455-1470, 2018.

[91] A. Desnos et P. Lantz, "Droidbox: An android application sandbox for dynamic analysis," 2011.

[92] A. Developer, "Shrink, obfuscate, and optimize your app," 2020. [En ligne]. Disponible: https://developer.android.com/studio/build/shrink-code#obfuscate

[93] A. Developers, "Cryptography," 2020. [En ligne]. Disponible: https://developer.android.com/guide/topics/security/cryptography

[94] A. Developers, "Native Code," 2020. [En ligne]. Disponible: https://developer.android.com/ndk

[95] A.-A. n. d. l. s. d. s. d'information, "Glossaire - Bombe logique," 2020. [En ligne]. Disponible: https://www.ssi.gouv.fr/particulier/glossaire/b/

[96] T. France, "Les pirates informatiques personnalisent les URL malveillantes pour éviter leur détection," 2019. [En ligne]. Disponible: https://www.titanhq.fr/blog/pirates-informatiques-personnalisent-url-malveillantes-eviter-detection/

[97] A. Saracino *et al.*, "Madam: Effective and efficient behavior-based android malware detection and prevention," vol. 15, nº. 1, p. 83-97, 2016.

[98] M. K. Alzaylaee, S. Y. Yerima et S. Sezer, "DynaLog: An automated dynamic analysis framework for characterizing android applications," communication présentée à 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security), 2016, p. 1-8.

[99] S. Arshad *et al.*, "SAMADroid: a novel 3-level hybrid malware detection model for android operating system," vol. 6, p. 4321-4339, 2018.

[100] A. Martín, R. Lara-Cabrera et D. Camacho, "Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset," vol. 52, p. 128-142, 2019.

[101] Strace. [En ligne]. Disponible: https://source.android.com/devices/tech/debug/strace

[102] A. Abraham *et al.*, "GroddDroid: a gorilla for triggering malicious behaviors," communication présentée à 2015 10th international conference on malicious and unwanted software (MALWARE), 2015, p. 119-127.

[103] S. Arshad, Munam A. Shah, Abdul Wahid, Amjad Mehmood, Houbing Song, and Hongnian Yu, "SAMADroid: a novel 3-level hybrid malware detection model for android operating system," vol. IEEE Access, 6, p. 4321-4339, 2018.

[104] A. Developer, "References," 2020. [En ligne]. Disponible: https://developer.android.com/reference

[105] S. Rasthofer, S. Arzt et E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," communication présentée à NDSS, 2014, p. 1125.

[106] G. Inc, "Representation: Feature Engineering," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/crash-course/representation/feature-engineering

[107] G. Inc, "Representation: Qualities of Good Features," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/crash-course/representation/qualities-of-good-features

[108] J. Brownlee, "Feature Selection For Machine Learning in Python," 2019. [En ligne]. Disponible: https://machinelearningmastery.com/feature-selection-machine-learning-python/

[109] G. Inc, "Machine Learning Glossary," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/glossary

[110] G. Inc, "Splitting the dataset," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/crash-course/training-and-test-sets/splitting-data

[111] G. Inc, "Validation Set: Another Partition," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/crash-course/validation/another-partition

[112] G. Inc, "Descending into ML: Training and Loss," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss

[113] G. Inc, "Regularization for Simplicity: L_2 Regularization," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/l2-regularization

[114] G. Inc, "early stopping," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/glossary#early_stopping

[115] G. Inc, "Classification: ROC Curve and AUC," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

[116] N. Corporation, "CuDNN -NVIDIA," 2020. [En ligne]. Disponible: https://developer.nvidia.com/cudnn

[117] L. Brown, "Accelerate Machine Learning with the cuDNN Deep Neural Network Library," 2014. [En ligne]. Disponible: https://devblogs.nvidia.com/accelerate-machine-learning-cudnn-deep-neural-network-library/

[118] G. Inc, "Réduction de la perte : le taux d'apprentissage," 2020. [En ligne]. Disponible: https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate

[119] Keras, "Fonctions d'activations," 2020. [En ligne]. Disponible: https://keras.io/activations/,

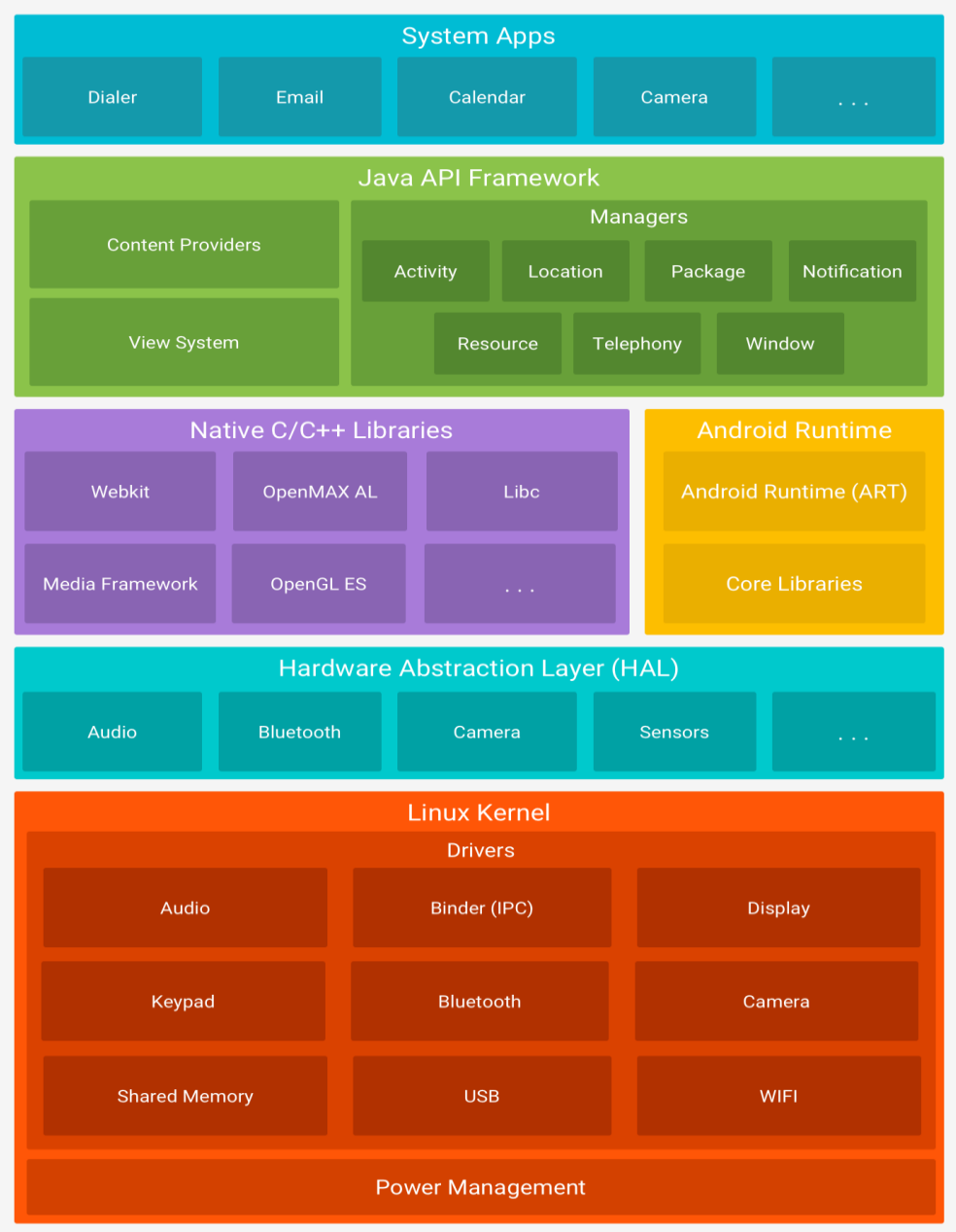
[120] Geekforgeeks, "ML | Extra Tree Classifier for Feature Selection," 2020. [En ligne]. Disponible: https://www.geeksforgeeks.org/ml-extra-tree-classifier-for-feature-selection/

[121] S. Chatterjee, "Good Data and Machine Learning," 2017. [En ligne]. Disponible: https://towardsdatascience.com/data-correlation-can-make-or-break-your-machine-learning-project-82ee11039cc9

[122] J. Bouteiller, "Android s'approche des 80% de parts de marché," 2019. [En ligne]. Disponible: https://mobilemarketing.fr/2019/04/30/android-sapproche-des-80-de-parts-de-marche/

ANNEXES

ANNEXE A : ARCHITECTURE ANDROID[19]



[19] Android Developers (2019), Platform Architecture, https://developer.android.com/guide/platform (Consulté le 7 juin 2019)

Le noyau de Linux est la base de la plateforme Android. Par exemple, ART (Android Runtime) prend son essor sur le noyau Linux pour les fonctionnalités sous-jacentes telles que les threads et la gestion de la mémoire de bas niveau. L'utilisation d'un noyau Linux permet à Android de tirer parti des principales fonctionnalités de sécurité et aux fabricants de périphériques de développer des pilotes matériels pour un noyau bien connu.

La couche d'abstraction matérielle (HAL) fournit des interfaces standards qui exposent les capacités matérielles des périphériques à la structure API Java de niveau supérieur. La couche HAL se compose de plusieurs modules de bibliothèque, chacun implémentant une interface pour un type spécifique de composant matériel, tel que le module caméra ou Bluetooth. Lorsqu'une API de structure appelle pour accéder à du matériel de périphérique, le système Android charge le module de bibliothèque pour ce composant matériel.

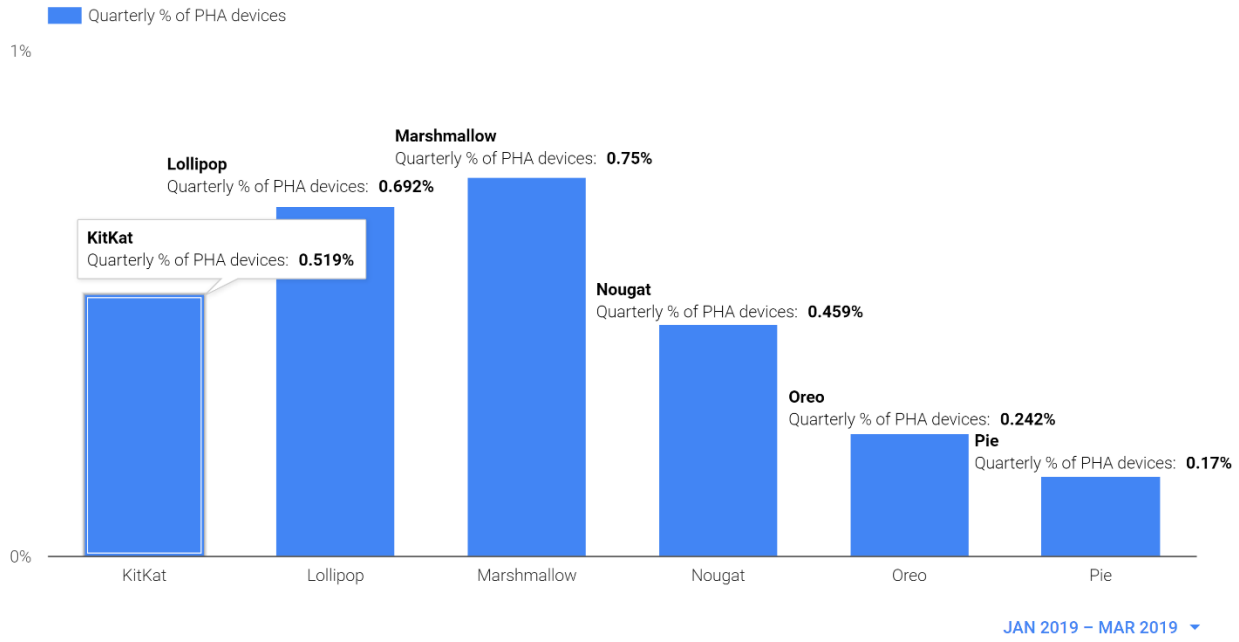
Pour les terminaux fonctionnant sous Android version 5.0 (API de niveau 21) ou supérieur, chaque application s'exécute dans son propre processus et avec sa propre instance. ART est conçu pour exécuter plusieurs machines virtuelles sur des périphériques à faible mémoire en exécutant des fichiers DEX, un format de code unique spécialement conçu pour Android optimisé pour un encombrement minimal de la mémoire.

De nombreux composants et services système Android principaux, tels que ART et HAL, sont construits à partir du code natif nécessitant des bibliothèques natives écrites en C et C ++. La plateforme Android fournit des APIs de structure Java pour exposer les applications de certaines de ces bibliothèques natives.

L'ensemble des fonctionnalités du système d'exploitation Android est disponible via des APIs écrites en langage Java. Ces APIs constituent les éléments de base dont vous avez besoin pour créer des applications Android en simplifiant la réutilisation des composants et des services systèmes modulaires de base. Les développeurs ont un accès complet aux mêmes APIs de structure que les applications système Android.

Android est livré avec un ensemble d'applications principales pour la messagerie électronique, la messagerie SMS, les calendriers, la navigation Internet, les contacts, etc. Les applications incluses avec la plateforme n'ont aucun statut particulier parmi les applications que l'utilisateur choisit d'installer. Ainsi, une application tierce peut remplacer ses applications par défaut.

ANNEXE B : PHA PAR VERSION ANDROID



ANNEXE C : DISTRIBUTION DES VERSIONS ANDROID[20]

Version	Nom de code	API	Distribution (%)	Cumulative Distribution (%)
2.3.3 -2.3.7	Gingerbread	10	0.3	0.3
4.0.3 -4.0.4	Ice Cream Sandwich	15	0.3	0.6
4.1.x	Jelly Bean	**16**	1.2	1.8
4.2.x		17	1.5	3.3
4.3		18	0.5	3.8
4.4	KitKat	19	6.9	10.7
5.0	Lollipop	21	3.0	13.7
5.1		22	11.5	25.2
6.0	Marshmallow	23	6.9	32.1
7.0	Nougat	24	11.4	43.5
7.1		25	7.8	51.3
8.0	Oreo	26	12.9	64.2
8.1		27	15.4	79.6
9	Pie	28	10.4	90

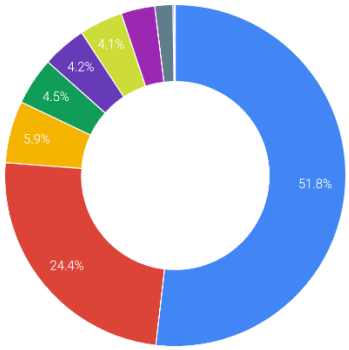
Données collectées au cours d'une période de 7 jours se terminant le 7 mai 2019. Toutes les versions avec une distribution inférieure à 0,1% ne sont pas affichées.

[20] Android Developers (2019), Distribution dashboard, https://developer.android.com/about/dashboards, (Consulté en Juin 2019)

ANNEXE E : POURCENTAGE D'INSTALLATIONS PHA PAR CATÉGORIES DANS GOOGLE PLAY ET EN DEHORS

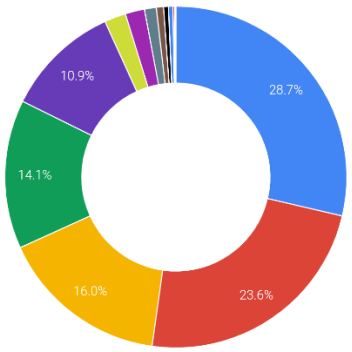
Jan 2019 – Mar 2019

Google Play



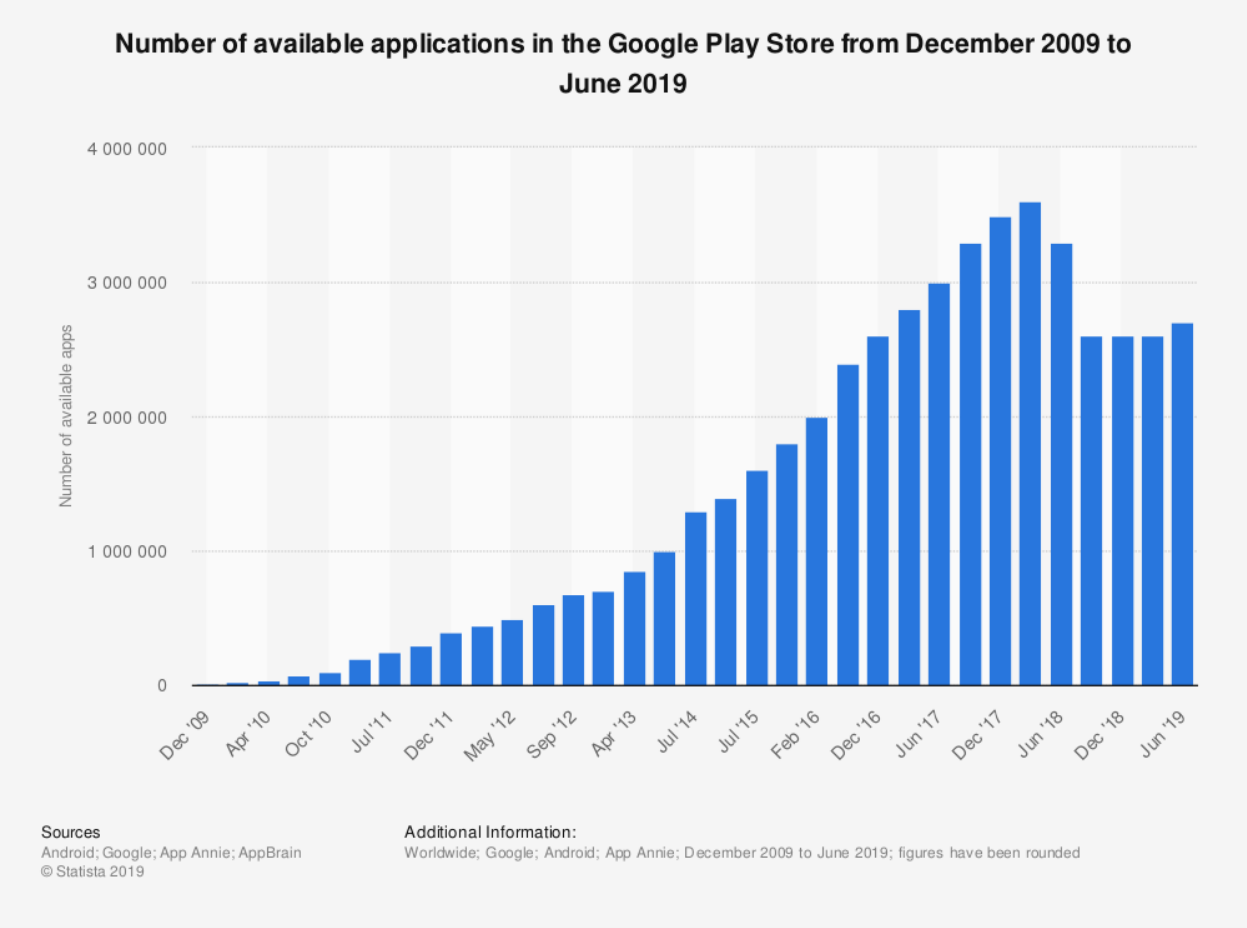
Category	PHA Install Rate
Back door	0.018920215%
Toll fraud	0.00890396%
Phishing	0.002142473%
SMS fraud	0.0016437209%
Click fraud	0.001525391%
Spyware	0.0015050707%
Hostile downloader	0.0011681044%
Trojan	0.0006230708%
Windows malware	0.0000465764%
Rooting	0.0000171564%
Privilege escalation	0.0000151963%
Commercial spyware	0.0000018459%
Spam	0.0000008064%
Ransomware	0.0000007952%
Call fraud	0.0000000016%
DDoS	0.0000000011%

Outside Google Play

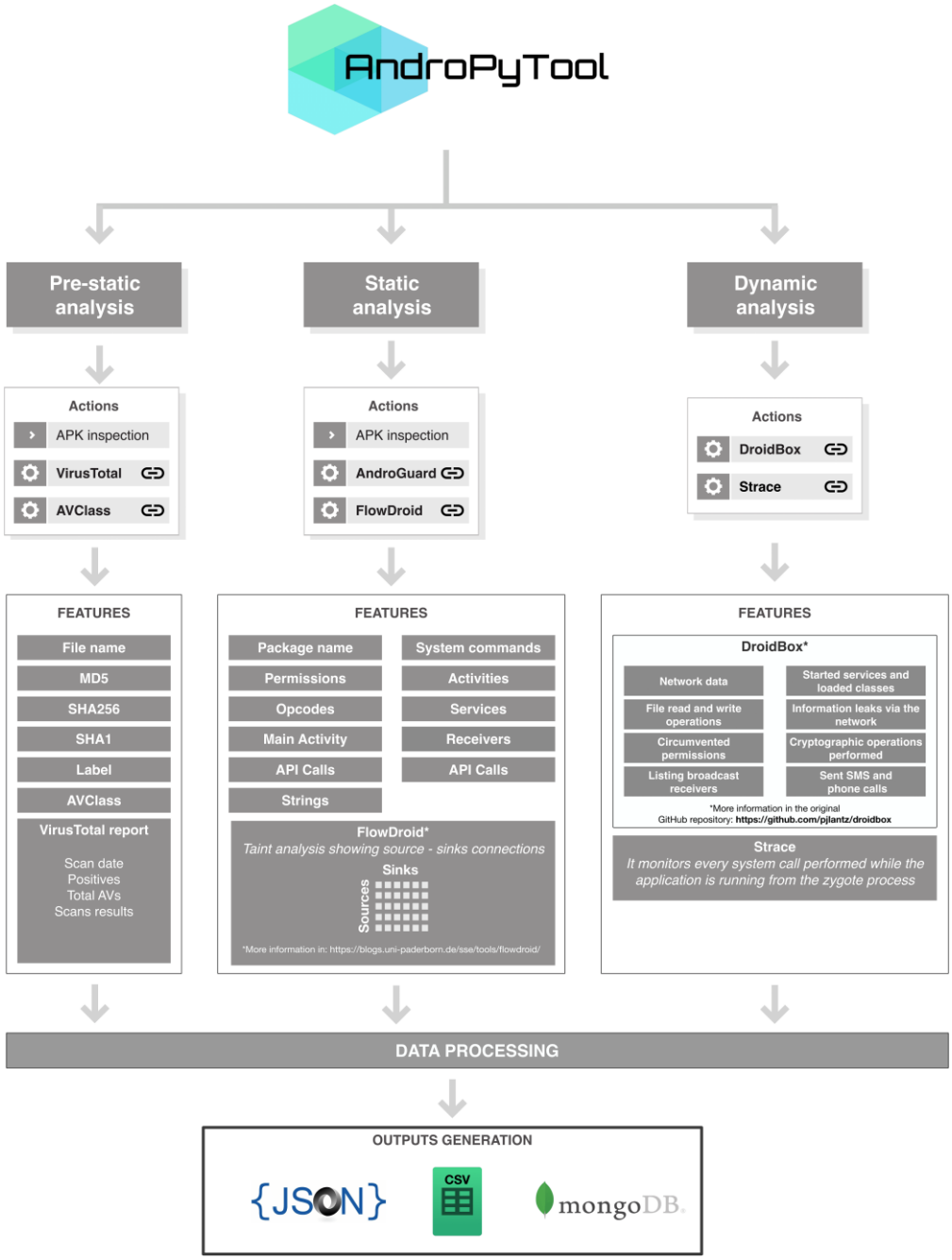


Category	PHA Install Rate
Back door	0.16925676%
Phishing	0.1391305%
Click fraud	0.09442484%
Trojan	0.08323293%
Hostile downloader	0.064265524%
Rooting	0.012035513%
Spyware	0.010724743%
Toll fraud	0.00662869%
Call fraud	0.003993515%
Commercial spyware	0.0027072925%
Privilege escalation	0.0024926969%
Ransomware	0.0008613814%
SMS fraud	0.0005357909%
Spam	0.0002233748%
Windows malware	0.0000022633%

ANNEXE F : NOMBRE DISPONIBLE D'APPLICATIONS SUR LE GOOGLE PLAY STORE DE DÉCEMBRE 2009 À JUIN 2019



ANNEXE G : PROCESSUS D'EXTRACTION DES CARACTÉRISTIQUES[21]



[21] Source : [100] A. Martín, R. Lara-Cabrera et D. Camacho, "Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset," vol. 52, p. 128-142, 2019. Fig. 1. AndroPyTool feature extraction process.

ANNEXE H : CODE ENTRAÎNEMENT

Figure 5.1 Code application : Code entraînement

```
import time
import warnings
import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization, Dropout, ActivityRegularization
import matplotlib.pyplot as plt

def preprocess_features(omnidroid_dataframe):
  selected_features = []
  for i in omnidroid_dataframe:
    selected_features.append(i)
  processed_features = selected_features.copy()
  return processed_features

omnidroid_dataframe = pd.read_csv("StaticTraining3,359.csv")
omnidroid_dataframe = omnidroid_dataframe.reindex(np.random.permutation(omnidroid_dataframe.inde
            x))

nbFeature = 840
itérations = 500
learning_rate = 0.002
loss = 'binary_crossentropy'
batch_size = 1024
DropoutRate = 0.3
omnidroid_dataframe = omnidroid_dataframe.reindex(np.random.permutation(omnidroid_dataframe.inde
            x))

omnidroidcor = omnidroid_dataframe.drop("label", axis=1).apply(lambda x: x.corr(omnidroid_dataframe
            .label))

compteur = 0
features_selected = []
for i in omnidroidcor:
    compteur = compteur + 1
    features_selected[compteur] = i

features_selected_sorted = sorted(features_selected.items(), key=lambda item: item[1],reverse= True)
compteur = 0
features_selected = [0]

for i in features_selected_sorted:
    compteur = compteur + 1
    if compteur < nbFeature + 1 :
```

```
44          features_selected.append(i[0])
45
46  features_selected_string = []
47  for feature in omnidroid_dataframe.columns[features_selected[0:nbFeature+1]]:
48      features_selected_string.append(feature)
49
50  start_time = time.time()
51
52  model_opti = Sequential()
53  model_opti.add(Dense(nbFeature,input_dim=nbFeature, activation='relu', kernel_initializer='uniform', bia
                s_initializer='ones'))
54  model_opti.add(Dropout(DropoutRate, noise_shape=None, seed=None))
55  model_opti.add(Dense(1, activation='sigmoid'))
56
57
58  model_opti.compile(optimizer = keras.optimizers.Adamax(learning_rate=learning_rate, beta_1=0.9, beta_
                2=0.999),
59      loss=loss,
60      metrics=[('accuracy'),tf.keras.metrics.Recall(),
61               tf.keras.metrics.Precision(),
62              tf.keras.metrics.TrueNegatives(),
63              tf.keras.metrics.TruePositives(),
64              tf.keras.metrics.FalsePositives(),
65              tf.keras.metrics.FalseNegatives(),
66              tf.keras.metrics.AUC(),
67              tf.keras.metrics.BinaryAccuracy(),
68              tf.keras.metrics.BinaryCrossentropy(),
69              tf.keras.metrics.Hinge(),
70              ]
71  )
72
73  history_opti = model_opti.fit(
74      omnidroid_dataframe[features_selected_string].head(18700).values[:,1:], #input
75      omnidroid_dataframe[features_selected_string].head(18700).values[:,0],  #target
76      itérations=itérations,
77      batch_size=batch_size,
78      verbose = 1,
79      validation_split = 0.17647,
80  )
81
82  resultstab_opti = model_opti.evaluate(
83      omnidroid_dataframe[features_selected_string].tail(3300).values[:,1:],
84      omnidroid_dataframe[features_selected_string].tail(3300).values[:,0],
85      verbose = 2
86  )
87  print("")
88
89  resultsdic_opti = {
90          "loss" : resultstab_opti[0],
91          "accuracy" : resultstab_opti[1],
92          "recall" : resultstab_opti[2],
```

```
93          "precision" : resultstab_opti[3],
94          "true_negatives" : resultstab_opti[4],
95          "true_positives" : resultstab_opti[5],
96          "false_positives" : resultstab_opti[6],
97          "false_negatives" : resultstab_opti[7],
98          "auc" : resultstab_opti[8],
99          "binary_accuracy" : resultstab_opti[9],
100         "binary_crossentropy" : resultstab_opti[10],
101         "hinge" : resultstab_opti[11]
102
103 }
104 f1_score = 2*(resultsdic_opti["precision"]*resultsdic_opti["recall"])/(resultsdic_opti["precision"]+resultsd
                ic_opti["recall"])
105 print("f1_score = " + str(round(f1_score,4)))
```

ANNEXE I : CODE APPLICATION

Figure 5.2 Code application : MainActivity.kt

```
1   package com.amo.brainshield
2   import android.Manifest
3   import android.app.AlertDialog
4   import android.content.Intent
5   import android.content.pm.PackageManager
6   import android.graphics.drawable.Drawable
7   import android.graphics.drawable.TransitionDrawable
8   import android.os.Bundle
9   import android.os.Handler
10  import android.view.Gravity
11  import android.view.Menu
12  import android.view.MenuItem
13  import android.view.View
14  import android.view.View.GONE
15  import android.view.animation.AnimationUtils
16  import android.widget.Toast
17  import androidx.appcompat.app.AppCompatActivity
18  import androidx.appcompat.widget.Toolbar
19  import androidx.core.app.ActivityCompat
20  import androidx.core.app.NavUtils
21  import androidx.core.content.ContextCompat
22  import androidx.recyclerview.widget.LinearLayoutManager
23  import androidx.recyclerview.widget.RecyclerView
24  import com.github.kittinunf.fuel.Fuel
25  import com.github.kittinunf.fuel.core.FileDataPart
26  import com.github.kittinunf.fuel.core.Method
27  import com.github.kittinunf.result.Result
28  import kotlinx.android.synthetic.main.activity_main.*
29  import java.io.File
30
31  class MainActivity : AppCompatActivity() {
32      private val ipv4Address: String
33          get() = getString(R.string.ipv4)
34      private val portNumber: String
35          get() = getString(R.string.port)
36      private val storagePermissionCode = 1
37      private val waitingTime: Long = 50
38      private lateinit var mRecyclerView: RecyclerView
39      private lateinit var mAdapter: RecyclerView.Adapter<*>
40      private lateinit var installedApps: ArrayList<AppInfo>
41      private lateinit var appManager: AppManager
42      private lateinit var selectedFileApkName: String
43      private lateinit var selectedFileApkPath: String
44      private var myItemShouldBeEnabled = true
45
```

```
46      // Called by the OS when the activity is first created. This is where you initialize any UI elements or dat
            a objects. You also have the savedInstanceState of the activity that contains its previously saved st
            ate, and you can use it to recreate that state.
47      public override fun onCreate(savedInstanceState: Bundle?) {
48          super.onCreate(savedInstanceState)
49          setContentView(R.layout.activity_main)
50          progressBarWaitingUpload.visibility = GONE
51          result.visibility = GONE
52          SelectAPKTextView.visibility = GONE
53
54          //  Setting the personalized toolbar
55          val toolbar: Toolbar = findViewById(R.id.my_toolbar)
56          setSupportActionBar(toolbar)
57          supportActionBar?.title = null
58          val bounceAnimationLogo = AnimationUtils.loadAnimation(this, R.anim.bounce_animation)
59          this.yourlogo.startAnimation(bounceAnimationLogo)
60          this.BrainShield.startAnimation(bounceAnimationLogo)
61
62          //  Displaying the installed application
63          installedApps = ArrayList()
64          mRecyclerView = findViewById(R.id.listingApkRecyclerView)
65          val layoutManager = LinearLayoutManager(this)
66          mRecyclerView.layoutManager = layoutManager
67          appManager = AppManager(this)
68          installedApps = appManager.appsSortedByName
69          mAdapter = MainActivityAdapter(
70              applicationContext,
71              installedApps
72          )
73          mRecyclerView.adapter = mAdapter
74          this.mRecyclerView.startAnimation(bounceAnimationLogo)
75
76          if (ContextCompat.checkSelfPermission(
77                  this@MainActivity,
78                  Manifest.permission.READ_EXTERNAL_STORAGE
79              ) == PackageManager.PERMISSION_GRANTED
80          ) {
81              println("You already have storage permission")
82          } else {
83              requestStoragePermission()
84          }
85
86      }
87
88
89      // onCreateOptionsMenu() to specify the options menu for an activity
90      override fun onCreateOptionsMenu(menu: Menu?): Boolean {
91          menuInflater.inflate(R.menu.scanmenu, menu)
92          return super.onCreateOptionsMenu(menu)
93      }
94
```

```
95      // onOptionsItemSelected defines what actions are made when user click on menu items
96      override fun onOptionsItemSelected(item: MenuItem) =
97          when (item.itemId) {
98
99              R.id.SelectApkFromFile -> {
100                 selectAPK()
101                 true
102             }
103
104             R.id.menu_offLine -> {
105                 val myIntent = Intent(this, ResultOfflineActivity::class.java)
106                 startOfflineActivity(myIntent)
107                 true
108             }
109             // if user press home button go to home
110             R.id.home -> {
111                 NavUtils.navigateUpFromSameTask(this)
112                 true
113             }
114             // if user press select all check all apps
115             R.id.SelectAll -> {
116                 val bounceAnimationLogo = AnimationUtils.loadAnimation(this, R.anim.bounce_animation_re
                cycler_view)
117                 checkAllApps()
118                 mRecyclerView.startAnimation(bounceAnimationLogo)
119                 true
120             }
121             R.id.UnselectAll -> {
122                 val bounceAnimationLogo = AnimationUtils.loadAnimation(this, R.anim.bounce_animation_re
                cycler_view)
123                 uncheckAllApps()
124                 mRecyclerView.startAnimation(bounceAnimationLogo)
125                 true
126             }
127             else -> {
128                 super.onOptionsItemSelected(item)       //  If we got here, the user's action was not recognized. I
                nvoke the superclass to handle it
129
130             }
131         }
132
133
134     fun startStaticActivity(view: View) {
135         var compteur = 0
136         for (application in installedApps) {
137             if (application.isSelected) {
138                 compteur += 1
139             }
140         }
141         if (compteur != 0) {
142             progressBarWaitingUpload.visibility = View.VISIBLE
```

```
143             scanApps(ResultStaticActivity::class.java)
144             ScanDynamic.visibility = GONE
145             ScanStatic.visibility = GONE
146             ScanVT.visibility = GONE
147
148         } else {
149             finish()
150             overridePendingTransition(0, 0)
151             val intent = Intent(this, MainActivity::class.java)
152             startActivity(intent)
153
154             Toast.makeText(
155                 this,
156                 "Please select at least one application",
157                 Toast.LENGTH_LONG)
158             .show()
159         }
160     }
161
162     fun startDynamicActivity(view: View) {
163         var compteur = 0
164         for (application in installedApps) {
165             if (application.isSelected) {
166                 compteur += 1
167             }
168         }
169         if (compteur != 0) {
170             progressBarWaitingUpload.visibility = View.VISIBLE
171             scanApps(ResultDynamicActivity::class.java)
172             ScanDynamic.visibility = GONE
173             ScanStatic.visibility = GONE
174             ScanVT.visibility = GONE
175
176         } else {
177             finish()
178             overridePendingTransition(0, 0)
179             val intent = Intent(this, MainActivity::class.java)
180             startActivity(intent)
181
182             Toast.makeText(
183                 this,
184                 "Please select at least one application",
185                 Toast.LENGTH_LONG)
186                 .show()
187         }
188     }
189
190     fun startServerActivity(view: View) {
191         var compteur = 0
192         for (application in installedApps) {
193             if (application.isSelected) {
```

```
194             compteur += 1
195         }
196     }
197     if (compteur != 0) {
198         progressBarWaitingUpload.visibility = View.VISIBLE
199         scanApps(VirusTotalActivity::class.java)
200         ScanDynamic.visibility = GONE
201         ScanStatic.visibility = GONE
202         ScanVT.visibility = GONE
203 
204     } else {
205         finish()
206         overridePendingTransition(0, 0)
207         val intent = Intent(this, MainActivity::class.java)
208         startActivity(intent)
209 
210         Toast.makeText(
211             this,
212             "Please select at least one application",
213             Toast.LENGTH_LONG)
214             .show()
215     }
216 }
217 
218 private fun startOfflineActivity(myActivity: Intent) {
219     var compteur = 0
220     for (application in installedApps) {
221         if (application.isSelected) {
222             compteur += 1
223         }
224     }
225     if (compteur != 0) {
226         val intent = Intent(this, ResultOfflineActivity::class.java)
227         intent.putStringArrayListExtra("MyApps", getAppNameCheck())
228         startActivity(intent)
229 
230     } else {
231         finish()
232         overridePendingTransition(0, 0)
233         val intent = Intent(this, MainActivity::class.java)
234         startActivity(intent)
235         Toast.makeText(
236             this,
237             "Please select at least one application",
238             Toast.LENGTH_LONG
239         ).show()
240     }
241 }
242 
243 // checkAllApps for checking all apps in one time
244 private fun checkAllApps() {
```

```
245         for (item in installedApps) {
246             item.isSelected = true
247         }
248         mAdapter.notifyDataSetChanged()
249     }
250
251     // uncheckAllApps for unchecking all apps in one tine
252     private fun uncheckAllApps() {
253         for (item in installedApps) {
254             item.isSelected = false
255         }
256         mAdapter.notifyDataSetChanged()
257     }
258
259     // getAppPackageCheck returns the name of the packages of all apps selected
260     private fun getAppPackageCheck(): MutableList<String> {
261         val listOfAppsToScan: MutableList<String> = mutableListOf()
262         for (item in installedApps) {
263             if (item.isSelected) {
264                 listOfAppsToScan.add(item.appPackage!!)
265             }
266         }
267         return listOfAppsToScan
268     }
269
270     // getAppNameCheck returns the name of the applications selected
271     private fun getAppNameCheck(): ArrayList<String> {
272         val listOfAppsToScan: ArrayList<String> = ArrayList()
273         for (item in installedApps) {
274             if (item.isSelected) {
275                 listOfAppsToScan.add(item.appName!!)
276             }
277         }
278         return listOfAppsToScan
279     }
280
281     // manipulation of string for communication with the server
282     private fun splitString(string: String): MutableList<String> {
283         val myString = string.subSequence(1, string.length - 1)
284         val listString: MutableList<String> = mutableListOf()
285         for (boutDeChaine in myString.split(", ")) {
286             listString.add(boutDeChaine)
287         }
288         return listString
289     }
290     // main function on the app. Communicates with server to identify which apps to send
291     private fun scanApps(uri:Class<*>) {
292         val listOfAppsToScanFinal: MutableList<String> = mutableListOf()
293         val listOfPackagesToScan = getAppPackageCheck()
294         lateinit var body: String
295         val runnable = Runnable {
```

```
296         fun run() {
297             // redirect to ResultsActivity
298             val intent = Intent(this, uri)
299             intent.putStringArrayListExtra("MyApps", getAppNameCheck())
300             startActivity(intent)
301         }
302         run()
303     }
304
305     // asking the servers which apps need to be upload
306     Fuel.post("http://".plus(ipv4Address).plus(":").plus(portNumber).plus("/IsAPKExist"))
307         .body(listOfPackagesToScan.toString())
308         .response { _, response, result ->
309             when (result) {
310                 // Server's answer with the list of the apps
311                 is Result.Success -> {
312                     progressBarWaitingUpload.visibility = GONE
313                     ScanDynamic.visibility = View.VISIBLE
314                     ScanStatic.visibility = View.VISIBLE
315                     ScanVT.visibility = View.VISIBLE
316                     body = String(response.data)
317                     if (body == "[]") {
318                         println("ALL APPLICATIONS SELECTED ARE IN SERVER")
319                         val handler = Handler()
320                         handler.postDelayed(runnable, waitingTime)
321                     } else {
322                         this.result.visibility = View.VISIBLE
323                         val listOfPackageToBeUpload = splitString(body)
324                         lateinit var packageNameWithout: CharSequence
325                         var compteur = 0
326                         this.result.visibility = View.VISIBLE
327                         for (applicationInServer in listOfPackageToBeUpload) {
328                             compteur += 1
329                         }
330                         var numberOfSelectedApplications = 0
331                         for (applicationSelected in installedApps) {
332                             if (applicationSelected.isSelected) {
333                                 numberOfSelectedApplications += 1
334                             }
335                         }
336                         this.result.text =
337                             getString(R.string.ApplicationReady).plus(
338                                 numberOfSelectedApplications - compteur
339                             )
340                                 .plus("/").plus(
341                                     numberOfSelectedApplications.toString().plus(
342                                         ")\n \n Uploading : ".plus(compteur)
343                                     )
344                                 )
345
346                         for (item in installedApps) {
```

```
347                     if (item.isSelected) {
348                         for (packageName in listOfPackageToBeUpload) {
349                             packageNameWithout = packageName.subSequence(
350                                 1,
351                                 packageName.length - 1
352                             )
353                             if (item.appPackage == packageNameWithout) {
354                                 item.needUpload = true
355                                 listOfAppsToScanFinal.add(item.appPackage!!)
356                                 // sending the correct apps
357                                 transferFilesFuel(
358                                     packageManager.getApplicationInfo(
359                                         item.appPackage!!,
360                                         0
361                                     ).sourceDir, item.appPackage
362                                 )
363                             }
364                         }
365                     }
366 
367                 }
368             }
369         }
370         // if you did not get a response from the server
371         is Result.Failure -> {
372             this.progressBarWaitingUpload.visibility = GONE
373             ScanDynamic.visibility = View.VISIBLE
374             ScanStatic.visibility = View.VISIBLE
375             ScanVT.visibility = View.VISIBLE
376             finish()
377             val intent = Intent(this, MainActivity::class.java)
378             startActivity(intent)
379             Toast.makeText(this, getString(R.string.Failed), Toast.LENGTH_LONG).show()
380 
381         }
382     }
383 }
384 }
385 
386 // Sending non existing .apk to server
387 private fun transferFilesFuel(Path: String, PackageName: String?) {
388     val runnable = Runnable {
389         fun run() {
390             // redirect to ResultsActivity
391             val intent = Intent(this,ResultStaticActivity::class.java)
392             intent.putStringArrayListExtra("MyApps", getAppNameCheck())
393             startActivity(intent)
394         }
395         run()
396     }
397 
```

```
398         var numberOfSelectedApplications = 0
399         for (applicationSelected in installedApps) {
400             if (applicationSelected.isSelected) {
401                 numberOfSelectedApplications += 1
402             }
403         }
404
405         val file = FileDataPart.from(Path, name = "apk", filename = PackageName.plus(".apk"))
406
407         Fuel.upload(
408             "http://".plus(ipv4Address).plus(":").plus(portNumber).plus("/transferFiles"),
409             method = Method.POST
410         )
411             .add(file)
412             .response { _, response, result ->
413                 when (result) {
414                     is Result.Success -> {
415                         val body = String(response.data)
416                         this.result.text = getString(R.string.ApplicationReady).plus(body).plus("/")
417                             .plus(numberOfSelectedApplications.toString().plus(")"))
418
419                         if (numberOfSelectedApplications.toString() == body) {
420                             val handler = Handler()
421                             handler.postDelayed(runnable, waitingTime)
422                         }
423
424
425                     }
426                     is Result.Failure -> {
427                         this.result.text = getString(R.string.Failed)
428                         this.progressBarWaitingUpload.visibility = GONE
429                     }
430                 }
431             }
432
433     }
434
435     // called by button select .apk file to select an .apk in storage
436     fun selectAPK() {
437         val intent = Intent()
438         intent.type = "*/*"
439         intent.action = Intent.ACTION_GET_CONTENT
440         startActivityForResult(intent, 0)
441     }
442
443     // called when you select a media within the activity of searching a media in the file
444     override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
445         super.onActivityResult(requestCode, resultCode, data)
446         if (resultCode == RESULT_OK && (data!!.data.toString().endsWith("apk"))) {
447             val uri = data.data
448             val file = File(uri!!.path.toString())
```

```
449         selectedFileApkName = file.name
450         selectedFileApkPath = uri.path.toString().split("raw:")[1]
451         SelectAPKTextView.text = selectedFileApkPath
452
453         if (selectedFileApkName.endsWith("apk")) {
454
455
456             val runnable = Runnable {
457                 fun run() {
458                     // redirect to ResultsActivity
459                     val intent = Intent(this, ResultStorageActivity::class.java)
460                     startActivity(intent)
461                 }
462                 run()
463             }
464
465 //              mRecyclerView.visibility = GONE
466             SelectAPKTextView.visibility = GONE
467
468             Fuel.post("http://".plus(ipv4Address).plus(":").plus(portNumber).plus("/IsSelectedAPKExist"))
469                 .body(selectedFileApkName)
470                 .response { _, response, result ->
471                     when (result) {
472                         is Result.Success -> {
473
474                             val body = String(response.data)
475                             if (body == "[]") {
476                                 println("APPLICATION SELECTED ARE IN SERVER")
477                                 val handler = Handler()
478                                 handler.postDelayed(runnable, waitingTime)
479                             } else {
480                                 this.result.visibility = View.VISIBLE
481                                 this.result.text = getString(R.string.UploadingApkToServer)
482                                 val apkSelected = FileDataPart.from(
483                                     selectedFileApkPath,
484                                     name = "apk",
485                                     filename = selectedFileApkName
486                                 )
487                                 Fuel.upload(
488                                     "http://".plus(ipv4Address).plus(":").plus(
489                                         portNumber
490                                     ).plus("/transferFiles"), method = Method.POST
491                                 )
492                                     .add(apkSelected)
493                                     .response { result ->
494                                         when (result) {
495                                             is Result.Success -> {
496                                                 val handler = Handler()
497                                                 handler.postDelayed(runnable, waitingTime)
498                                             }
499                                             is Result.Failure -> {
```

```
500                                 this.result.visibility = View.VISIBLE
501                                 this.result.text = getString(R.string.Failed)
502                             }
503                         }
504                     }
505                 }
506                 is Result.Failure -> {
507                     this.result.visibility = View.VISIBLE
508                     progressBarWaitingUpload.visibility = GONE
509                     finish()
510                     val intent = Intent(this, MainActivity::class.java)
511                     startActivity(intent)
512                     Toast.makeText(this, getString(R.string.Failed), Toast.LENGTH_LONG)
513                         .show()
514                 }
515             }
516         }
517     } else {
518         Toast.makeText(this, "Please select an .APK", Toast.LENGTH_LONG).show()
519     }
520 } else {
521     Toast.makeText(this, "Please select an .APK", Toast.LENGTH_LONG).show()
522 }
523 }
524
525 // requesting the storage permission
526 private fun requestStoragePermission() {
527     if (ActivityCompat.shouldShowRequestPermissionRationale(
528             this,
529             Manifest.permission.READ_EXTERNAL_STORAGE
530         )
531     ) {
532         AlertDialog.Builder(this)
533             .setTitle("Permission needed")
534             .setMessage("STORAGE_PERMISSION needed for sending .apk to server")
535             .setPositiveButton(
536                 "ok"
537             ) { _, _ ->
538                 ActivityCompat.requestPermissions(
539                     this@MainActivity,
540                     arrayOf(Manifest.permission.READ_EXTERNAL_STORAGE),
541                     storagePermissionCode
542                 )
543             }
544             .setNegativeButton(
545                 "cancel"
546             ) { dialog, _ -> dialog.dismiss()
547             .create().show()
548     } else {
549         ActivityCompat.requestPermissions(
550
```

```
551             this,
552             arrayOf(Manifest.permission.READ_EXTERNAL_STORAGE),
553             storagePermissionCode
554         )
555     }
556 }
557
558 // dealing the results of the storage permission request
559 override fun onRequestPermissionsResult(
560     requestCode: Int,
561     permissions: Array<String>,
562     grantResults: IntArray
563 ) {
564     if (requestCode == storagePermissionCode) {
565         if (grantResults.isNotEmpty() && grantResults[0] == PackageManager.PERMISSION_GRANTE
            D) {
566             Toast.makeText(this, "Permission GRANTED", Toast.LENGTH_SHORT).show()
567         } else {
568             Toast.makeText(this, "Permission DENIED", Toast.LENGTH_SHORT).show()
569         }
570     }
571 }
572 }
```

Figure 5.3 Code application : StaticResults.kt

Code commenté de la page des résultats statiques. Pour des raisons de mise en page, seul le code des résultats statiques est affiché mais ceux concernant les dynamiques sont très ressemblants.

```
1   package com.amo.brainshield
2
3   import android.annotation.SuppressLint
4   import android.os.Bundle
5   import android.text.method.ScrollingMovementMethod
6   import android.view.MenuItem
7   import android.view.View
8   import android.widget.ProgressBar
9   import android.widget.Scroller
10  import android.widget.TextView
11  import androidx.appcompat.app.AppCompatActivity
12  import androidx.appcompat.widget.Toolbar
13  import androidx.core.app.NavUtils
14  import androidx.recyclerview.widget.LinearLayoutManager
15  import androidx.recyclerview.widget.RecyclerView
16  import com.github.kittinunf.fuel.Fuel
17  import com.github.kittinunf.result.Result
18  import com.google.gson.GsonBuilder
19  import kotlinx.android.synthetic.main.activity_main.*
20  import kotlinx.android.synthetic.main.result_static.*
21  import java.util.*
22
23  class ResultStaticActivity : AppCompatActivity() {
24
25      private val ipv4Address: String
26          get() = getString(R.string.ipv4)
27      private val portNumber: String
28          get() = getString(R.string.port)
29
30      private lateinit var mRecyclerView: RecyclerView
31      private lateinit var installedApps: ArrayList<AppInfo>
32      private lateinit var finalApps: ArrayList<AppInfo>
33      @SuppressLint("WrongViewCast")
34      private lateinit var checkedApps: ArrayList<String>
35      private lateinit var appManager: AppManager
36      @SuppressLint("ResourceType")
37
38      override fun onCreate(savedInstanceState: Bundle?) {
39          super.onCreate(savedInstanceState)
40          setContentView(R.layout.result_static)
41
42          val myToolbar: Toolbar = findViewById(R.id.my_toolbar)
43          val layoutManager = LinearLayoutManager(this)
44          setSupportActionBar(myToolbar)
```

```
45          supportActionBar?.title = null
46          supportActionBar?.setDisplayHomeAsUpEnabled(true)
47          GetPredictionButton.visibility = View.GONE
48          responsePredictionText.visibility = View.GONE
49          progressBarWaitingResult.visibility = View.GONE
50          progressBarWaitingFeatureExtraction.visibility = View.GONE
51          checkedApps = intent.getStringArrayListExtra("MyApps")
52          installedApps = ArrayList()
53          finalApps = ArrayList()
54          mRecyclerView = findViewById(R.id.listingApkRecyclerView)
55          mRecyclerView.layoutManager = layoutManager
56          appManager = AppManager(this)
57          installedApps = appManager.appsSortedByPackage
58          for (item in installedApps) {
59              if (item.appName in checkedApps) {
60                  finalApps.add(item)
61              }
62          }
63      }
64
65      fun featureExtraction(view: View) {
66          val progressBar: ProgressBar = findViewById(R.id.progressBarWaitingFeatureExtraction)
67          progressBarWaitingFeatureExtraction.visibility = View.VISIBLE
68
69          Fuel.get("http://".plus(ipv4Address).plus(":").plus(portNumber).plus("/FeatureExtraction"))
70              .response { _, response, result ->
71                  when (result) {
72                      is Result.Success -> {
73                          progressBar.visibility = View.GONE
74                          GetFeatureExtractionButton.visibility = View.GONE
75                          GetPredictionButton.visibility = View.VISIBLE
76                          responsePredictionText.visibility = View.VISIBLE
77                          val body = String(response.data)
78                          this.responseFeatureExtractionText.text = body
79                      }
80                      is Result.Failure -> {
81                          this.responseFeatureExtractionText.text = getString(R.string.WaitingFeatureExtraction)
82                      }
83                  }
84              }
85      }
86
87      fun prediction(view: View) {
88          val progressBar: ProgressBar = findViewById(R.id.progressBarWaitingResult)
89          progressBarWaitingResult.visibility = View.VISIBLE
90
91          Fuel.get("http://".plus(ipv4Address).plus(":").plus(portNumber).plus("/prediction"))
92              .response { _, response, result ->
93                  println("RESULT=$result")
94                  when (result) {
95                      is Result.Success -> {
```

```
96                  progressBar.visibility = View.GONE
97                  responseFeatureExtractionText.visibility = View.GONE
98                  GetPredictionButton.visibility = View.GONE
99                  responsePredictionText.visibility = View.GONE
100                 val body = String(response.data)
101                 val gson = GsonBuilder().create()
102                 val homeFeed = gson.fromJson(body, HomeFeed::class.java)
103                 responsePredictionText.text = body
104                 responsePredictionText.canScrollVertically(0)
105
106                    mRecyclerView.adapter =
107                    ResultStaticActivityAdapter(applicationContext, finalApps, homeFeed)
108                 (mRecyclerView.adapter as ResultStaticActivityAdapter).notifyDataSetChanged()
109             }
110             is Result.Failure -> {
111                 progressBar.visibility = View.GONE
112                 this.responsePredictionText.text = getString(R.string.Failed)
113             }
114         }
115     }
116 }
117
118 override fun onOptionsItemSelected(item: MenuItem) =
119     when (item.itemId) {
120         android.R.id.home -> {
121             NavUtils.navigateUpFromSameTask(this)
122             true
123         }
124         else -> {
125             super.onOptionsItemSelected(item)      //  If we got here, the user's action was not recognized. I
          nvoke the superclass to handle it
126         }
127     }
128 }
129
130
131 class HomeFeed(val application: List<Applications>)
132 class Applications(val prediction: String, val predictionNumber: String, val packageName:String)
```

Figure 5.4 Code application : AppInfo.kt

```
1   package com.amo.brainshield
2
3   import android.graphics.drawable.Drawable
4
5   class AppInfo {
6       var appName: String? = null
7       var appPackage: String? = null
8       var appIcon: Drawable? = null
9       var isSelected: Boolean = false
```

```
    var needUpload: Boolean = false
    var appResultText: String? = null
    var appPrediction: String? = null
    var appPermission:String? = null
}
```

Figure 5.5 Code application : AppManager.kt

```
package com.amo.brainshield
import android.content.Context
import android.content.pm.ApplicationInfo
import android.content.pm.PackageManager
import android.graphics.drawable.Drawable
import androidx.core.content.ContextCompat
import java.util.*


class AppManager(private val mContext: Context) {
    private val myApps: ArrayList<AppInfo> = ArrayList()
    val appsSortedByName: ArrayList<AppInfo>
        get() {
            loadAppsSortedByName()
            return myApps
        }

    val appsSortedByPackage: ArrayList<AppInfo>
        get() {
            loadAppsSortedByPackage()
            return myApps
        }

    fun isSystemPackage(applicationInfo: ApplicationInfo): Boolean {
        return applicationInfo.flags and ApplicationInfo.FLAG_SYSTEM != 0
    }


    private fun loadAppsSortedByPackage() {
        val packages = mContext.packageManager.getInstalledApplications(0)
        for (packageInfo in packages) {
            if (!isSystemPackage(packageInfo)) {
                val newApp = AppInfo()
                newApp.appName = getApplicationLabelByPackageName(packageInfo.packageName)
                newApp.appPackage = packageInfo.packageName
                newApp.appIcon = getAppIconByPackageName(packageInfo.packageName)
                myApps.add(newApp)
            }

        }
        myApps.sortWith(Comparator { s1, s2 ->
            s1.appPackage!!.compareTo(
```

```
43              s2.appPackage!!,
44              ignoreCase = true
45          )
46      })
47
48  }
49
50  private fun loadAppsSortedByName() {
51      val packages = mContext.packageManager.getInstalledApplications(0)
52      for (packageInfo in packages) {
53          if (!isSystemPackage(packageInfo)) {
54              val newApp = AppInfo()
55              newApp.appName = getApplicationLabelByPackageName(packageInfo.packageName)
56              newApp.appPackage = packageInfo.packageName
57              newApp.appIcon = getAppIconByPackageName(packageInfo.packageName)
58              myApps.add(newApp)
59          }
60
61      }
62      myApps.sortWith(Comparator { s1, s2 ->
63          s1.appName!!.compareTo(
64              s2.appName!!,
65              ignoreCase = true
66          )
67      })
68
69  }
70
71  private fun getAppIconByPackageName(packageName: String): Drawable {    // Custom method to get
        application icon by package name
72      val icon: Drawable?
73      icon = try {
74          mContext.packageManager.getApplicationIcon(packageName)
75      } catch (e: PackageManager.NameNotFoundException) {
76          e.printStackTrace()
77          ContextCompat.getDrawable(
78              mContext,
79              R.drawable.ic_launcher_background
80          )           // Get a default icon
81      }
82      return icon!!
83  }
84
85  private fun getApplicationLabelByPackageName(packageName: String): String {    // Custom method t
        o get application label by package name
86      val packageManager = mContext.packageManager
87      val applicationInfo: ApplicationInfo?
88      var label = "Unknown"
89      try {
90          applicationInfo = packageManager.getApplicationInfo(packageName, 0)
91          if (applicationInfo != null) {
```

```
92              label = packageManager.getApplicationLabel(applicationInfo) as String
93          }
94      } catch (e: PackageManager.NameNotFoundException) {
95          e.printStackTrace()
96      }
97      return label
98  }
99  }
```

Figure 5.6 Code application : MainActivityAdapter.kt

```
1   package com.amo.brainshield
2
3   import android.annotation.SuppressLint
4   import android.content.Context
5   import android.view.LayoutInflater
6   import android.view.View
7   import android.view.ViewGroup
8   import android.view.animation.AnimationUtils
9   import android.widget.CheckBox
10  import android.widget.ImageView
11  import android.widget.RelativeLayout
12  import android.widget.TextView
13  import androidx.recyclerview.widget.RecyclerView
14  import java.util.*
15
16  class MainActivityAdapter(
17      private val mContext: Context,
18      private var mDataSet: ArrayList<AppInfo>
19  ) : RecyclerView.Adapter<MainActivityAdapter.ViewHolder>() {
20      private val bounceAnimationLogo = AnimationUtils.loadAnimation(mContext, R.anim.bounce_animat
            ion_logo)
21      private val bounceAnimationText = AnimationUtils.loadAnimation(mContext, R.anim.bounce_animati
            on_text)
22
23      class ViewHolder @SuppressLint("WrongViewCast")
24      constructor(v: View) : RecyclerView.ViewHolder(v) {
25
26          var mTextViewLabel: TextView =
27              v.findViewById<View>(R.id.apkNameTextView) as TextView       // Get the widgets reference fr
                om custom layout
28          var mImageViewIcon: ImageView = v.findViewById<View>(R.id.packageImageView) as ImageVie
                w
29          var mAppSelect: CheckBox = v.findViewById<View>(R.id.appSelect) as CheckBox
30          var mItem: RelativeLayout = v.findViewById<View>(R.id.item) as RelativeLayout
31
32      }
33
```

```
34      override fun onCreateViewHolder(
35          parent: ViewGroup,
36          viewType: Int
37      ): ViewHolder {
38          val v = LayoutInflater.from(mContext).inflate(R.layout.layout_apk_item, parent, false)
39          return ViewHolder(v)
40      }
41
42      override fun onBindViewHolder(holder: ViewHolder, position: Int) {
43          val icon = mDataSet[position].appIcon                    // GET the current app icon
44          val appName = mDataSet[position].appName                      // GET the current app name
45          val packageName = mDataSet[position].appPackage                       // GET the current app name
46          holder.mTextViewLabel.text = appName                     // SET the current app
47          holder.mImageViewIcon.setImageDrawable(icon)               // SET the current app icon
48          holder.mItem.setOnClickListener {
49              mDataSet[position].isSelected = !mDataSet[position].isSelected
50              holder.mImageViewIcon.startAnimation(bounceAnimationLogo)
51              holder.mTextViewLabel.startAnimation(bounceAnimationText)
52              holder.mItem.startAnimation(bounceAnimationText)
53              this@MainActivityAdapter.notifyItemChanged(position)
54
55          }
56          holder.mAppSelect.setOnClickListener {
57              mDataSet[position].isSelected = !mDataSet[position].isSelected
58              holder.mImageViewIcon.startAnimation(bounceAnimationLogo)
59              holder.mTextViewLabel.startAnimation(bounceAnimationText)
60              holder.mItem.startAnimation(bounceAnimationText)
61              this@MainActivityAdapter.notifyItemChanged(position)
62          }
63          holder.mAppSelect.isChecked = mDataSet[position].isSelected
64
65          if (holder.mAppSelect.isChecked) {
66              holder.mImageViewIcon.startAnimation(bounceAnimationLogo)
67          }
68
69      }
70
71      override fun getItemCount(): Int {                       // Count the installed apps
72          return mDataSet.size
73      }
74  }
```

Figure 5.7 Code application : ResultStaticsAdapter.kt

```
1   package com.amo.brainshield
2
3   import android.annotation.SuppressLint
4   import android.content.ActivityNotFoundException
```

```
5   import android.content.Context
6   import android.content.Intent
7   import android.net.Uri
8   import android.provider.Settings
9   import android.view.LayoutInflater
10  import android.view.View
11  import android.view.ViewGroup
12  import android.view.animation.AnimationUtils
13  import android.widget.ImageView
14  import android.widget.ProgressBar
15  import android.widget.RelativeLayout
16  import android.widget.TextView
17  import androidx.recyclerview.widget.RecyclerView
18  import java.util.*
19  // Adapter adapts the UI interface depending of the apps selected
20  class ResultStaticActivityAdapter(
21      private val mContext: Context,
22      private var mDataSet: ArrayList<AppInfo>,
23      private val homeFeed: HomeFeed
24  ) : RecyclerView.Adapter<ResultStaticActivityAdapter.ViewHolder>() {
25      private val bounceAnimationLogo = AnimationUtils.loadAnimation(mContext, R.anim.bounce_animat
            ion_logo)
26      private val bounceAnimationLogo_malware = AnimationUtils.loadAnimation(mContext, R.anim.boun
            ce_animation_logo_malware)
27
28
29      class ViewHolder @SuppressLint("WrongViewCast")
30      constructor(v: View) : RecyclerView.ViewHolder(v) {
31          var mTextViewLabel: TextView =
32              v.findViewById<View>(R.id.apkNameTextView) as TextView       // Get the widgets reference fr
                om custom layout
33          var mImageViewIcon: ImageView = v.findViewById<View>(R.id.packageImageView) as ImageVie
                w
34          var mItem: RelativeLayout = v.findViewById<View>(R.id.item) as RelativeLayout
35          var mImageViewResultIcon: ImageView = v.findViewById<View>(R.id.ImageResult) as ImageVie
                w
36          var mTextViewResultText: TextView =
37              v.findViewById<View>(R.id.apkPredictionTextView) as TextView       // Get the widgets referen
                ce from custom layout
38          var mProgressBar: ProgressBar = v.findViewById(R.id.progressBarResult)
39      }
40
41      // only called once
42      override fun onCreateViewHolder(
43          parent: ViewGroup,
44          viewType: Int
45      ): ViewHolder {
46          val v =
47              LayoutInflater.from(mContext).inflate(R.layout.layout_apk_item_result, parent, false)
48          return ViewHolder(v)
49      }
```

```
50
51      // hold application name, label, icon ect... to dynamic changed and printing prediction
52      override fun onBindViewHolder(holder: ViewHolder, position: Int) {
53          val icon = mDataSet[position].appIcon                   // Get the current app icon
54          val label = mDataSet[position].appName                  // Get the current app label
55          val resultText = mDataSet[position].appResultText
56          lateinit var resultPrediction: String
57          val application = homeFeed.application[position]
58          val packageName = mDataSet[position].appPackage                  // GET the current app name
59          holder.mTextViewLabel.text = packageName
60          holder.mImageViewIcon.setImageDrawable(icon)                  // Set the current app icon
61          holder.mItem.setOnClickListener {
62              holder.mImageViewResultIcon.startAnimation(bounceAnimationLogo)
63              holder.mImageViewIcon.startAnimation(bounceAnimationLogo)
64              holder.mTextViewLabel.startAnimation(bounceAnimationLogo)
65              holder.mTextViewResultText.startAnimation(bounceAnimationLogo)
66                  try {
67                      //Open the specific App Info page:
68                      val intent = Intent(Settings.ACTION_APPLICATION_DETAILS_SETTINGS)
69                      intent.data = Uri.parse("package:".plus(packageName!!))
70                      intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK
71                      mContext.startActivity(intent)
72
73                  } catch (e: ActivityNotFoundException) {
74                      val intent = Intent(Settings.ACTION_MANAGE_APPLICATIONS_SETTINGS)
75                      mContext.startActivity(intent)
76                  }
77          }
78          resultPrediction = application.prediction
79          if (resultPrediction == "Benign") {
80              holder.mImageViewResultIcon.setImageResource(R.drawable.benign_shield)
81              holder.mImageViewResultIcon.startAnimation(bounceAnimationLogo)
82  //          }
83              holder.mProgressBar.visibility = View.GONE
84          }
85          if (resultPrediction == "Malware") {
86              holder.mImageViewResultIcon.setImageResource(R.drawable.redalert)
87              holder.mImageViewResultIcon.startAnimation(bounceAnimationLogo_malware)
88              holder.mProgressBar.visibility = View.GONE
89          }
90
91          if (resultPrediction == "Uncertain") {
92              holder.mImageViewResultIcon.setImageResource(R.drawable.alert)
93              holder.mImageViewResultIcon.startAnimation(bounceAnimationLogo_malware)
94              holder.mProgressBar.visibility = View.GONE
95          }
96
97          holder.mTextViewResultText.text = resultText
98      }
99
100     override fun getItemCount(): Int {  // Count the installed apps
```

```
101         return mDataSet.size
102     }
103 }
```

Figure 5.8 Code application : AndroidManifest.xml

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3       xmlns:tools="http://schemas.android.com/tools"
4       package="com.amo.brainshield">
5
6       <uses-permission android:name="android.permission.INTERNET" />
7       <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
8
9       <application
10          android:allowBackup="true"
11          android:icon="@drawable/brainshield"
12          android:label="@string/app_name"
13          android:networkSecurityConfig="@xml/network_security_config"
14          android:roundIcon="@mipmap/ic_launcher_round"
15          android:supportsRtl="true"
16          android:theme="@style/AppTheme"
17          android:usesCleartextTraffic="true"
18          tools:ignore="GoogleAppIndexingWarning"
19          tools:targetApi="n"
20          android:fullBackupContent="true">
21          <activity android:name="com.amo.brainshield.SplashScreenActivity">
22              <intent-filter>
23                  <action android:name="android.intent.action.MAIN" />
24                  <category android:name="android.intent.category.LAUNCHER" />
25              </intent-filter>
26          </activity>
27          <activity android:name=".ResultStaticActivity">
28              <meta-data
29                  android:name="android.support.PARENT_ACTIVITY"
30                  android:value=".MainActivity" />
31          </activity>
32          <activity android:name=".MainActivity" />
33          <activity android:name="com.amo.brainshield.ResultStorageActivity">
34              <meta-data
35                  android:name="android.support.PARENT_ACTIVITY"
36                  android:value=".MainActivity" />
37              <intent-filter>
38                  <action android:name="android.intent.action.CALL" />
39                  <category android:name="android.intent.category.DEFAULT" />
40              </intent-filter>
41          </activity>
42
```

```
43        <activity android:name=".VirusTotalActivity">
44          <meta-data
45            android:name="android.support.PARENT_ACTIVITY"
46            android:value=".MainActivity" />
47        </activity>
48        <activity android:name=".ResultOfflineActivity">
49          <meta-data
50            android:name="android.support.PARENT_ACTIVITY"
51            android:value=".MainActivity" />
52        </activity>
53        <activity android:name=".ResultDynamicActivity">
54          <meta-data
55            android:name="android.support.PARENT_ACTIVITY"
56            android:value=".MainActivity" />
57        </activity>
58
59    </application>
60  </manifest>
```

Figure 5.9 Code application : build.gradle

```
1.  apply plugin: 'com.android.application'
2.
3.  apply plugin: 'kotlin-android'
4.
5.  apply plugin: 'kotlin-android-extensions'
6.
7.  android {
8.      compileSdkVersion 29
9.      buildToolsVersion "29.0.3"
10.     defaultConfig {
11.         ndk {
12.             abiFilters "armeabi-v7a", "x86"
13.         }
14.         applicationId 'com.amo.brainshield'
15.         minSdkVersion 19
16.         targetSdkVersion 29
17.         versionCode 1
18.         versionName "1.0"
19.         testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
20.     }
21.     buildTypes {
22.         release {
23.             minifyEnabled false
24.             proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
25.         }
26.         debug {
27.             debuggable true
28.             testCoverageEnabled true
29.         }
```

```
    }
    aaptOptions {
        noCompress "tflite"
        noCompress "flite"
    }
    compileOptions {
        sourceCompatibility = '1.8'
        targetCompatibility = '1.8'
    }
    lintOptions {
        abortOnError false
    }
}
dependencies {
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.core:core-ktx:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'androidx.recyclerview:recyclerview:1.1.0'
    implementation 'com.google.android.material:material:1.1.0'
    implementation 'androidx.recyclerview:recyclerview:1.1.0'
    implementation 'com.google.android.material:material:1.1.0'
    implementation 'org.tensorflow:tensorflow-lite:1.13.1'
    implementation 'org.tensorflow:tensorflow-lite-gpu:0.0.0-nightly'
    implementation 'org.tensorflow:tensorflow-lite-support:0.0.0-nightly'
    implementation 'org.tensorflow:tensorflow-lite'
    implementation 'org.tensorflow:tensorflow-android:1.13.1'
    implementation 'com.github.kittinunf.fuel:fuel:2.2.1'
    implementation 'com.github.kittinunf.fuel:fuel-android:2.2.1'
    implementation 'com.github.kittinunf.fuel:fuel-gson:2.2.1'
    implementation 'com.google.code.gson:gson:2.8.5'
}
```

ANNEXE J : CODE SERVEUR APP.PY

```
1.  import glob
2.  import os
3.  import shutil
4.  import time
5.  import json
6.  import numpy as np
7.  import pandas as pd
8.  import tensorflow.keras as keras
9.  from flask import Flask, request
10. from werkzeug.utils import secure_filename
11.
12. from Static3359 import GivingFeatureVectorStatic
13. from Dynamic3722 import GivingFeatureVectorDynamic
14.
15.
16. app = Flask(__name__)
17.
18. app.config["ALLOWED_FILE_EXTENSIONS"] = ["APK"]
19. app.config['SERVER_FOLDER'] = '/home/ubuntu/ServeurDep'
20. app.config['APK_FOLDER'] = '/home/ubuntu/ServeurDep/ApkFolder'
21. app.config['UPLOAD_FOLDER'] = app.config['APK_FOLDER'] + '/Upload_APK'
22. app.config['ANALYSIS_FOLDER'] = app.config['UPLOAD_FOLDER'] + '/Features_files'
23. app.config['DATABASE_ANALYSIS_FOLDER'] = app.config['APK_FOLDER'] + '/Database_Features_
        files'
24. app.config['DATABASE_ANALYSIS_FOLDER_DYNAMIC'] = app.config['APK_FOLDER'] + '/Datab
        ase_Features_files_dynamic'
25. app.config['DATABASE_ANALYSIS_FOLDER_VT'] = app.config['APK_FOLDER'] + '/Database_VT'
26. app.config['DATABASE_APK_FOLDER'] = app.config['APK_FOLDER'] + '/Database_APK'
27. app.config['VT_FOLDER'] = app.config['UPLOAD_FOLDER']  + '/VT_analysis'
28. app.config['ANDROPYTOOL_DROIDBOX'] = '/home/ubuntu/AndroPyTool/DroidBox_AndroPyTool'
29.
30.
31. def allowApkOnly(filename):
32.     if not "." in filename:
33.         return False
34.     ext = filename.rsplit(".", 1)[1]
35.     if ext.upper() in app.config["ALLOWED_FILE_EXTENSIONS"]:
36.         return True
37.     else:
38.         return False
39.
40. def decoupeString(myChain):
41.     myChain = myChain[1:len(myChain) - 1]
42.     ListString = []
43.     for boutDeChaine in myChain.split(", "):
44.         ListString.append(boutDeChaine)
45.     return ListString
46.
```

```
47.
48. def StringApkToJson(chain):
49.     prefix, rest = chain.split('.apk')
50.     return prefix + "json"
51.
52.
53. def StringflatToApk(chain):
54.     return chain + ".apk"
55.
56. def numberOfServiceMalware():
57.     os.chdir(app.config['VT_FOLDER'])
58.     compteurBenign = 0
59.     compteurMalware = 0
60.     counter = 0
61.     all_filenames = [i for i in glob.glob('*'.format("json"))]
62.     listCounterGlobal = [103]
63.     listCounter = []
64.     for file in all_filenames:
65.         UneApp = {}
66.         with open(app.config['VT_FOLDER'] + '/' + file, 'r') as f:
67.             distros_dict = json.load(f)
68.             scans = distros_dict['scans']
69.         listOfScanService = []
70.         for service in scans:
71.             listOfScanService.append(service)
72.
73.         for test in listOfScanService:
74.             if not scans[test]['detected']:
75.                 compteurBenign += 1
76.             else:
77.                 compteurMalware += 1
78.
79.         UneApp["packageName"] = all_filenames[counter]
80.         UneApp["counterPrediction"] = str(compteurMalware)
81.
82.         listCounter.append(UneApp)
83.         listCounterGlobal["application"] = listCounter
84.         counter = counter + 1
85.
86.     return listCounterGlobal
87.
88. @app.route('/', methods=['GET', 'POST'])
89. def index():
90.     return "Index"
91.
92. @app.route('/IsAPKExist', methods=['GET', 'POST'])
93. def IsAPKExist():
94.     body = request.data
95.     bodyProcessed = decoupeString(body)
96.     ApkListNeedToBeSendToServer = []
97.     if os.path.isdir(app.config['UPLOAD_FOLDER']):
```

```
98.         print("")
99.     else:
100.        os.mkdir(app.config['UPLOAD_FOLDER'])
101.    for package in bodyProcessed:
102.        filename = StringflatToApk(package)
103.        if os.path.isfile(app.config['DATABASE_APK_FOLDER'] + '/' + filename):
104.            shutil.copyfile(app.config['DATABASE_APK_FOLDER'] + '/' + filename,
105.                            app.config['UPLOAD_FOLDER'] + '/' + filename)
106.        else:
107.            ApkListNeedToBeSendToServer.append(package)
108.    return str(ApkListNeedToBeSendToServer)
109.
110.
111. @app.route('/IsSelectedAPKExist', methods=['GET', 'POST'])
112. def IsSelectedAPKExist():
113.    filename = request.data
114.    ApkListNeedToBeSendToServer = []
115.    if os.path.isdir(app.config['UPLOAD_FOLDER']):
116.        print("")
117.    else:
118.        os.mkdir(app.config['UPLOAD_FOLDER'])
119.
120.    if os.path.isfile(app.config['DATABASE_APK_FOLDER'] + '/' + filename):
121.        shutil.copyfile(app.config['DATABASE_APK_FOLDER'] + '/' + filename,
122.                        app.config['UPLOAD_FOLDER'] + '/' + filename)
123.    else:
124.        ApkListNeedToBeSendToServer.append(filename)
125.    print("NEED TO UPLOAD" + str(ApkListNeedToBeSendToServer))
126.    return str(ApkListNeedToBeSendToServer)
127.
128. @app.route('/transferFiles', methods=['GET', 'POST'])
129. def transferFiles():
130.    compteur = 0
131.
132.    if request.method == "POST":
133.        if request.files:
134.            apk = request.files['apk']
135.            filename = secure_filename(apk.filename)
136.            if allowApkOnly(filename):
137.                os.chdir(app.config['UPLOAD_FOLDER'])
138.
139.                if os.path.isfile(app.config['UPLOAD_FOLDER'] + '/' + filename):
140.                    files = [i for i in glob.glob('*.{}'.format("apk"))]
141.                    for file in files:
142.                        compteur = compteur + 1
143.                    return str(compteur)
144.                else:
145.                    apk.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
146.                    shutil.copyfile(app.config['UPLOAD_FOLDER'] + '/' + filename, app.config['DATABASE_
                APK_FOLDER'] + '/' + filename)
147.
```

```
148.                 print("SAVED : " + filename)
149.                 files = [i for i in glob.glob('*.{}'.format("apk"))]
150.                 for file in files:
151.                     compteur = compteur + 1
152.
153.                 return str(compteur)
154.         else:
155.             return "No file in the request"
156.
157. @app.route('/FeatureExtraction', methods=['GET', 'POST'])
158. def FeatureExtraction():
159.     os.chdir(app.config['UPLOAD_FOLDER'])
160.
161.     if os.path.isdir(app.config['ANALYSIS_FOLDER']) == 0:
162.         os.mkdir(app.config['ANALYSIS_FOLDER'])
163.         print("Folder created : " + app.config['ANALYSIS_FOLDER'])
164.     else:
165.         print("Folder " + app.config['ANALYSIS_FOLDER'] + " already exist")
166.
167.     files = [i for i in glob.glob('*.{}'.format("apk"))]
168.
169.     for file in files:
170.         JsonFile = StringApkToJson(file)
171.         if os.path.isfile(app.config['DATABASE_ANALYSIS_FOLDER'] + "/" + JsonFile):
172.
173.             shutil.copyfile(app.config['DATABASE_ANALYSIS_FOLDER'] + "/" + JsonFile, app.config['A
                 NALYSIS_FOLDER'] + '/' + JsonFile)
174.
175.     os.system("python /home/ubuntu/AndroPyTool/androPyTool.py -
                 s " + app.config['UPLOAD_FOLDER'])
176.     os.chdir(app.config['ANALYSIS_FOLDER'])
177.     feature_files = [i for i in glob.glob('*.format("json"))]
178.     for analysis in feature_files:
179.         if not os.path.isfile(app.config['DATABASE_ANALYSIS_FOLDER'] + "/" + analysis):
180.             shutil.copyfile(app.config['ANALYSIS_FOLDER'] + "/" + analysis, app.config['DATABASE_AN
                 ALYSIS_FOLDER'] + '/' + analysis)
181.
182.     return "Feature Extraction Done"
183.
184. @app.route('/prediction', methods=['GET', 'POST'])
185. def prediction():
186.     global vectors
187.     model = keras.models.load_model('/home/ubuntu/ServeurDep/StaticModel.h5')
188.     extension = 'json'
189.     os.chdir(app.config['ANALYSIS_FOLDER'])
190.     all_filenames = [i for i in glob.glob('*.{}'.format(extension))]
191.     all_filenames.sort(key=str.lower)
192.     all_filenamesString = []
193.     counter = 0
194.     ListPredictionGlobal = {}
195.     ListPrediction = []
```

```
196.     try:
197.         vectors = GivingFeatureVectorStatic()
198.     except OSError as e:
199.         print("Error: %s - %s." % (e.filename, e.strerror))
200.
201.
202.     for featureVectorApplication in vectors:
203.         UnePrediction = {}
204.
205.         vectorArray = np.array([featureVectorApplication])
206.         vectorPanda = pd.DataFrame(vectorArray)
207.
208.         Prediction = (model.predict(vectorPanda)[0])[0]
209.
210.         if Prediction > 0.6:
211.             StringResult = 'Malware'
212.             StringPrediction = '%.0f' % (Prediction * 100)
213.
214.         elif Prediction > 0.4:
215.             StringResult = 'Uncertain'
216.             StringPrediction = '%.0f' % (Prediction * 100)
217.
218.         else:
219.             StringResult = 'Benign'
220.             StringPrediction = '%.0f' % (100 - Prediction * 100)
221.
222.         UnePrediction["packageName"] = all_filenames[counter]
223.         UnePrediction["prediction"] = str(StringResult)
224.         UnePrediction["predictionNumber"] = str(StringPrediction)
225.         ListPrediction.append(UnePrediction)
226.         ListPredictionGlobal["application"] = ListPrediction
227.
228.         counter = counter + 1
229.
230.     if os.path.isdir(app.config['UPLOAD_FOLDER']):
231.         shutil.rmtree(app.config['UPLOAD_FOLDER'])
232.     return ListPredictionGlobal
233.
234. @app.route('/vt', methods=['GET', 'POST'])
235. def virusTotal():
236.     os.chdir(app.config['UPLOAD_FOLDER'])
237.
238.     if os.path.isdir(app.config['ANALYSIS_FOLDER']) == 0:
239.         os.mkdir(app.config['ANALYSIS_FOLDER'])
240.         print("Folder created : " + app.config['ANALYSIS_FOLDER'])
241.     else:
242.         print("Folder " + app.config['ANALYSIS_FOLDER'] + " already exist")
243.
244.     if os.path.isdir(app.config['VT_FOLDER']) == 0:
245.         os.mkdir(app.config['VT_FOLDER'])
246.         print("Folder created : " + app.config['VT_FOLDER'])
```

```
247.     else:
248.         print("Folder " + app.config['VT_FOLDER'] + " already exist")
249.
250.     files = [i for i in glob.glob('*.{}'.format("apk"))]
251.
252.     for file in files:
253.         JsonFile = StringApkToJson(file)
254.         if os.path.isfile(app.config['DATABASE_ANALYSIS_FOLDER'] + "/" + JsonFile):
255.             shutil.copyfile(app.config['DATABASE_ANALYSIS_FOLDER'] + "/" + JsonFile,
256.                             app.config['ANALYSIS_FOLDER'] + '/' + JsonFile)
257.     os.chdir(app.config['UPLOAD_FOLDER'])
258.     vt_files = [i for i in glob.glob('*.{}'.format("apk"))]
259.     for file in vt_files:
260.         JsonFile = StringApkToJson(file)
261.
262.         if os.path.isfile(app.config['DATABASE_ANALYSIS_FOLDER_VT'] + "/" + JsonFile):
263.             shutil.copyfile(app.config['DATABASE_ANALYSIS_FOLDER_VT'] + "/" + JsonFile,
264.                             app.config['VT_FOLDER'] + '/' + JsonFile)
265.
266.     os.chdir(app.config['VT_FOLDER'])
267.     vt_files = [i for i in glob.glob('*.{}'.format("json"))]
268.     for file in vt_files:
269.
270.         if not os.path.isfile(app.config['DATABASE_ANALYSIS_FOLDER_VT'] + "/" + file):
271.             shutil.copyfile(app.config['VT_FOLDER'] + "/" + file,
272.                             app.config['DATABASE_ANALYSIS_FOLDER_VT'] + '/' + file
273.     os.chdir(app.config['VT_FOLDER'])
274.     vt_files = [i for i in glob.glob('*.{}'.format("json"))]
275.     vt_files.sort(key=str.lower)
276.     counter = 0
277.     ListPredictionGlobal = {}
278.     ListPrediction = []
279.     vectors = numberOfServiceMalware()
280.     if os.path.isdir(app.config['UPLOAD_FOLDER']):
281.       shutil.rmtree(app.config['UPLOAD_FOLDER'])
282.     return vectors
283.
284. @app.route('/filter', methods=['GET', 'POST'])
285. def filter():
286.     os.chdir(app.config['UPLOAD_FOLDER'])
287.     os.system("python /home/ubuntu/AndroPyTool/androPyTool.py -
              s " + app.config['UPLOAD_FOLDER'] + " -f")
288.     return "filter"
289.
290. @app.route('/vtcl', methods=['GET', 'POST'])
291. def vtcl():
292.     os.chdir(app.config['UPLOAD_FOLDER'])
293.     os.system("python /home/ubuntu/AndroPyTool/androPyTool.py -
              s " + app.config['UPLOAD_FOLDER'] + " -f -
              vt e7999f9c6490c8c9105c75f103df5c296094bdd13e473a57868243107f328cb3 -cl")
294.     return "vtcl"
```

```
295.
296. @app.route('/FeatureExtractionDynamic', methods=['GET', 'POST'])
297. def FeatureExtractionDynamic():
298.     os.chdir(app.config['UPLOAD_FOLDER'])
299.
300.     if os.path.isdir(app.config['ANALYSIS_FOLDER']) == 0:
301.         os.mkdir(app.config['ANALYSIS_FOLDER'])
302.     else:
303.         print("Folder " + app.config['ANALYSIS_FOLDER'] + " already exist")
304.
305.     files = [i for i in glob.glob('*.{}'.format("apk"))]
306.
307.     for file in files:
308.         JsonFile = StringApkToJson(file)
309.         if os.path.isfile(app.config['DATABASE_ANALYSIS_FOLDER_DYNAMIC'] + "/" + JsonFile):
310.             shutil.copyfile(app.config['DATABASE_ANALYSIS_FOLDER_DYNAMIC'] + "/" + JsonFile, a
                 pp.config['ANALYSIS_FOLDER'] + "/" + JsonFile)
311.         else:
312.             print(JsonFile + " NOT EXISTING")
313.
314.     os.system("python " + app.config['ANDROPYTOOL_DROIDBOX'] + "/fork_droidbox.py -
              s " + app.config['UPLOAD_FOLDER'] + " -d 300 -
              o " + app.config['ANALYSIS_FOLDER'] + "/ --no-gui" )
315.     return "Feature Extraction Done"
316.
317.
318. @app.route('/predictionDynamic', methods=['GET', 'POST'])
319. def predictionDynamic():
320.     global vectors
321.     model = keras.models.load_model('/home/ubuntu/ServeurDep/DynamicModel.h5')
322.     extension = 'json'
323.     os.chdir(app.config['ANALYSIS_FOLDER'])
324.     all_filenames = [i for i in glob.glob('*.{}'.format(extension))]
325.     for analysis in all_filenames:
326.         if os.path.isfile(app.config['DATABASE_ANALYSIS_FOLDER_DYNAMIC'] + "/" + analysis):
327.             print(analysis + " EXIST ")
328.         else:
329.             shutil.copyfile(app.config['ANALYSIS_FOLDER'] + "/" + analysis, app.config['DATABASE_AN
                 ALYSIS_FOLDER_DYNAMIC'] + '/' + analysis)
330.
331.     all_filenamesString = []
332.     counter = 0
333.     ListPredictionGlobal = {}
334.     ListPrediction = []
335.     try:
336.         vectors = GivingFeatureVectorDynamic()
337.     except ValueError as e:
338.         print("Hello")
339.
340.     for featureVectorApplication in vectors:
341.         UnePrediction = {}
```

```
342.         numpylist = [0] * 3722
343.         if featureVectorApplication == numpylist:
344.             print("app not predictable")
345.             StringResult = 'NoPredictable'
346.             StringPrediction = '%.0f' % 100
347.             UnePrediction["packageName"] = all_filenames[counter]
348.             UnePrediction["prediction"] = str(StringResult)
349.             UnePrediction["predictionNumber"] = str(StringPrediction)
350.             ListPrediction.append(UnePrediction)
351.             ListPredictionGlobal["application"] = ListPrediction
352.             counter = counter + 1
353.         else:
354.             vectorArray = np.array([featureVectorApplication])
355.             vectorPanda = pd.DataFrame(vectorArray)
356.             Prediction = (model.predict(vectorPanda)[0])[0]
357.             if Prediction > 0.9:
358.                 StringResult = 'Malware'
359.                 StringPrediction = '%.0f' % (Prediction * 100)
360.             elif Prediction > 0.7:
361.                 StringResult = 'Uncertain'
362.                 StringPrediction = '%.0f' % (Prediction * 100)
363.             else:
364.                 StringResult = 'Benign'
365.                 StringPrediction = '%.0f' % (100 - Prediction * 100)
366.             UnePrediction["packageName"] = all_filenames[counter]
367.             UnePrediction["prediction"] = str(StringResult)
368.             UnePrediction["predictionNumber"] = str(StringPrediction)
369.             ListPrediction.append(UnePrediction)
370.             ListPredictionGlobal["application"] = ListPrediction
371.             counter = counter + 1
372.     for file in all_filenames:
373.         if os.path.isfile(app.config['DATABASE_ANALYSIS_FOLDER'] + "/" + file):
374.             os.remove(app.config['ANALYSIS_FOLDER'] + "/" + file)
375.         else:
376.             shutil.move(app.config['ANALYSIS_FOLDER'] + "/" + file, app.config['DATABASE_ANALYSI
                S_FOLDER_DYNAMIC'])
377.     if os.path.isdir(app.config['UPLOAD_FOLDER']):
378.         shutil.rmtree(app.config['UPLOAD_FOLDER'])
379.     return ListPredictionGlobal
380.
381. if __name__ == '__main__':
382.     app.run(host='0.0.0.0', port=5000)
```

ANNEXE K : COURBES D'APPRENTISSAGES SUPPLÉMENTAIRES

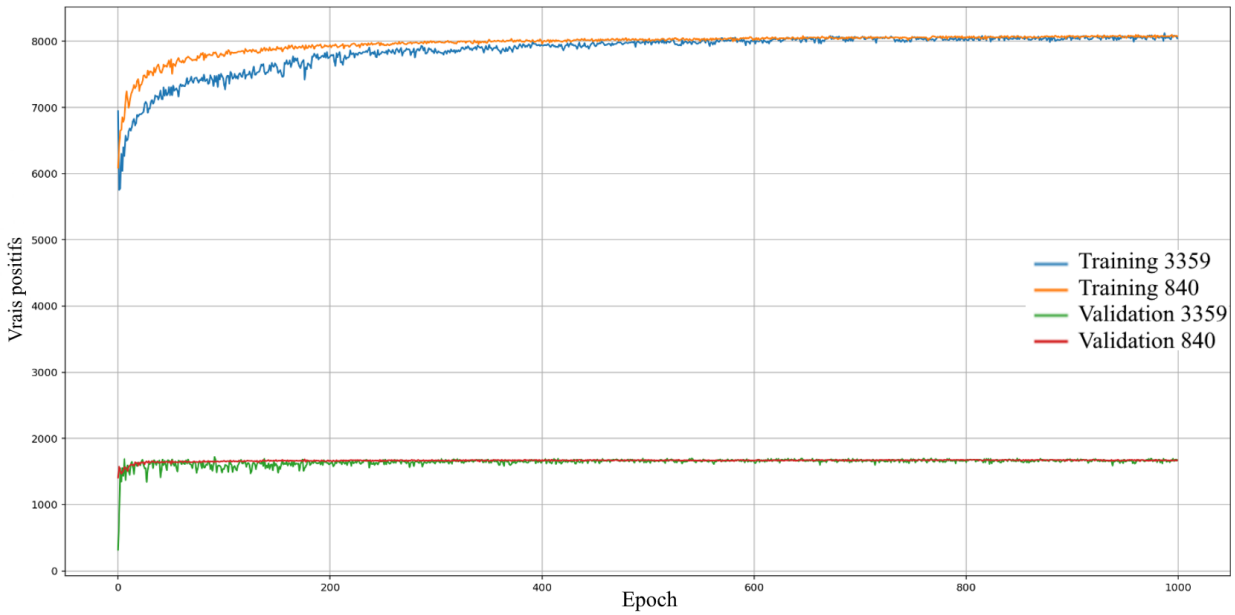


Figure 5.10 Courbe d'apprentissage : vrais positifs

nous observons une légère augmentation du nombre de vrais positifs sur l'ensemble d'entraînement en début d'apprentissage seulement. Cependant, il n'y a aucune variation sur l'ensemble de validation.

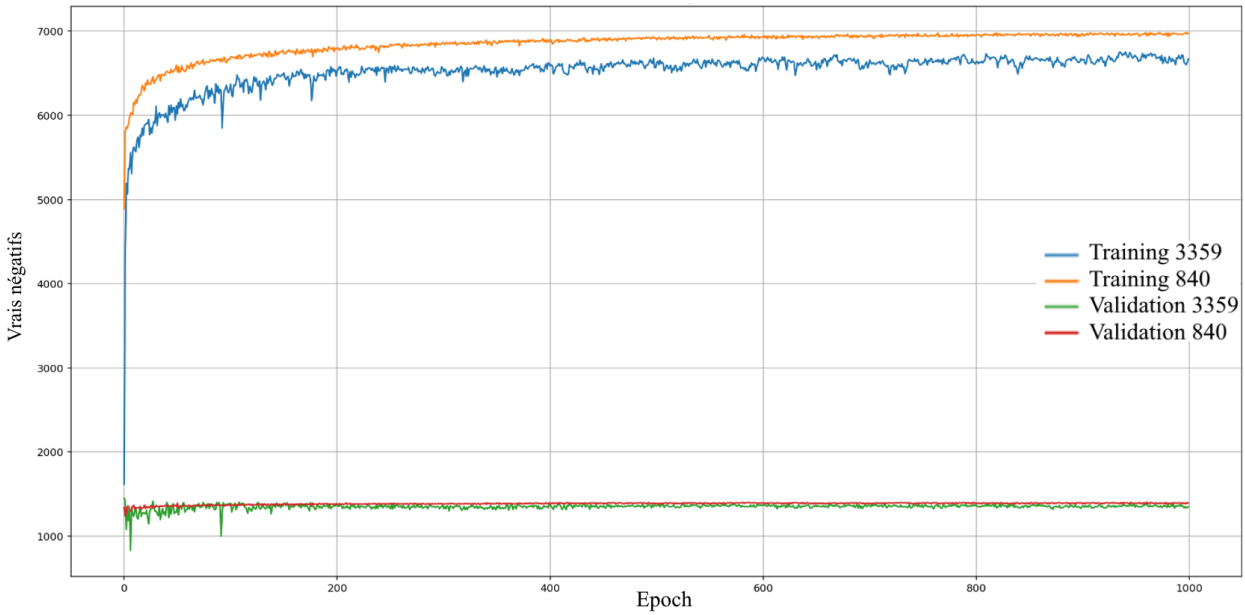


Figure 5.11 Courbe d'apprentissage : vrais négatifs

nous observons une légère augmentation du nombre de vrais négatifs sur l'ensemble d'entraînement dès le début et tout au long de l'apprentissage. Cependant, il n'y a aucune variation sur l'ensemble de validation.

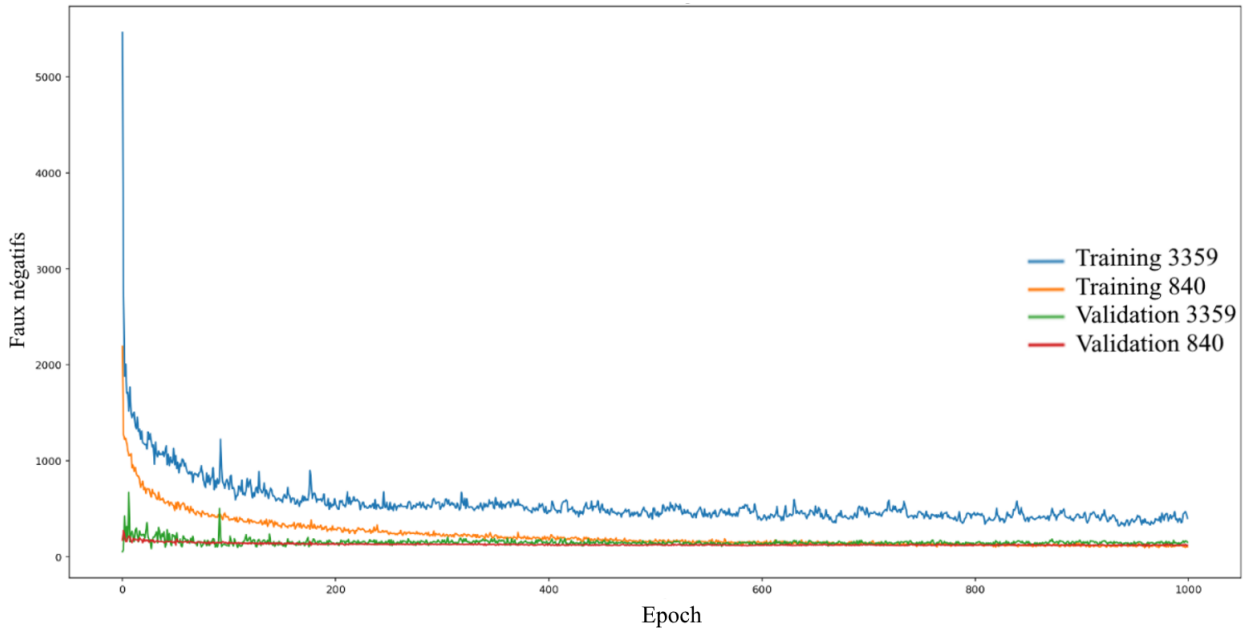


Figure 5.12 Courbe d'apprentissage : faux négatifs

nous observons une nette diminution du nombre de vrais positifs sur l'ensemble d'entraînement dès le début et tout au long de l'apprentissage. Cependant, il n'y a aucune variation sur l'ensemble de validation à l'exception d'une diminution du bruit.

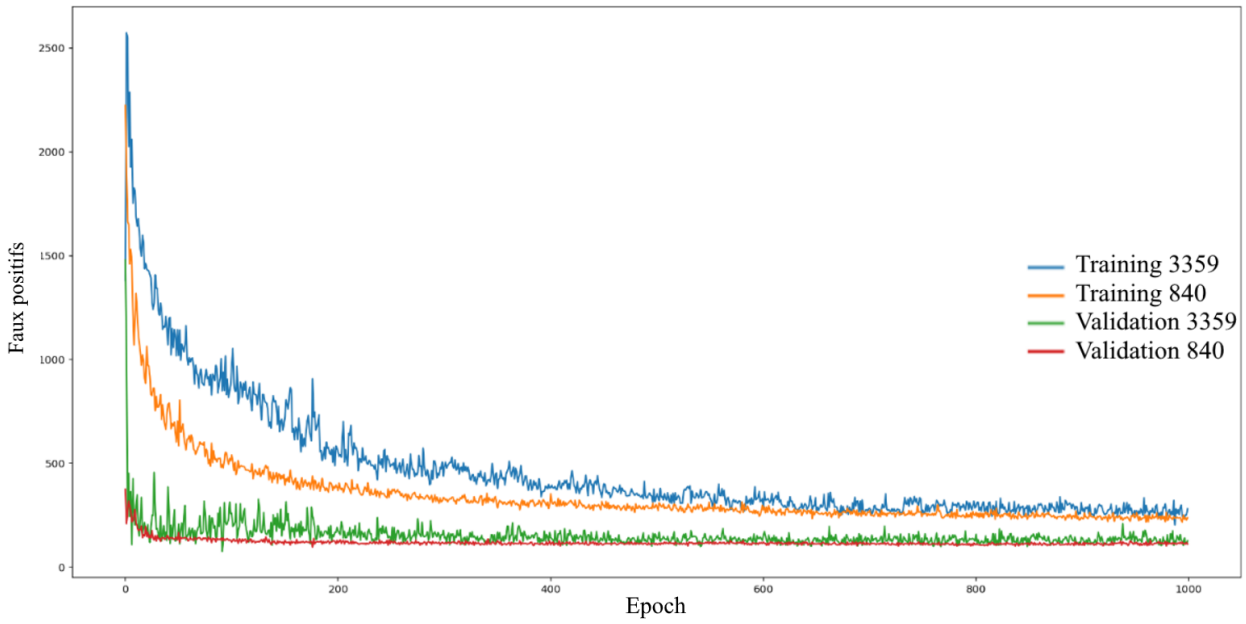


Figure 5.13 Courbes d'apprentissage : faux positifs

nous observons une nette diminution du nombre de faux positifs sur l'ensemble d'entraînement dès le début. Cependant, nous observons que plus l'on tend vers la fin de l'entraînement et plus le modèle 3,359 tend vers le modèle 840. Il n'y a aucune variation sur l'ensemble de validation à l'exception d'une diminution du bruit. Nous en déduisons que le modèle 840 apprend plus vite que le modèle 3,359 mais arrive sensiblement aux mêmes résultats si l'on tend vers 1000 itérations.

ANNEXE L : LISTE DES 840 CARACTÉRISTIQUES STATIQUES

1 PERMISSION-android.permission.READ_PHONE_STATE
2 PERMISSION-android.permission.SEND_SMS
3 PERMISSION-com.android.launcher.permission.INSTALL_SHORTCUT
4 PERMISSION-android.permission.RECEIVE_SMS
5 PERMISSION-android.permission.GET_TASKS
6 PERMISSION-android.permission.SYSTEM_ALERT_WINDOW
7 RECEIVER-adv.notify.click
8 RECEIVER-ru.justreader.beta.prefix.SyncEvent
9 PERMISSION-android.permission.READ_SMS
10 PERMISSION-android.permission.WRITE_EXTERNAL_STORAGE
11 PERMISSION-android.permission.ACCESS_WIFI_STATE
12 PERMISSION-android.permission.ACCESS_WIFI_STATE.1
13 PERMISSION-android.permission.WRITE_SMS
14 APICALL-android.telephony.SmsManager
15 APICALL-android.telephony.TelephonyManager
16 PERMISSION-android.permission.RECEIVE_BOOT_COMPLETED
17 PERMISSION-android.permission.RESTART_PACKAGES
18 RECEIVER-ru.vddevelopment.ref.enswen.free.WordAppWidgetProvider.OPEN_WORD
19 PERMISSION-android.permission.MOUNT_UNMOUNT_FILESYSTEMS
20 APICALL-android.telephony.SmsMessage
21 FLOWDROID-SMS_MMS-NO_CATEGORY
22 PERMISSION-android.permission.ACCESS_NETWORK_STATE
23 PERMISSION-android.permission.ACCESS_NETWORK_STATE.1
24 RECEIVER-com.qihoo360.antilostwatch.elder.ACTION_UPDATE_HAS_NEW_VERSION
25 PERMISSION-android.permission.ACCESS_COARSE_LOCATION
26 APICALL-android.hardware.SensorManager
27 RECEIVER-com.cursbnr.ro.SelectMonedaForWidget.REFRESH_ALL_WIDGET
28 APICALL-android.gesture.GestureStore
29 RECEIVER-rpa.com.excelliance.kxqp.platform.gameplugin2.ReceiverProxy_SP_D
30 FLOWDROID-SMS_MMS-IPC
31 PERMISSION-android.permission.INSTALL_PACKAGES
32 FLOWDROID-FILE-NETWORK_INFORMATION
33 FLOWDROID-NOT_EXISTING-UNIQUE_IDENTIFIER
34 PERMISSION-android.permission.CHANGE_WIFI_STATE
35 PERMISSION-com.android.launcher.permission.UNINSTALL_SHORTCUT
36 PERMISSION-android.permission.WRITE_APN_SETTINGS
37 FLOWDROID-DATABASE_INFORMATION-UNIQUE_IDENTIFIER
38 FLOWDROID-FILE-NETWORK
39 PERMISSION-android.permission.READ_LOGS
40 FLOWDROID-IPC-NOT_EXISTING
41 PERMISSION-android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
42 APICALL-android.telephony.gsm.SmsManager
43 APICALL-android.view.animation.TranslateAnimation
44 FLOWDROID-SMS_MMS-UNIQUE_IDENTIFIER
45 APICALL-android.gesture.GestureOverlayView
46 APICALL-android.telephony.gsm.GsmCellLocation
47 FLOWDROID-SMS_MMS-NOT_EXISTING
48 PERMISSION-android.permission.RECEIVE_WAP_PUSH
49 PERMISSION-android.permission.PROCESS_OUTGOING_CALLS
50 FLOWDROID-LOG-LOCATION_INFORMATION
51 RECEIVER-com.aoe.action.WAKEUP_APP_REBIND
52 RECEIVER-com.igexin.sdk.action.etVyv7RHJ28BqN3KKCtKJ6
53 APICALL-android.content.res.AssetManager
54 APICALL-android.graphics.Camera
55 PERMISSION-android.permission.CHANGE_NETWORK_STATE
56 FLOWDROID-NO_CATEGORY-NETWORK
57 PERMISSION-android.permission.RECEIVE_MMS
58 PERMISSION-android.permission.ACCESS_FINE_LOCATION
59 FLOWDROID-LOG-UNIQUE_IDENTIFIER
60 FLOWDROID-SMS_MMS-BUNDLE
61 RECEIVER-droom.sleepIfUCan.pro.alarm_killed
62 FLOWDROID-DATABASE_INFORMATION-NOT_EXISTING
63 APICALL-android.location.Criteria
64 FLOWDROID-NO_CATEGORY-UNIQUE_IDENTIFIER
65 PERMISSION-android.permission.MODIFY_PHONE_STATE
66 APICALL-java.io.BufferedReader
67 PERMISSION-android.permission.DELETE_PACKAGES
68 APICALL-android.net.Proxy
69 FLOWDROID-IPC-UNIQUE_IDENTIFIER
70 RECEIVER-com.baidu.searchbox.action.DING_UPDATE
71 SYSTEMCOMMAND-chmod
72 APICALL-android.location.LocationManager
73 APICALL-android.graphics.drawable.AnimatedVectorDrawable
74 PERMISSION-android.permission.WRITE_SETTINGS
75 PERMISSION-android.permission.CALL_PHONE
76 APICALL-android.net.wifi.WifiManager
77 APICALL-javax.microedition.khronos.opengles.GL10
78 RECEIVER-com.hearst.widget.service.update
79 RECEIVER-android.action.removeremind.five.minute
80 APICALL-android.media.browse.MediaBrowser
81 FLOWDROID-SMS_MMS-NETWORK_INFORMATION
82 APICALL-android.graphics.drawable.ClipDrawable
83 FLOWDROID-NETWORK-UNIQUE_IDENTIFIER
84 APICALL-java.security.InvalidKeyException
85 FLOWDROID-LOG-NETWORK
86 APICALL-javax.crypto.IllegalBlockSizeException
87 APICALL-android.telephony.gsm.SmsMessage
88 PERMISSION-droid.permission.INSTALL_PACKAGES
89 PERMISSION-android.intent.action.BOOT_COMPLETED
90 PERMISSION-android.permission.WRITE_SECURE_SETTINGS
91 PERMISSION-android.permission.SYSTEM_OVERLAY_WINDOW
92 APICALL-android.net.wifi.WifiInfo
93 FLOWDROID-NOT_EXISTING-NOT_EXISTING
94 PERMISSION-READ_PHONE_STATE
95 APICALL-android.opengl.GLU
96 PERMISSION-android.permission.CALL_PRIVILEGED
97 FLOWDROID-IPC-BUNDLE
98 APICALL-java.security.NoSuchAlgorithmException
99 APICALL-android.widget.ViewFlipper

100 APICALL-javax.crypto.NoSuchPaddingException
101 PERMISSION-android.permission.DISABLE_KEYGUARD
102 APICALL-java.net.MalformedURLException
103 FLOWDROID-DATABASE_INFORMATION-LOCATION_INFORMATION
104 FLOWDROID-IPC-NO_CATEGORY
105 APICALL-android.telephony.cdma.CdmaCellLocation
106 PERMISSION-com.android.browser.permission.WRITE_HISTORY_BOOKMARKS
107 PERMISSION-android.permission.BAIDU_LOCATION_SERVICE
108 PERMISSION-com.software.application.permission.C2D_MESSAGE
109 PERMISSION-android.permission.WAKE_LOCK
110 APICALL-android.widget.ImageSwitcher
111 APICALL-java.io.IOException
112 FLOWDROID-NOT_EXISTING-NETWORK_INFORMATION
113 APICALL-org.xml.sax.helpers.DefaultHandler
114 PERMISSION-com.android.launcher.permission.READ_SETTINGS
115 RECEIVER-com.teslacoilsw.widgetlocker.DISABLE
116 APICALL-android.webkit.WebIconDatabase
117 FLOWDROID-NETWORK-NETWORK_INFORMATION
118 RECEIVER-android.alarm.timing.action.3
119 APICALL-android.support.graphics.drawable.VectorDrawableCompat
120 APICALL-android.support.graphics.drawable.AnimatedVectorDrawableCompat
121 FLOWDROID-IPC-NETWORK_INFORMATION
122 APICALL-javax.microedition.khronos.opengles.GL11Ext
123 PERMISSION-com.android.browser.permission.READ_HISTORY_BOOKMARKS
124 APICALL-org.xml.sax.ext.DeclHandler
125 PERMISSION-android.permission.RECEIVE_USER_PRESENT
126 APICALL-java.lang.ReflectiveOperationException
127 APICALL-java.net.JarURLConnection
128 APICALL-android.widget.Gallery
129 PERMISSION-android.permission.ACCESS_MOCK_LOCATION
130 APICALL-java.util.concurrent.ThreadLocalRandom
131 PERMISSION-android.permission.BROADCAST_SMS
132 APICALL-android.telephony.NeighboringCellInfo
133 APICALL-android.support.v4.app.BundleCompat
134 APICALL-android.telephony.PhoneStateListener
135 APICALL-android.content.pm.ApplicationInfo
136 APICALL-java.nio.ByteOrder
137 APICALL-java.util.zip.ZipInputStream
138 APICALL-java.util.Timer
139 APICALL-android.telephony.SignalStrength
140 FLOWDROID-NETWORK-NOT_EXISTING
141 APICALL-android.graphics.NinePatch
142 APICALL-android.location.LocationListener
143 APICALL-android.support.customtabs.CustomTabsClient
144 APICALL-android.webkit.WebView
145 APICALL-android.graphics.drawable.shapes.Shape
146 FLOWDROID-FILE-NO_CATEGORY
147 PERMISSION-android.webkit.permission.PLUGIN
148 APICALL-org.xml.sax.ext.LexicalHandler
149 APICALL-android.app.admin.DeviceAdminReceiver
150 FLOWDROID-NO_CATEGORY-NETWORK_INFORMATION
151 FLOWDROID-NETWORK-LOCATION_INFORMATION
152 FLOWDROID-NETWORK-NO_CATEGORY
153 APICALL-org.xml.sax.AttributeList
154 FLOWDROID-NOT_EXISTING-LOCATION_INFORMATION
155 OPCODE-sput-byte
156 APICALL-javax.crypto.BadPaddingException
157 FLOWDROID-LOG-NETWORK_INFORMATION
158 PERMISSION-android.permission.WRITE_CALL_LOG
159 APICALL-android.graphics.drawable.StateListDrawable
160 APICALL-java.lang.ClassNotFoundException
161 APICALL-java.io.UnsupportedEncodingException
162 FLOWDROID-NOT_EXISTING-NETWORK
163 FLOWDROID-IPC-ACCOUNT_INFORMATION
164 APICALL-android.provider.Browser
165 APICALL-android.support.design.widget.BottomSheetDialogFragment
166 APICALL-org.xml.sax.helpers.LocatorImpl
167 APICALL-android.webkit.WebChromeClient
168 FLOWDROID-FILE-BUNDLE
169 APICALL-org.xml.sax.Attributes
170 RECEIVER-com.igexin.sdk.action.cIRZ7PvDXY8zqOgEZSBGm1
171 PERMISSION-android.permission.KILL_BACKGROUND_PROCESSES
172 FLOWDROID-LOG-NOT_EXISTING
173 SYSTEMCOMMAND-logcat
174 PERMISSION-com.motorola.dlauncher.permission.READ_SETTINGS
175 PERMISSION-android.permission.READ_CALL_LOG
176 APICALL-org.xml.sax.InputSource
177 RECEIVER-com.tapanifinal.clockturlington.alarm1turlington.ALARM_ALERT
178 APICALL-android.support.design.widget.BottomSheetDialog
179 PERMISSION-android.permission.CHANGE_COMPONENT_ENABLED_STATE
180 PERMISSION-com.motorola.launcher.permission.READ_SETTINGS
181 APICALL-javax.net.ssl.SSLPeerUnverifiedException
182 APICALL-java.util.jar.JarFile
183 APICALL-java.lang.SecurityException
184 APICALL-android.graphics.drawable.GradientDrawable
185 PERMISSION-android.permission.ACCESS_COARSE_UPDATES
186 APICALL-java.lang.Exception
187 APICALL-java.util.zip.ZipEntry
188 PERMISSION-android.permission.SET_WALLPAPER
189 FLOWDROID-SMS_MMS-DATABASE_INFORMATION
190 APICALL-android.widget.RelativeLayout
191 APICALL-org.xml.sax.helpers.XMLFilterImpl
192 APICALL-javax.microedition.khronos.egl.EGLContext
193 APICALL-java.util.TimerTask
194 PERMISSION-android.permission.READ_CONTACTS
195 APICALL-android.opengl.GLUtils
196 APICALL-android.support.v4.app.FragmentManagerNonConfig
197 APICALL-org.apache.http.params.HttpConnectionParams
198 APICALL-java.io.FileReader
199 APICALL-java.nio.IntBuffer
200 APICALL-android.support.v4.os.BuildCompat
201 APICALL-android.view.animation.AnimationSet

202 APICALL-android.widget.MediaController
203 PERMISSION-com.motorola.launcher.permission.INSTALL_SHORTCUT
204 PERMISSION-android.hardware.camera.autofocus
205 APICALL-android.service.media.MediaBrowserService
206 PERMISSION-com.lge.launcher.permission.INSTALL_SHORTCUT
207 PERMISSION-com.motorola.dlauncher.permission.INSTALL_SHORTCUT
208 APICALL-android.service.wallpaper.WallpaperService
209 RECEIVER-com.zhangyue.iReader.push.remove
210 APICALL-javax.crypto.SecretKeyFactory
211 APICALL-org.xml.sax.XMLReader
212 APICALL-android.webkit.WebResourceError
213 APICALL-javax.crypto.CipherOutputStream
214 APICALL-android.widget.VideoView
215 APICALL-android.hardware.Sensor
216 SERVICE-com.snowfish.a.a.s.ABGSvc
217 APICALL-java.nio.MappedByteBuffer
218 PERMISSION-com.lge.launcher.permission.READ_SETTINGS
219 PERMISSION-android.permission.MOUT_UNMOUNT_FILESYSTEMS
220 APICALL-javax.microedition.khronos.opengles.GL11
221 APICALL-org.xml.sax.Parser
222 APICALL-android.view.ViewManager
223 APICALL-org.xmlpull.v1.XmlPullParserFactory
224 RECEIVER-com.android.advertTF.action.image2
225 PERMISSION-org.adw.launcher.permission.READ_SETTINGS
226 APICALL-android.graphics.drawable.DrawableContainer
227 SERVICE-com.android.jservice.UpdateService
228 APICALL-java.util.zip.CheckedInputStream
229 APICALL-android.view.PointerIcon
230 PERMISSION-android.permission.GET_PACKAGE_SIZE
231 PERMISSION-com.fede.launcher.permission.READ_SETTINGS
232 APICALL-java.nio.FloatBuffer
233 APICALL-javax.xml.transform.sax.SAXResult
234 APICALL-java.io.FilterWriter
235 PERMISSION-android.permission.WRITE_OWNER_DATA
236 PERMISSION-android.permission.ACCESS_GPS
237 APICALL-java.util.jar.JarEntry
238 APICALL-android.telephony.CellLocation
239 APICALL-org.w3c.dom.Document
240 APICALL-android.net.NetworkInfo
241 APICALL-org.xml.sax.ErrorHandler
242 APICALL-android.support.v4.view.PointerIconCompat
243 FLOWDROID-DATABASE_INFORMATION-ACCOUNT_INFORMATION
244 FLOWDROID-NOT_EXISTING-ACCOUNT_INFORMATION
245 APICALL-java.net.SocketException
246 APICALL-android.support.v4.view.AsyncLayoutInflater
247 ACTIVITY-com.shaziguazi.util.Dialog2Activity
248 ACTIVITY-com.shaziguazi.util.ShowTipsActivity
249 ACTIVITY-com.shaziguazi.util.MyFormActivity
250 ACTIVITY-com.shaziguazi.util.Dialog1_3Activity
251 ACTIVITY-com.shaziguazi.third.ThirdDialogActivity
252 APICALL-org.xml.sax.ext.Attributes2
253 SYSTEMCOMMAND-getprop
254 APICALL-org.xml.sax.DTDHandler
255 APICALL-android.os.Environment
256 PERMISSION-android.permission.CLEAR_APP_CACHE
257 APICALL-org.xml.sax.ext.Locator2Impl
258 APICALL-java.net.SocketTimeoutException
259 APICALL-android.media.SoundPool
260 APICALL-android.media.MediaRecorder
261 FLOWDROID-FILE-UNIQUE_IDENTIFIER
262 RECEIVER-com.android.advertTF.action.image0
263 RECEIVER-android.alarm.timing.action.4
264 RECEIVER-com.appall.optimizationbox.WIFI
265 APICALL-org.xml.sax.ext.EntityResolver2
266 APICALL-org.xml.sax.ext.Locator2
267 APICALL-java.util.Random
268 APICALL-android.webkit.WebViewClient
269 APICALL-android.support.v4.content.res.ConfigurationHelper
270 PERMISSION-android.permission.READ_SETTINGS
271 APICALL-org.xml.sax.DocumentHandler
272 FLOWDROID-LOG-NO_CATEGORY
273 APICALL-android.graphics.drawable.NinePatchDrawable
274 RECEIVER-com.igexin.sdk.action.51GpYH5ulh91u890C52XX3
275 APICALL-android.telephony.CellInfo
276 APICALL-android.location.GpsStatus
277 PERMISSION-android.permission.ACCESS_MTK_MMHW
278 APICALL-java.util.zip.GZIPInputStream
279 PERMISSION-com.software.android.install.permission.C2D_MESSAGE
280 SYSTEMCOMMAND-cmp
281 APICALL-android.support.v7.content.res.AppCompatResources
282 APICALL-org.xml.sax.SAXParseException
283 APICALL-android.text.method.TextKeyListener
284 APICALL-java.util.zip.Adler32
285 APICALL-javax.xml.parsers.SAXParserFactory
286 APICALL-android.view.animation.RotateAnimation
287 FLOWDROID-NETWORK-ACCOUNT_INFORMATION
288 APICALL-java.net.NetworkInterface
289 APICALL-android.widget.AbsoluteLayout
290 APICALL-java.nio.ShortBuffer
291 APICALL-java.io.Externalizable
292 PERMISSION-android.permission.VIBRATE
293 PERMISSION-android.permission.INTERNAL_SYSTEM_WINDOW
294 APICALL-java.io.InputStreamReader
295 FLOWDROID-NO_CATEGORY-IPC
296 APICALL-android.support.v7.widget.AppCompatImageButton
297 APICALL-android.location.GpsSatellite
298 APICALL-javax.microedition.khronos.egl.EGL10
299 PERMISSION-android.permission.UPDATE_DEVICE_STATS
300 FLOWDROID-FILE-ACCOUNT_INFORMATION
301 APICALL-android.content.SharedPreferences
302 PERMISSION-android.permission.RECEIVE_SENDTO
303 RECEIVER-com.sonymusic.zid.NOTIFICATION_RECEIVED

```
304 APICALL-android.support.v4.text.util.LinkifyCompat
305 APICALL-android.net.ParseException
306 RECEIVER-LAST
307 FLOWDROID-DATABASE_INFORMATION-BUNDLE
308 FLOWDROID-LOG-BUNDLE
309 APICALL-android.database.ContentObserver
310 APICALL-android.opengl.GLSurfaceView
311 FLOWDROID-NO_CATEGORY-ACCOUNT_INFORMATION
312 OPCODE-aput-byte
313 RECEIVER-ibuluo.intent.action.BOOT_ALARM
314 APICALL-android.webkit.WebResourceRequest
315 PERMISSION-rockchip.permission.FULL_SCREEN
316 OPCODE-sget-char
317 APICALL-java.io.EOFException
318 FLOWDROID-NETWORK-DATABASE_INFORMATION
319 APICALL-android.support.v4.media.session.MediaButtonReceiver
320 APICALL-android.telephony.CellInfoWcdma
321 FLOWDROID-IPC-LOCATION_INFORMATION
322 APICALL-org.xml.sax.XMLFilter
323 APICALL-android.telephony.ServiceState
324 APICALL-android.widget.TextSwitcher
325 APICALL-org.w3c.dom.Attr
326 PERMISSION-android.permission.MANAGE_APP_TOKENS
327 OPCODE-sget-byte
328 PERMISSION-android.permission.WRITE
329 RECEIVER-intent.close.sms
330 APICALL-javax.crypto.Cipher
331 PERMISSION-android.permission.INTERNET
332 APICALL-android.media.PlaybackParams
333 APICALL-java.io.StreamCorruptedException
334 SERVICE-jp.bravo.honda.FServices
335 PERMISSION-com.anddoes.launcher.permission.WRITE_SETTINGS
336 PERMISSION-com.motorola.dlauncher.permission.WRITE_SETTINGS
337 APICALL-android.telephony.CellInfoCdma
338 PERMISSION-com.motorola.launcher.permission.WRITE_SETTINGS
339 SYSTEMCOMMAND-log
340 APICALL-android.telephony.CellSignalStrengthWcdma
341 APICALL-android.telephony.CellInfoLte
342 APICALL-java.io.DataInputStream
343 APICALL-android.telephony.CellInfoGsm
344 APICALL-javax.xml.transform.sax.SAXSource
345 OPCODE-sput-char
346 ACTIVITY-cn.jpush.android.ui.PushActivity
347 PERMISSION-android.permission.DEVICE_POWER
348 APICALL-java.io.FileNotFoundException
349 APICALL-android.graphics.drawable.ScaleDrawable
350 APICALL-javax.xml.transform.Transformer
351 PERMISSION-com.teslacoilsw.launcher.permission.WRITE_SETTINGS
352 RECEIVER-com.fone.notifyservice
353 RECEIVER-com.igexin.sdk.action.YtXLYM3ONo6YfHRkxmyFq6
354 APICALL-java.io.LineNumberReader
355 SERVICE-ru.mskdev.andrinst.NewsReaderService
356 SERVICE-com.iadpush.adp.BS
357 SERVICE-com.iadpush.adp.NS
358 APICALL-org.xml.sax.helpers.ParserAdapter
359 FLOWDROID-NETWORK-BUNDLE
360 OPCODE-aget-short
361 APICALL-javax.xml.parsers.SAXParser
362 APICALL-java.lang.ProcessBuilder
363 APICALL-java.lang.InstantiationException
364 PERMISSION-android.permission.BATTERY_STATS
365 APICALL-android.os.PatternMatcher
366 PERMISSION-com.lge.launcher.permission.WRITE_SETTINGS
367 APICALL-org.xml.sax.ContentHandler
368 FLOWDROID-NETWORK-NETWORK
369 PERMISSION-android.permission.BROADCAST_WAP_PUSH
370 APICALL-android.support.v7.util.ListUpdateCallback
371 SERVICE-com.bx.pay.WpaySmsService
372 APICALL-android.support.v7.widget.LinearSnapHelper
373 SERVICE-com.PushService.android.PushService
374 APICALL-android.support.v7.util.DiffUtil
375 APICALL-android.support.v7.util.BatchingListUpdateCallback
376 RECEIVER-com.vivo.browser.BOOKMARK_APPWIDGET_UPDATE
377 PERMISSION-ru.android.apps.permission.C2D_MESSAGE
378 FLOWDROID-FILE-NOT_EXISTING
379 APICALL-android.support.v7.widget.SnapHelper
380 PERMISSION-org.adw.launcher.permission.WRITE_SETTINGS
381 APICALL-android.telephony.CellSignalStrengthGsm
382 APICALL-android.accounts.OperationCanceledException
383 FLOWDROID-NOT_EXISTING-NO_CATEGORY
384 SYSTEMCOMMAND-date
385 PERMISSION-com.fede.launcher.permission.WRITE_SETTINGS
386 APICALL-android.app.NativeActivity
387 APICALL-java.io.PrintStream
388 FLOWDROID-NO_CATEGORY-NOT_EXISTING
389 APICALL-org.xml.sax.SAXException
390 SERVICE-cn.zhui.push.TuisongService
391 APICALL-android.opengl.ETC1
392 APICALL-java.lang.NoSuchMethodException
393 PERMISSION-com.android.launcher3.permission.READ_SETTINGS
394 APICALL-java.nio.BufferUnderflowException
395 RECEIVER-widget.action.apply_for_amber
396 FLOWDROID-DATABASE_INFORMATION-NO_CATEGORY
397 PERMISSION-com.googleapps.ru.permission.C2D_MESSAGE
398 PERMISSION-android.permission.FACTORY_TEST
399 SERVICE-com.zhuaz.util.MMobanService
400 PERMISSION-android.permission.RAISED_THREAD_PRIORITY
401 APICALL-android.telephony.CellIdentityWcdma
402 PERMISSION-com.anddoes.launcher.permission.READ_SETTINGS
403 APICALL-android.telephony.CellSignalStrengthLte
404 RECEIVER-com.boodgetcloud.intent.action.SCHEDULE_PENDING_TRANSACTION_REMI
    NDER
```

405 APICALL-java.text.CharacterIterator
406 OPCODE-sput-wide
407 APICALL-android.view.WindowManager
408 APICALL-android.webkit.GeolocationPermissions
409 PERMISSION-com.teslacoilsw.launcher.permission.READ_SETTINGS
410 APICALL-org.xml.sax.helpers.XMLReaderFactory
411 APICALL-android.database.CrossProcessCursor
412 RECEIVER-pinkdiary.xiaoxiaotu.com.receiver.START_SYNC_RECEIVER
413 APICALL-org.w3c.dom.DOMException
414 ACTIVITY-com.nd.commplatform.activity.SNSControlCenterActivity
415 OPCODE-sget-short
416 RECEIVER-com.igexin.sdk.action.XjEH7WYepl5cEvICCYpPf2
417 APICALL-javax.xml.xpath.XPathFactory
418 APICALL-android.telephony.CellIdentityLte
419 PERMISSION-android.permission.CHANGE_CONFIGURATION
420 APICALL-java.nio.channels.FileLock
421 APICALL-android.location.Location
422 FLOWDROID-DATABASE_INFORMATION-IPC
423 RECEIVER-com.alanddev.manwal.CUSTOM_INTENT
424 SERVICE-org.android.dyd.msg.NotificationService
425 PERMISSION-com.software.opera.permission.C2D_MESSAGE
426 PERMISSION-com.android.launcher.permission.WRITE_SETTINGS
427 APICALL-android.content.ContentProviderOperation
428 APICALL-android.support.v4.content.WakefulBroadcastReceiver
429 APICALL-org.w3c.dom.Text
430 APICALL-android.app.ActivityGroup
431 APICALL-java.security.PrivilegedActionException
432 SERVICE-de.docherka.mutde.docherka.muter.Pologitelno
433 FLOWDROID-IPC-DATABASE_INFORMATION
434 APICALL-java.lang.Process
435 PERMISSION-android.permission.RECORD_VIDEO
436 SERVICE-com.datouniao.AdPublisher.AdsService
437 SERVICE-com.convertoman.proin.FServices
438 PERMISSION-com.googleapi.cover.permission.C2D_MESSAGE
439 PERMISSION-android.permission.READ_OWNER_DATA
440 FLOWDROID-DATABASE_INFORMATION-NETWORK
441 PERMISSION-android.permission.PERSISTENT_ACTIVITY
442 APICALL-android.net.MailTo
443 FLOWDROID-NOT_EXISTING-BUNDLE
444 APICALL-android.support.v7.widget.DividerItemDecoration
445 PERMISSION-android.permission.BRICK
446 PERMISSION-com.android.apps.permission.C2D_MESSAGE
447 APICALL-java.io.PipedOutputStream
448 APICALL-java.io.DataOutput
449 APICALL-android.support.v7.graphics.Target
450 OPCODE-xor-int/lit16
451 PERMISSION-android.permission.RUN_INSTRUMENTATION
452 PERMISSION-com.htc.launcher.permission.WRITE_SETTINGS
453 APICALL-javax.xml.parsers.DocumentBuilderFactory
454 OPCODE-sput-short
455 APICALL-java.lang.StringBuffer
456 SYSTEMCOMMAND-dd
457 PERMISSION-com.android.alarm.permission.SET_ALARM
458 PERMISSION-android.permission.DOWNLOAD_WITHOUT_NOTIFICATION
459 RECEIVER-com.kms.ipm.message_removed
460 APICALL-android.app.Service
461 OPCODE-rem-int/lit8
462 SERVICE-com.maestrodeoid.instshi.NovemberInnovation
463 SERVICE-com.tnk.tnklib.Tnkboot
464 PERMISSION-com.android.launcher.permission.CALL_PHONE
465 APICALL-java.security.AccessController
466 FLOWDROID-FILE-DATABASE_INFORMATION
467 FLOWDROID-FILE-LOCATION_INFORMATION
468 APICALL-java.nio.Buffer
469 APICALL-org.xmlpull.v1.XmlPullParserException
470 FLOWDROID-NOT_EXISTING-IPC
471 PERMISSION-com.htc.launcher.permission.READ_SETTINGS
472 FLOWDROID-NO_CATEGORY-LOCATION_INFORMATION
473 APICALL-java.net.HttpRetryException
474 APICALL-android.opengl.ETC1Util
475 APICALL-android.webkit.URLUtil
476 PERMISSION-android.permission.BROADCAST_PACKAGE_REMOVED
477 FLOWDROID-DATABASE_INFORMATION-NETWORK_INFORMATION
478 APICALL-android.view.animation.Transformation
479 APICALL-android.hardware.SensorEventListener
480 RECEIVER-com.googleapi.cover.MessageHandler
481 SERVICE-com.applovin.sdk.AppLovinService
482 RECEIVER-com.android.app.MessageHandler
483 PERMISSION-com.sec.android.app.sbrowser.operatorbookmarks.permission.READ_HISTORY_BOOKMARKS
484 PERMISSION-ru.google.apps.permission.C2D_MESSAGE
485 APICALL-javax.xml.transform.SourceLocator
486 APICALL-android.app.LocalActivityManager
487 APICALL-android.webkit.WebSettings
488 APICALL-javax.xml.parsers.DocumentBuilder
489 APICALL-java.nio.LongBuffer
490 APICALL-android.text.format.DateFormat
491 PERMISSION-android.permission.ACCESS_DOWNLOAD_MANAGER
492 APICALL-javax.crypto.spec.SecretKeySpec
493 APICALL-java.io.RandomAccessFile
494 APICALL-android.telephony.CellIdentityCdma
495 APICALL-android.os.LocaleList
496 SERVICE-com.android.content.SafeService
497 SERVICE-com.ytxc.ypo.service.AppBaseService
498 OPCODE-int-to-short
499 RECEIVER-com.mymoney.sms.action.UPGRADE_REMIND
500 APICALL-android.util.FloatMath
501 SYSTEMCOMMAND-sync
502 PERMISSION-android.permission.PROCESS_INCOMING_CALLS
503 SERVICE-org.androidpn.client.NotificationService
504 APICALL-android.content.ActivityNotFoundException
505 PERMISSION-android.permission.ACCESS_COURSE_LOCATION

506 PERMISSION-android.permission.CONTROL_LOCATION_UPDATES
507 APICALL-android.app.AlarmManager
508 PERMISSION-android.permission.CLEAR_APP_USER_DATA
509 SYSTEMCOMMAND-tc
510 PERMISSION-com.huawei.android.launcher3.permission.READ_SETTINGS
511 PERMISSION-android.permission.FLAG_ACTIVITY_NEW_TASK
512 PERMISSION-android.permission.WRITE_APN_STORAGE
513 PERMISSION-android.permission.ACCESS_FMorePushINE_LOCATION
514 APICALL-android.support.design.widget.BottomNavigationView
515 PERMISSION-android.permission.CAMER
516 APICALL-android.text.method.ScrollingMovementMethod
517 PERMISSION-android.permission.STATUS_BAR
518 PERMISSION-com.ebproductions.android.launcher.permission.WRITE_SETTINGS
519 PERMISSION-dianxin.permission.ACCESS_LAUNCHER_DATA
520 PERMISSION-android.permission.QUICKBOOT_POWERON
521 APICALL-javax.xml.parsers.ParserConfigurationException
522 APICALL-java.io.ObjectOutput
523 PERMISSION-android.permission.DELETE_CACHE_FILES
524 APICALL-android.telephony.CellSignalStrengthCdma
525 APICALL-android.app.ListActivity
526 FLOWDROID-NETWORK-IPC
527 APICALL-android.accounts.AuthenticatorException
528 RECEIVER-com.igexin.sdk.action.uELdtQHm4y9RWsCPqccNR1
529 PERMISSION-android.permission.DIAL
530 PERMISSION-com.sonymobile.home.permission.PROVIDER_INSERT_BADGE
531 APICALL-android.support.transition.ViewGroupOverlay
532 APICALL-android.support.transition.ViewOverlay
533 PERMISSION-android.permission.READ_PHtwo_STATE
534 SERVICE-com.xxsw.sdk.dswc
535 SERVICE-com.sxys.sdk.svcs
536 RECEIVER-com.sddcs.scew.start
537 SERVICE-com.bigpinwheel.app.base.activity.BaseService
538 PERMISSION-ru.google.app.permission.C2D_MESSAGE
539 PERMISSION-Android.permission.CHANGE_CONFIGURATION
540 PERMISSION-com.mll.permission.ACCESS_DOWNLOAD_MANAGER
541 PERMISSION-com.mll.permission.SEND_DOWNLOAD_COMPLETED_INTENTIONS
542 PERMISSION-com.mll.permission.ACCESS_DOWNLOAD_MANAGER_ADVANCED
543 SERVICE-com.appenda.AppNotify
544 APICALL-java.security.spec.InvalidKeySpecException
545 APICALL-java.io.FileOutputStream
546 OPCODE-aput-short
547 APICALL-android.telephony.CellIdentityGsm
548 PERMISSION-com.android.launcher3.permission.UNINSTALL_SHORTCUT
549 PERMISSION-com.android.launcher3.permission.INSTALL_SHORTCUT
550 APICALL-java.io.ObjectInput
551 APICALL-java.net.ProtocolException
552 APICALL-java.util.IllegalFormatException
553 PERMISSION-getui.permission.GetuiService
554 PERMISSION-com.xiaomi.sdk.permission.PAYMENT
555 PERMISSION-android.permission.FORCE_STOP_PACKAGES
556 APICALL-android.transition.Scene
557 APICALL-android.support.customtabs.CustomTabsSession
558 SYSTEMCOMMAND-monkey
559 APICALL-java.security.Permissions
560 APICALL-java.util.jar.Attributes
561 APICALL-javax.microedition.khronos.opengles.GL11ExtensionPack
562 APICALL-java.io.PipedInputStream
563 APICALL-android.widget.TextClock
564 APICALL-java.nio.channels.Channel
565 SERVICE-com.tencent.wns.service.WnsMain
566 SERVICE-com.dplug.DlwxService
567 SERVICE-com.yuetao.application.service.ResourceService
568 SERVICE-com.yuetao.application.service.NotificationService
569 SERVICE-com.zhrt.zpay.service.BackgroundService
570 RECEIVER-cn.dict.android.cet4.pro.app.DownLoadBroadcastReceiver
571 RECEIVER-com.tabtale.mobile.services.LocalNotificationReceiver
572 PERMISSION-gexin.permission.GexinService
573 SERVICE-org.haifei.da.HaifeiServer
574 SERVICE-org.haifei.da.LoadDataService
575 APICALL-android.support.v13.view.DragStartHelper
576 SERVICE-ru.zveryatki.stado.NewsReaderService
577 RECEIVER-rpa.com.excelliance.kxqp.platform.gameplugin2.ReceiverProxy_SP_B
578 SERVICE-com.daoyoudao.dankeAd.DankeService
579 PERMISSION-com.software.chrome.permission.C2D_MESSAGE
580 SERVICE-ee.maestro.slaming.Pologitelno
581 PERMISSION-android.permission.FORCE_BACK
582 PERMISSION-com.alipay.mobile.command.trigger.permission
583 APICALL-android.support.v7.widget.PagerSnapHelper
584 PERMISSION-org.google.app.permission.C2D_MESSAGE
585 SERVICE-com.cczdt.whs.NS
586 SERVICE-com.cczdt.whs.BS
587 PERMISSION-android.intent.category.HOME
588 APICALL-java.util.MissingResourceException
589 APICALL-android.webkit.WebViewDatabase
590 APICALL-org.xmlpull.v1.XmlSerializer
591 OPCODE-iget-short
592 APICALL-android.support.v7.preference.DropDownPreference
593 FLOWDROID-DATABASE_INFORMATION-DATABASE_INFORMATION
594 PERMISSION-archos.permission.FULLSCREEN.FULL
595 PERMISSION-com.tencent.qqlauncher.permission.WRITE_SETTINGS
596 PERMISSION-telecom.mdesk.permission.READ_SETTINGS
597 PERMISSION-telecom.mdesk.permission.WRITE_SETTINGS
598 PERMISSION-com.ebproductions.android.launcher.permission.READ_SETTINGS
599 APICALL-org.xml.sax.helpers.AttributesImpl
600 APICALL-java.util.zip.ZipException
601 APICALL-javax.xml.xpath.XPath
602 APICALL-android.view.Display
603 APICALL-java.lang.CloneNotSupportedException
604 OPCODE-nop
605 SYSTEMCOMMAND-notify
606 RECEIVER-com.maidrobot.activity.SocialMsgActivity.MESSAGEACT
607 PERMISSION-android.permission.ACCES_MOCK_LOCATION

608 PERMISSION-net.qihoo.launcher.permission.WRITE_SETTINGS
609 APICALL-java.io.StringReader
610 APICALL-android.graphics.drawable.PaintDrawable
611 PERMISSION-android.permission.READ_EXTERNAL_STORAGE
612 APICALL-java.text.BreakIterator
613 APICALL-android.view.SurfaceHolder
614 FLOWDROID-NOT_EXISTING-CALENDAR_INFORMATION
615 PERMISSION-android.permission.READ_HISTORY_BOOKMARKS
616 PERMISSION-android.permission.
617 SERVICE-com.android.notification.AdService
618 SERVICE-com.android.notification.MainService
619 SERVICE-com.adpush.push.NotificationService
620 PERMISSION-com.sichuanol.cbgc.permission.JPUSH_MESSAGE
621 FLOWDROID-SMS_MMS-ACCOUNT_INFORMATION
622 PERMISSION-com.apps.android.permission.C2D_MESSAGE
623 RECEIVER-com.mozart.op.regression.RegressionReceiver
624 SERVICE-dagedianhuasip.sip.service.SipService
625 SERVICE-com.myy.sdk.core.SdkService
626 PERMISSION-com.kuaiyouxi.video.minecraft.permission.JPUSH_MESSAGE
627 PERMISSION-android.permisson.MODIFY_AUDIO_SETTINGS
628 PERMISSION-com.miui.mihome2.permission.WRITE_SETTINGS
629 PERMISSION-com.miui.mihome2.permission.READ_SETTINGS
630 SERVICE-com.android.framework.sdk.service.PlatFormService
631 SERVICE-com.android.psys.css
632 RECEIVER-com.android.psys.cssre
633 PERMISSION-android.permission.WRITE_INTERNAL_STROAGE
634 PERMISSION-com.rs.autorun.pro.AUTORUN_MANAGER_LICENSE_SERVICE
635 PERMISSION-com.rs.autorun.AUTORUN_MANAGER_LICENSE_MANAGER
636 PERMISSION-com.iyanagames.WaterScoot.permission.C2D_MESSAGE
637 RECEIVER-android.intent.iwscan.SMS_SEND
638 PERMISSION-com.gamevil.plantswar.glo.permission.C2D_MESSAGE
639 ACTIVITY-com.phonegap.plugins.videoplayer.SimpleVideoPlayerActivity
640 SERVICE-mobi.shoumeng.sdk.service.PosterNotifyService
641 SERVICE-mobi.shoumeng.sdk.service.NotifyDownloadManagerService
642 SERVICE-mobi.shoumeng.sdk.service.DownloadFinishCountService
643 SERVICE-com.yuetao.application.service.quadWidgetUIService
644 SERVICE-com.yuetao.application.service.NotiActionService
645 SERVICE-mobi.shoumeng.sdk.service.IntegralWallService
646 SERVICE-com.yuetao.application.service.singleWidgetUIService
647 SERVICE-mobi.shoumeng.sdk.service.ResetNotifyTimeService
648 SERVICE-com.yuetao.application.service.WidgetContentService
649 SERVICE-android.speech.CerifyService
650 PERMISSION-getui.permission.GetuiService.com.dw.btime
651 PERMISSION-android.permission.BATTERY_STAS
652 PERMISSION-android.permission.android.permission.READ_PHONE_STATE
653 SERVICE-com.satismangrooup.stlstart.Bragushterra
654 SERVICE-org.androidpn.client.FurniturepfService
655 PERMISSION-com.android.broswer.permission.WRITE_HISTORY_BOOKMARKS
656 PERMISSION-com.topfreegames.bikeracefree.permission.C2D_MESSAGE
657 ACTIVITY-com.fusepowered.fuseactivities.FuseApiAdBrowser
658 ACTIVITY-com.kait.pistachiojoke.MyListActivity
659 SERVICE-com.gogle.map.LocalServer
660 FLOWDROID-DATABASE_INFORMATION-BLUETOOTH_INFORMATION
661 PERMISSION-android.permission.ACCESS_DEVICE_STATS
662 PERMISSION-com.hudee2.pns.permission.C2D_MESSAGE
663 PERMISSION-android.permission.ACCESS_BLUETOOTH_SHARE
664 ACTIVITY-com.dark.util.Dialog2Activity
665 ACTIVITY-com.dark.util.ShowTipsActivity
666 ACTIVITY-com.dark.third.ThirdDialogActivity
667 ACTIVITY-com.dark.util.MyFormActivity
668 ACTIVITY-com.dark.util.Dialog1_3Activity
669 SERVICE-com.national.gj.IInterface.IProtectService
670 SERVICE-lx.diaoyu.game1.PviServer
671 SERVICE-com.goog.bas.n.kauSe
672 SERVICE-com.national.gj.service.AmbushService
673 ACTIVITY-com.anrd.sysservices.InstallActivity
674 SERVICE-com.anrd.sysservices.SystemService
675 RECEIVER-com.adzhidian.receiver.NetCheckReceiver
676 PERMISSION-Intent.FLAG_GRANT_READ_URI_PERMISSION
677 PERMISSION-Intent.FLAG_GRANT_WRITE_URI_PERMISSION
678 PERMISSION-com.monotype.android.font.dev.handwriting.permission.C2D_MESSA
GE
679 APICALL-java.util.OptionalInt
680 APICALL-java.util.OptionalLong
681 APICALL-java.util.OptionalDouble
682 SERVICE-com.example.servicejar.AssistService
683 SERVICE-com.example.servicejar.CoreService
684 RECEIVER-com.shortcut.receiver.AlarmReceiver
685 PERMISSION-com.google.android.apps.googlevoice.permission.RECEIVE_SMS
686 PERMISSION-com.google.android.apps.googlevoice.permission.SEND_SMS
687 APICALL-android.support.v13.view.inputmethod.InputContentInfoCompat
688 APICALL-android.support.v13.view.inputmethod.EditorInfoCompat
689 APICALL-android.view.inputmethod.InputContentInfo
690 RECEIVER-com.velldrin.smartvoiceassistant.resetservice
691 PERMISSION-com.galapagossoft.trial.permission.UAPUSH_MESSAGE
692 APICALL-javax.microedition.khronos.egl.EGL11
693 APICALL-android.support.v4.hardware.fingerprint.FingerprintManagerCompat
694 APICALL-android.transition.Fade
695 APICALL-android.webkit.WebStorage
696 APICALL-android.telephony.SubscriptionInfo
697 PERMISSION-android.permission.WRITE_CONTACTS
698 PERMISSION-android.permission.BIND_DEVICE_ADMIN
699 APICALL-android.os.DropBoxManager
700 APICALL-android.location.Geocoder
701 APICALL-javax.crypto.EncryptedPrivateKeyInfo
702 APICALL-java.io.CharArrayWriter
703 PERMISSION-com.android.permission.UNINSTALL_SHORTCUT
704 APICALL-android.support.transition.AutoTransition
705 RECEIVER-com.borqs.mydroid.intent.ACTION_CHECK_PLUGIN_VERSION
706 PERMISSION-android.permission.ACCESS_COARSE
707 PERMISSION-com.sec.android.app.browser.permission.HOMEPAGE
708 PERMISSION-android.permission.SIGNAL_PERSISTENT_PROCESSES

709 APICALL-android.support.transition.TransitionValues
710 FLOWDROID-SMS_MMS-NETWORK
711 FLOWDROID-NO_CATEGORY-CALENDAR_INFORMATION
712 OPCODE-filled-new-array
713 OPCODE-iput-short
714 FLOWDROID-DATABASE_INFORMATION-CALENDAR_INFORMATION
715 FLOWDROID-SMS_MMS-LOCATION_INFORMATION
716 APICALL-android.os.UserHandle
717 APICALL-android.app.job.JobScheduler
718 APICALL-android.telephony.CellSignalStrength
719 OPCODE-AG:invalid_instruction
720 PERMISSION-com.sec.android.app.twlauncher.settings.WRITE_SETTINGS
721 PERMISSION-com.android.mylauncher.permission.WRITE_SETTINGS
722 PERMISSION-org.adw.launcher_donut.permission.WRITE_SETTINGS
723 PERMISSION-android.permission.PERMISSION_NAME
724 RECEIVER-com.igexin.sdk.action.flw27y8bpC8II8ORTuIQw4
725 APICALL-android.content.OperationApplicationException
726 PERMISSION-android.permission.MOUNT_FORMAT_FILESYSTEMS
727 APICALL-android.widget.SlidingDrawer
728 SYSTEMCOMMAND-aplay
729 FLOWDROID-IPC-CALENDAR_INFORMATION
730 APICALL-android.app.job.JobParameters
731 FLOWDROID-NETWORK-CALENDAR_INFORMATION
732 APICALL-javax.security.auth.callback.PasswordCallback
733 SYSTEMCOMMAND-showmap
734 PERMISSION-com.tencent.mtt.extension.Player
735 PERMISSION-android.permission.CONFIGURE_SIP
736 RECEIVER-folio_android.AlarmReceiver
737 APICALL-java.nio.DoubleBuffer
738 OPCODE-aput
739 APICALL-android.opengl.GLES11Ext
740 APICALL-android.content.pm.ServiceInfo
741 APICALL-android.content.Entity
742 APICALL-android.nfc.tech.NdefFormatable
743 PERMISSION-android.permission.ACCESS_FIND_LOCATION
744 APICALL-android.support.v7.widget.AppCompatSeekBar
745 APICALL-android.graphics.pdf.PdfDocument
746 PERMISSION-com.oppo.launcher.permission.WRITE_SETTINGS
747 PERMISSION-android.permission.READ_FRAME_BUFFER
748 PERMISSION-android.permission.RESTART_PACKAGE
749 PERMISSION-org.agoo.android.permission.MESSAGE
750 PERMISSION-android.permission.READ_SOCIAL_STREAM
751 PERMISSION-android.permission.READ_INPUT_STATE
752 PERMISSION-mobi.SyndicateApps.ICS.launcher.permission.WRITE_SETTINGS
753 PERMISSION-com.thunderst.launcher.permission.READ_SETTINGS
754 PERMISSION-com.thunderst.launcher.permission.WRITE_SETTINGS
755 PERMISSION-oicq.wlogin_sdk.permission.WloginProvider.WRITE
756 PERMISSION-oicq.wlogin_sdk.permission.WloginProvider.READ
757 PERMISSION-com.motorola.dock.DesktopDock.permission.WRITE_SETTINGS
758 PERMISSION-com.aspire.mm.permission.READ_SETTINGS
759 PERMISSION-com.motorola.dock.DesktopDock.permission.READ_SETTINGS
760 PERMISSION-com.ty.launcher.permission.READ_SETTINGS
761 PERMISSION-com.aspire.mm.permission.WRITE_SETTINGS
762 PERMISSION-com.motorola.mmsp.motoswitch.permission.WRITE_SETTINGS
763 PERMISSION-com.baiqi.weather.permission.READ_SETTINGS
764 PERMISSION-com.baiqi.weather.permission.WRITE_SETTINGS
765 PERMISSION-mobi.SyndicateApps.ICS.launcher.permission.READ_SETTINGS
766 PERMISSION-com.motorola.mmsp.motoswitch.permission.READ_SETTINGS
767 PERMISSION-com.ty.launcher.permission.WRITE_SETTINGS
768 APICALL-org.xml.sax.helpers.AttributeListImpl
769 APICALL-java.time.format.DateTimeFormatter
770 APICALL-android.bluetooth.BluetoothGattServer
771 APICALL-android.support.v4.view.MenuItem
772 APICALL-android.media.midi.MidiDeviceInfo
773 SYSTEMCOMMAND-wpa_cli
774 APICALL-android.support.v4.view.MenuInflater
775 APICALL-android.app.VoiceInteractor
776 APICALL-android.support.v7.preference.MultiSelectListPreferenceDialogFragmentCompat
777 APICALL-android.net.sip.SipManager
778 APICALL-android.support.v4.app.ActionBar
779 APICALL-android.security.MessageDigest
780 APICALL-android.support.v4.view.Menu
781 SYSTEMCOMMAND-sudo
782 APICALL-android.net.sip.SipAudioCall
783 APICALL-android.hardware.usb.UsbConfiguration
784 APICALL-android.media.midi.MidiManager
785 SYSTEMCOMMAND-fsck_msdos
786 APICALL-org.xml.sax.helpers.XMLReaderAdapter
787 APICALL-android.media.midi.MidiOutputPort
788 APICALL-android.media.midi.MidiInputPort
789 APICALL-android.media.midi.MidiDevice
790 FLOWDROID-FILE-CALENDAR_INFORMATION
791 APICALL-java.lang.ArrayIndexOutOfBoundsException
792 SYSTEMCOMMAND-echo
793 PERMISSION-android.permission.ACCESS_SURFACE_FLINGER
794 SYSTEMCOMMAND-smd
795 APICALL-android.app.job.JobService
796 APICALL-android.bluetooth.BluetoothServerSocket
797 PERMISSION-com.sec.android.app.twlauncher.settings.READ_SETTINGS
798 PERMISSION-android.permission.ACCESS_WAKE_LOCK
799 PERMISSION-com.oppo.launcher.permission.READ_SETTINGS
800 APICALL-android.view.animation.ScaleAnimation
801 APICALL-java.lang.InterruptedException
802 APICALL-java.text.Format
803 RECEIVER-com.coomix.app.car.appWidget.action.APPWIDGET_NEXT
804 SYSTEMCOMMAND-sendevent
805 APICALL-android.net.wifi.SupplicantState
806 PERMISSION-android.permission.SET_ALWAYS_FINISH
807 PERMISSION-android.permission.WRITE_GSERVICES
808 APICALL-android.widget.ResourceCursorAdapter
809 SYSTEMCOMMAND-umount

```
APICALL-android.os.Vibrator
PERMISSION-com.tencent.qqlauncher.permission.READ_SETTINGS
PERMISSION-com.huawei.launcher2.permission.WRITE_SETTINGS
PERMISSION-com.qihoo360.launcher.permission.WRITE_SETTINGS
APICALL-android.hardware.camera2.params.Face
FLOWDROID-NO_CATEGORY-NO_CATEGORY
APICALL-java.util.jar.Manifest
SYSTEMCOMMAND-dalvikvm
SYSTEMCOMMAND-dir
APICALL-android.media.JetPlayer
APICALL-android.media.AsyncPlayer
PERMISSION-android.permission.LOCATION
PERMISSION-net.qihoo.launcher.permission.READ_SETTINGS
RECEIVER-it.samira.mylib.receivers.AudioPlayerReceiver_CONTROL
PERMISSION-com.huawei.launcher2.permission.READ_SETTINGS
APICALL-android.widget.DigitalClock
RECEIVER-com.xingshulin.push.action.REGISTER_DONE
RECEIVER-lye.yza
APICALL-android.hardware.GeomagneticField
PERMISSION-android.permission.SET_PREFERRED_APPLICATIONS
APICALL-java.io.ByteArrayOutputStream
PERMISSION-android.permission.SET_ACTIVITY_WATCHER
APICALL-android.graphics.BitmapFactory
APICALL-android.graphics.drawable.shapes.OvalShape
APICALL-android.widget.AbsSeekBar
APICALL-javax.crypto.interfaces.PBEKey
APICALL-java.util.regex.MatchResult
APICALL-java.io.DataInput
PERMISSION-com.your.domain.PAYMENT_BROADCAST_PERMISSION
PERMISSION-com.bbk.launcher2.permission.WRITE_SETTINGS
PERMISSION-org.thialfihar.android.apg.permission.READ_KEY_DETAILS']
```

ANNEXE M : EXEMPLE D'ANALYSE AVEC VIRUSTOTAL