Titre: Title:	Détection de programmes malveillants dédiée aux appareils mobiles
Auteur: Author:	Arthur Fournier
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Fournier, A. (2018). Détection de programmes malveillants dédiée aux appareils mobiles [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/3700/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/3700/
Directeurs de recherche: Advisors:	Samuel Pierre
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

DÉTECTION DE PROGRAMMES MALVEILLANTS DÉDIÉE AUX APPAREILS MOBILES

ARTHUR FOURNIER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

DÉTECTION DE PROGRAMMES MALVEILLANTS DÉDIÉE AUX APPAREILS MOBILES

présenté par : FOURNIER Arthur
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

Mme BELLAÏCHE Martine, Ph. D., présidente
M. PIERRE Samuel, Ph. D., membre et directeur de recherche
M. QUINTERO Alejandro, Doctorat, membre

REMERCIEMENTS

Je tiens à remercier Samuel Pierre, mon directeur de recherche, ainsi que le personnel de Flexgroups, sans qui le projet ATISCOM, et a fortiori, mon projet de maîtrise n'auraient pas pu avoir lieu. Je tiens aussi à les remercier pour le support et l'aide apportée tout au long de la maîtrise. Je remercie aussi le CRSNG pour sa subvention du projet dans son ensemble, ainsi que la fondation Arbour pour m'avoir choisi comme bénéficiaire d'une bourse de maîtrise. Je remercie également les membres du Laboratoire de Réseautique et Informatique Mobile à Polytechnique Montréal pour leurs conseils avisés, et mes amis de Polytechnique pour leur support, technique ou moral. Ensuite, je tiens à remercier l'école CentraleSupélec en France qui m'a permis d'entrer à Polytechnique Montréal en maîtrise de recherche dans le cadre de mon double diplôme. Finalement, je remercie mes parents et mes soeurs qui, malgré la distance, m'ont constamment encouragé pendant ces deux ans de recherche.

RÉSUMÉ

La conception d'une méthode efficace de détection de programmes malveillants dédiée aux appareils mobiles se place dans le contexte d'une architecture multiservices centrée sur le paiement mobile appelée ATISCOM. Cette architecture est développée par le Laboratoire de recherche en réseautique et informatique mobile de Polytechnique Montréal en collaboration avec Flexgroups et subventionnée par le CRSNG. Plusieurs enjeux dans ce projet sont dédiés à la sécurité de la plateforme qui est très sensible puisque elle doit manipuler des informations privées et financières et fonctionner en réseau et sur des appareils mobiles. La menace la plus importante pour les téléphones intelligents est celle du malware, ou logiciel malveillant, et ce mémoire propose d'y répondre.

Nous avons établi une revue de littérature du domaine de la détection de malware sur Android, la plateforme choisie pour ce projet. Elle montre la présence importante des logiciels malveillants dans les environnements mobiles, la menace qu'ils représentent et leur évolution. Celle-ci décrit ensuite les domaines principaux de l'analyse statique et dynamique, sur serveur et sur appareil mobile. Elle montre de plus la présence grandissante de l'apprentissage automatique, et le meilleur équilibre entre précision et performance des systèmes hybrides. Après analyse des méthodes basées sur l'analyse dynamique (et statique) sur appareil mobile les plus prometteuses, nous distinguons leurs lacunes et décidons de bâtir une architecture client-serveur hybride utilisant l'apprentissage automatique pour pallier à ces dernières. La tâche se révèlera trop importante pour une simple maîtrise et nous concentrerons nos efforts sur une méthode d'analyse statique légère pouvant offrir une précision suffisante et rouler sur mobile. Ceci constituera la première pierre pour construire la méthode hybride de l'architecture idéale.

Pour réaliser un prototype de méthode de détection satisfaisant, nous avons réuni de nombreuses applications, soit malveillantes à partir de bases de données en ligne comme Drebin, ou bénignes à partir des top de Google Play. Nous avons utilisé les outils Android et Linux pour extraire les permissions des applications, qui serviront de caractéristiques pour notre modèle d'apprentissage machine. Nous créons ensuite des ensembles d'entraînement et de tests équilibrés (moitié malware et moitié applications bénignes) représentant respectivement 10608 et 502 applications. L'outil utilisé pour entraîner des modèles sera Weka, un programme écrit en Java et dont l'API pourra être incluse dans une application Android ou un programme Java pour le serveur. Nous décrivons la méthodologie et le code utilisé pour

construire les ensembles utilisables par Weka à partir des paquets d'applications Android.

Nous présentons ensuite nos efforts d'optimisation du modèle d'apprentissage automatique. Nous commençons avec un modèle utilisant l'algorithme naïf de Bayes avec un ensemble d'entraînement déséquilibré qui présente une très bonne performance mais une précision de 92,45% (en validation croisée) bien en dessous des meilleurs modèles de la littérature, comme MADAM ou SAMADroid. Nous l'améliorons en montrant que les ensembles équilibrés donnent une meilleur détection des malwares bien que la précision globale puisse paraître en souffrir. Ensuite, nous comparons tous les algorithmes de classification de Weka pour trouver que le classificateur SGD permet une meilleure précision, de 93,85% en validation croisée et 94,42% sur l'ensemble de test. Pour ouvrir le modèle à d'autres possibilités et profiter des algorithmes de régression offerts par Weka, nous changeons nos ensembles en remplaçant la classe `{malware,benign}` par un nombre compris entre -100 et 100. Nous comparons alors tous les nouveaux algorithmes disponibles et obtenons avec l'algorithme Random Forest un coefficent de corrélation égale à 0,9657 en validation croisée et 0,918 sur l'ensemble de test. Une étude détaillée des résultats montre qu'en prenant un seuil égal à 0 pour distinguer malwares et applications bénignes d'après les prédictions numériques de l'algorithme, notre système détecte tous les malwares de l'ensemble de test mais donne aussi 6 faux positifs.

Nous présentons ensuite le prototype d'application mettant en oeuvre ce dernier modèle, en résumé puis en décrivant le code, notamment pour la classe Prediction.java qui fait le lien entre les activités et l'API de Weka. L'application Android est capable d'effectuer des prédictions automatiques sur les applications nouvellement installées, et d'effectuer des prédictions manuelles sur des fichiers choisis et les applications installées. La précision semble parfaite sur les quelques applications de notre téléphone, bien que l'indice de certitude (valeur absolue de la prédiction numérique) ne soit pas toujours très haut. De plus, les prédictions prennent plus de 10 secondes par application avec cette méthode, ce qui justifie l'idée initiale d'une architecture client-serveur pour décharger les calculs trop lourds de l'appareil mobile. Nous donnons quelques pistes pour poursuivre ce travail en conclusion : l'application mobile est tout à fait portable en programme Java pour le serveur, et un début de communication client-serveur est déjà intégré dans le code fourni. Le travail réalisé est donc un début très prometteur et d'ores et déjà intégrable pour le système de détection de malware qui protègera l'architecture ATISCOM.

ABSTRACT

The design of an efficient malware detection method for mobile device is part of the ATISCOM architecture, which aims to be multiservices, centered on mobile payment. This architecture is developed by LARIM at Polytechnique Montreal, with its industrial partner Flexgroups and with the financial help of CRSNG. There are multiple goals in this project dedicated to improve the security of the platform, which is supposed to handle private and financial information on mobile devices and networks, and thus is very sensitive. The main threat for mobile security is mobile malware, and this work tries to answer it.

We start this paper with a literature review on malware detection for Android, which will be the chosen platform for this project. It first shows the high and increasing number of malware for smartphones in the news. We then describe the sub-domains, such as static and dynamic analysis, server-side and on-device detection. This also shows that machine learning takes a big chunk of the recent papers in the domain, and that the best compromise between precision and performance is often attained by hybrid systems. We review the latest and most interesting papers in the dynamic analysis sub-domain and a few static analysis papers, all for on-device detection. We list their weaknesses (and also the numbers on performance and precision for future comparison) and decide to make our own machine learning client-server hybrid method. But it would be too huge a work for a simple master so we'll focus on a lightweight static analysis on-device detection method for starters.

For a satisfying prototype, we need a whole lot of applications to build training and test set for machine learning models. Most of our malware will come from the Drebin database and all our "benign" applications will come from the top 500's of Google Play. To gather and format these we used Android and Linux tools. We extracted application permissions, which will be the features for the machine learning models in this work. Our final training and test sets will features 10608 and 502 applications, respectively, half of each being malware and the other half being benign. To train our algorithms we use Weka, which has a Java API that we can use on server but also in an Android application if stripped from the UI. We describe the methodology and code used to build the train and test sets andwith Weka from the raw application packages gathered.

We then present our optimization on the machine learning models. We start with Naive Bayes and a semi-representative training set, which gives us the almost good precision of

92,45% in cross-validation. It is still subpar compared to the literature, with systems such as MADAM or SAMADroid. To improve it, we first find that a balanced training and test set gives better malware detection, even if the global precision seems to suffer from this change when looking naively at the numbers. The best classifier we find on Weka is SGD, and it gives us a 93,85% precision in cross-validation for 94,42% on test set. To further improve the model, we try to include regression to our study : we change our binary class `{malware,benign}` to a numeric attribute ranging from -100 to 100 and try the regression algorithms offered by Weka. The best one, Random Forest, gives a correlation coefficient of 0,9657 in cross-validation and 0,918 on the test set. It might not seem like much but the detailed study of the results show that every single malware has been detected for only 6 false positive on the test set, if we use a naive threshold of 0 to decide whether the numeric value represents a malware or not.

To end this dissertation, we present the application prototype which uses the latest built model on Android. The main focus in on the class Prediction.java which links Weka API and our application activities. Our application is able to automatically make predictions on newly installed applications, while also providing the ability to manually scan other installed applications and files on the system for malware. The precision is near perfect on the applications we have on our testing phone, even if the confidence value (absolute of the numeric prediction) is not always high. We also see that the prediction takes more than 10 second per app, thus confirming our initial idea of making a client-server architecture where the Android smartphone can offload heavy computations to the Linux server. To this end, we give some tracks to improve the prototype : the Prediction class can be imported in a Java software for the server, and the network code has already been started in the Android app. The work done here is, in conclusion, already very promising and able to be included in the bigger ATISCOM architecture.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AAPT	Android Asset Packaging Tool
AIDL	Android Interface Definition Language
API	Application programming interface
APK	Android PacKage
ARFF	Attribute-Relation File Format
ART	Android Runtime
ATISCOM	Architecture et technologies innovantes de soutien au commerce mobile
CLI	Command-line Interface
CPU	Central processing unit
CRSNG	Conseil de recherches en sciences naturelles et en génie du Canada
FP	False Positive ; Faux Positif
HTTP	Hypertext Transfer Protocol
IBk	Instance-Based learner based on k-nearest neighbors algorithm
ICC	Inter-Component Communication
IP	Internet Protocol
JAR	Java ARchive
JS	JavaScript
JSON	JavaScript Object Notation
LARIM	Laboratoire de recherche en réseautique et informatique mobile
LMT	Logistic model tree
LSTM	Long Short-Term Memory
LTE	Long-Term Evolution
LWL	Locally weighted learning
MCC	Matthews correlation coefficient
MCCU	Multi Class Classifier Updateable
MD5	Message-Digest algorithm 5
ML	Machine Learning
OS	Operating System
PC	Personal Computer
PDS	Post-Decision State
PRC	Precision Recall Curve
RAM	Random-Access Memory
ROC	Receiver Operating Characteristic

SGD	Stochastic Gradient Descent
SIM	Subscriber identity module
SMO	Sequential Minimal Optimization
SMS	Short Message Service
SVM	Support Vector Machine
TFP	Taux de Faux Positif
TP	True Positive
UI	User Interface
USSD	Unstructured Supplementary Service Data
WEKA	Waikato Environment for Knowledge Analysis
XML	Extensible Markup Language

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Les appareils mobiles sont des systèmes informatiques à part entière, qui font aujourd'hui partie de notre quotidien, plus encore que les systèmes informatiques classiques, comme les ordinateurs personnels. Les informations manipulées par les téléphones intelligents sont nombreuses, et bien souvent privées ou même confidentielles, qu'ils soient utilisés comme outils de travail, comme moyens de paiement ou simplement comme moyens de communication. Et pourtant, ils sont bien souvent plus vulnérables encore que des systèmes informatiques classiques : ils utilisent toutes sortes de réseaux et de protocoles, comme le Wi-Fi, les réseaux mobiles (de troisième et quatrième génération notamment). C'est dans un contexte de commerce mobile, ou m-commerce, que ce projet de maîtrise se place. En effet, le projet ATISCOM du LARIM à Polytechnique Montréal, en collaboration avec Flexgroups et Q-Links, et subventionné par le CRSNG et PROMPT Québec, a pour objectif principal d'établir une plateforme de commerce mobile très sécuritaire et accessible pour se placer dans le marché informel important des pays en développement. Ce projet est découpé en plusieurs enjeux de recherche portant sur la sécurité ou l'accessibilité, et plusieurs enjeux de développement et déploiement. Le mémoire de recherche présentée ici s'inscrit dans le premier objectif du projet ATISCOM, qui vise a concevoir une méthode efficace de détection de programmes malveillants dédiée aux appareils mobiles.

Dans ce chapitre nous donnerons quelques définitions et concepts de base pour la compréhension du mémoire, les éléments de la problématique que nous chercherons à traiter et les objectifs ainsi définis, avant de présenter le plan du mémoire dans son ensemble.

1.1 Définitions et concepts de base

Notre étude repose sur la détection de **maliciels** (*Malware*), ou logiciels malveillants. Il peut s'agir de tout type de logiciel, pour tout support. Sa seule caractéristique est que son développement et son fonctionnement servent des buts illégaux, comme l'espionnage, l'extorsion ou le harcèlement. Dans les cas d'un logiciel dont le fonctionnement s'apparente à celui d'un malware sans forcément avoir été conçu comme tel (collection abusive de données ou envoi trop important de messages pour une application globalement légitime), on parlera parfois de *grayware*. C'est souvent le cas des *adware* qui envoient énormément de publicité. Dans notre étude, lorsqu'on nous parlerons de maliciel, malware ou logiciel malveillant, il s'agira généralement d'applications mobiles pour Android, et par extension du paquet (*package*) ou archive permettant de l'installer sur le support.

En plus des malwares, nous considérerons aussi des **applications bénignes**. Il s'agit de la dénomination que nous utiliserons le long de ce mémoire pour définir une application qui est, selon notre connaissance, légitime et inoffensive. Par définition, ce n'est donc pas un malware. Il est à noter que la certitude de la classification d'une application bénigne ou malveillante est sujette aux méthodes utilisés pour établir cet état de fait. Bien souvent, on considèrera toute application provenant d'une source sûre ou vérifiée par des moyens tiers comme bénigne, mais une incertitude existe presque toujours. Au contraire, une application qui présente des fonctionnalités ou des comportements malveillants et donc classée comme malware par des experts ne laisse généralement aucun doute. Bien que la précédente hypothèse soit presque obligatoire ici, elle représente l'une des plus grandes failles de ce type de projet.

L'**apprentissage automatique** (*Machine Learning*) est une discipline assez large qui est constituée de nombreuses méthodes et objectifs différents. Le dénominateur commun de ces méthodes est l'entrée d'un très grand nombre de données, étiquetées ou non (si oui, on parle d'apprentissage supervisé) dans un algorithme d'apprentissage. Cet algorithme a pour but de donner un sens aux données afin de définir des règles pour traiter de futures données du même type. Cela remplace une analyse exacte impossible à cause de l'explosion combinatoire, qui intervient dès lors que le nombre de variables et de données est trop important pour être traité par un humain ou un algorithme classique. Dans tous les cas, la quantité et l'équilibre des données est très important pour bâtir un bon modèle, et différents algorithmes sont plus adaptés à différents types de données et tâches. Les objectifs sont très variés mais dans notre cas il s'agira de prendre certaines caractéristiques connues de nombreuses applications identifiées comme étant des maliciels ou non, pour déterminer si de nouveaux échantillons sont des maliciels ou non à partir des règles précédemment établies par l'algorithme d'apprentissage.

Les **caractéristiques** (*Features*) sont choisies par l'expert (dans le cas de l'apprentissage supervisé) ou par l'algorithme (dans le cas de l'apprentissage non supervisé) et permettent d'étiqueter et d'identifier chacun des échantillons donnés. On pense par exemple, dans notre cas, à la présence ou non de chacune des 151 permissions du système Android dans le Manifest d'une application spécifique, ce qui donne un vecteur de booléens, et d'une dernière caractéristique définissant si l'application est un malware ou une application bénigne. Cette dernière peut être une classe, c'est-à-dire qu'elle peut uniquement prendre la valeur malware ou bénigne, ou bien un nombre qui définit la probabilité d'être un malware. Dans le premier cas, nous faisons face à un problème de classification, et dans le second à un problème de régression.

Un problème de **classification**, en apprentissage automatique comme ailleurs, est un problème dont l'enjeu est de déterminer la classe d'un échantillon. Les différentes classes sont,

dans notre cas, prédéfinies comme étant *malware* et *benign*. La décision pour un échantillon non étiqueté revient au modèle qui a été entraîné avec les données étiquetées. La majorité des algorithmes de classification testés dans ce mémoire sont assez rapides mais ils ne donnent pas d'information sur la certitude du modèle quant aux choix effectués.

A contrario, un problème de **régression** consiste à associer une certaine valeur (qui ici représente la probabilité d'être un malware ou une application bénigne d'une application) aux autres caractéristiques représentant l'échantillon. On aura par exemple une valeur égale à -100 si l'algorithme est certain que l'application est bénigne, et une valeur égale à 100 s'il est certain que l'application est malveillante. Une valeur de 0 indique que le modèle est totalement incapable de reconnaître la classe de l'échantillon, et toute autre valeur donne une indication sur la certitude de ce dernier lorsqu'il applique les règles qu'il a précédemment construit sur ce nouvel échantillon. On remarquera que plusieurs de ces algorithmes de régression demandent des calculs plus lourds, et requièrent potentiellement un déchargement s'ils sont exécutés sur mobile.

Bien souvent lorsque nous parlerons de la **précision** d'une méthode de détection dans ce mémoire, il s'agira de décrire la capacité du système à correctement prédire si une nouvelle application est malveillante ou bénigne. On considèrera le critère de précision pour évaluer un algorithme de classification qui indique le pourcentage d'applications correctement classées, et le coefficient de corrélation dans le cas d'un algorithme de régression, pour montrer le pourcentage de corrélation entre les valeurs prédites et réelles. On cherchera évidemment à maximiser la précision en priorité.

La précision est à absolument distinguer de la **performance**, que nous utiliserons pour décrire la rapidité d'exécution des algorithmes, en tenant aussi compte si possible de l'espace de stockage et de la mémoire vive utilisé par l'application. Il faut tenir compte du fait que la performance peut se traduire lors de la phase d'entraînement de l'algorithme, mais aussi et surtout lors de la classification/prédiction d'un nouvel échantillon, puisque c'est cette opération qui sera en général effectuée en temps réel sur l'appareil utilisateur.

On parlement aussi de **déchargement** (*Offloading*) à chaque fois qu'il s'agit de décharger des calculs sur le serveur à partir du client. En effet, dans le cadre de notre étude, certains algorithmes peuvent être lourds en calcul et leur exécution directement sur un téléphone Android aux performances limitées risque de prendre plus longtemps que leur offloading. Il s'agit alors d'envoyer certaines données au serveur, qui va faire les lourds traitement en utilisant sa puissance de calcul plus importante, et renvoyer les données au client. L'efficacité de cette opération est donc aussi dépendante de la qualité du réseau reliant client et serveur.

1.2 Éléments de la problématique

Dans le contexte du commerce mobile, les problématiques de sécurité des appareils mobiles introduits précédemment sont d'autant plus importantes. Les appareils, applications, réseaux et serveurs seront amenés à manipuler des informations de vie privée et des devises financières, virtuelles ou non. Les applications de paiement mobile populaire, comme Apple Pay [1], comportent aujourd'hui de nombreuses vulnérabilités qui freinent son adoption par le grand public et mettent en danger ses utilisateurs. Parmi les plus grandes menaces pour le commerce mobile, celle du malware, maliciel ou encore logiciel malveillant est une des plus préoccupante. En 2014 Symantec relève que plus d'un million d'applications sur le marché sont des malwares, et leur nombre ne cesse pas d'augmenter, avec des grands noms comme ZitMo qui font des ravages dans les données des utilisateurs. Les activités des logiciels malveillants sur les appareils mobiles sont l'enregistrement d'appels et de messages textes, la géolocalisation et la transmission des données privées [2]. Certains maliciels, tout aussi répandus que sur les PC par exemple, vont priver les utilisateurs de leurs données, comme les ransomware [3]. Aujourdhui, les méthodes de détection du malware en environnement mobile présentent toujours quelques lacunes et sont généralement insuffisantes, qu'ils s'agissent de forensique mobile, de tests de pénétration, ou encore d'analyse statique ou dynamique du logiciel [4]. Plusieurs méthodes sont couplées avec les capacités matérielles et logicielles des téléphones de nouvelle génération en terme d'exécution isolée ou en confiance [5], qui sont elles aussi nombreuses mais pas forcement optimales [6]. En ce qui concerne les méthodes de détection, le nombre des propositions ne cessent d'augmenter dans la littérature, mais il reste encore de nombreuses lacunes et challenges [7], que nous chercherons a surpasser tout en tenant compte des contraintes liées au marche visé au cours de l'avancement du projet. Une des failles récurrentes est l'utilisation de machines virtuelles [8] à la place d'appareils réels [9], que les pirates vont chercher à détecter afin de ne pas révéler le comportement réel de leurs maliciels. De plus, certaines des méthodes les plus prometteuses reposent sur l'utilisation de l'infonuagique (*cloud*) pour analyser les données [10], et sur le monitoring des communications effectuées par l'appareil [11] [12] [13]. Ceci représente donc une multitude de pistes a explorer pour définir la méthode de détection des logiciels malveillants en environnement mobile qui répond le mieux a notre problématique.

Dans le cadre du projet de plateforme de commerce mobile sécurisée ATISCOM, nous chercherons a déterminer et développer une méthode de détection des logiciels malveillants en environnement mobile. Cette méthode devra corriger les lacunes des méthodes existantes et satisfaire aux contraintes particulières de l'environnement de déploiement qui pourraient être données par les partenaires industriels et les autres équipes de recherche constituantes du

projet. Ces équipes de recherche et développement utiliseront le prototype développé, testé et validé pour atteindre les autres objectifs du projet global et la plateforme finale.

Notre question de recherche sera donc : Peut-on détecter efficacement les programmes malveillants sur les appareils mobiles pour sécuriser le commerce mobile ?

1.3 Objectifs de recherche

L'objectif général de notre travail est de concevoir une méthode efficace de détection de programmes malveillants dédiée aux périphériques mobiles pour pallier les lacunes des méthodes existantes. De manière spécifique, ce mémoire vise à :

- Analyser de manière approfondie les méthodes existantes de détection de programmes malveillants et d'exécution sécuritaire et de confiance dans l'environnement mobile et identifier leurs lacunes.
- Définir un ensemble de requis pouvant servir de base à l'élaboration d'une nouvelle méthode palliant ces lacunes.
- Implémenter cette méthode et optimiser sa précision et ses performances en respectant les contraintes du projet, telles que la simplicité d'utilisation et l'autonomie de la solution.

1.4 Plan du mémoire

Le plan du mémoire va suivre en grande partie les objectifs spécifiques du projet. En effet, comme rappelé plusieurs fois en introduction, nous commencerons par faire une revue de littérature détaillée du domaine. Le laboratoire n'ayant pas à ce jour de revue sur le sujet, nous commencerons par une vue d'ensemble, basée notamment sur l'actualité et sur d'autres articles scientifiques généralistes publiés dans les journaux scientifiques, et de méthodes dites de référence. Ces méthodes de référence présentent des techniques utilisées de manière récurrentes et sont donc souvent citées. Cela nous permettra de déterminer quel sous-domaine de la détection de malware sur Android est le plus propice pour notre architecture. L'analyse plus en détail de ce sous-domaine et des articles les plus prometteurs publiés récemment permettra de mettre en lumière les lacunes restantes, ainsi que des points de comparaison pour notre propre méthode.

Une fois ces lacunes déterminées, nous pourrons les résumer à travers une liste de requis pour la construction de notre architecture. Il s'agira là des points que nous voulons idéalement travailler sur le long terme pour proposer une méthode innovante et supérieure. Nous décrirons

ensuite les étapes nécessaires à cette construction, en détaillant celles qui ont été réalisées au cours de la maîtrise. Tout d'abord nous décrirons l'architecture client-serveur envisagée et les méthodes de détection à utiliser sur chaque support. Il s'agit ensuite de montrer quelles données nous avons rassemblé et quelles caractéristiques des applications nous avons considéré pour bâtir un modèle d'apprentissage automatique servant de coeur à notre méthode de détection de malware pour Android.

La dernière partie présente d'une part les résultats de précision et performance offerts par le prototype réalisé, et d'autre part la réalisation de ce dernier du point de vue technique. Ce pan se caractérise par la recherche de méthode optimale au niveau des algorithmes et des échantillons utilisés pour le modèle d'apprentissage machine. Nous procéderons à une comparaison des ensembles d'échantillons utilisés pour la construction du modèle afin d'en déterminer la composition optimale, puis nous comparerons les nombreux algorithmes disponibles. Nous verrons la précision et la performance des différents algorithmes de classification inclus dans Weka, puis nous présenterons les algorithmes de régression après avoir modifié les ensembles de données. Nous donnerons enfin une description de l'implémentation sous Android de l'application de démonstration mettant en oeuvre ce modèle.

CHAPITRE 2 DETECTION DE MALWARES SUR ANDROID

La premier chapitre de ce mémoire présente une revue de littérature détaillée sur la détection de malwares sur Android. En effet, afin de proposer et développer une méthode efficace pour l'architecture, nous devons en premier lieu étudier les travaux existants de manière générale. Nous pourrons, une fois la vue d'ensemble du domaine apprivoisée, déterminer ensuite un domaine d'étude plus restreint et prometteur pour nos travaux. L'étude et la comparaison des toutes dernières méthodes de ce domaine permettra alors de déterminer leurs forces et lacunes pour trouver la place de notre travail dans le paysage des méthodes de détection pour Android.

2.1 Point de vue général et références

Comme dit plus haut et détaillé plus bas, nous concentrerons nos recherches sur les méthodes pour Android. Cette section présente l'actualité du malware et des contre-mesures, suivi d'articles qui présentent le domaine dans son ensemble et les méthodes phares de la détection de malware pour Android.

2.1.1 Actualité du malware sur mobile et des contre-mesures

L'actualité parle de plusieurs mécanismes de défense dans Android Play Protect [1], dont la plupart étaient déjà intégrés avant la version 8 mais sont maintenant visibles aux utilisateurs. De plus, il faut rappeler que le framework Android est basé sur Linux, et que chaque application roule dans un environnement restreint, grâce à la machine virtuelle Android, anciennement Dalvik et aujourd'hui Android Runtime (ART) [14], ce qui garantie de base un certain degré de sécurité, mais force aussi des méthodes de détection adaptées en cas de lacune des systèmes existants. Les applications peuvent êtres installées depuis le Google Play Store, qui intègre déjà un scanner de malware appelé Bouncer basé sur l'analyse dynamique mais malheureusement imparfait [15]. De plus, il est possible d'installer des applications de sources inconnues. Une pratique commune consiste pour les auteurs de malware à repackager des applications connues (bien souvent payantes) en y intégrant des fonctionnalités malveillantes, avant de les redistribuer gratuitement à des utilisateurs naïfs par des Store non officiels libres, comme Aptoide [2]. Pour pallier aux lacunes de Bouncer, Play Protect, et se protéger des applications de sources inconnues, plusieurs fournisseurs d'antivirus pour

1. `https://www.android.com/play-protect/`
2. `https://fr.aptoide.com/`

smartphone Android sont distribués, en version gratuite ou premium, pour les particuliers ou les entreprises (au sein d'une architecture de gestion de flotte). On peut par exemple citer Lookout [16] pour les particuliers, et MI:RIAM de Wandera [17], z9Engine de Zimperium [18] ou Skycure de Symantec [19] pour les entreprises. Cependant ces logiciels n'indiquent pas toujours la méthode utilisée : il peut s'agir de simples vérifications de signatures (comparées à la base de données de l'éditeur de l'antivirus, potentiellement générée par de nombreuses méthodes disponibles dans la littérature ou développées en interne), ou d'analyse plus poussée des applications installées. En général, ces antivirus ne peuvent fonctionner qu'au niveau applicatif. Des méthodes plus lourdes requièrent une modification du framework Android ou même du matériel du téléphone, comme pour le HTC D4 de CogSystems [20], et sont donc très peu répandues.

De plus, les créateurs de malware sont toujours plus inventifs, et il arrive toujours qu'une nouvelle famille de malware parvienne à infecter de nombreux utilisateurs avant d'être détectée. On peut régulièrement en voir des exemples sur le site SecureList de Kaspersky Lab [3], ou sur d'autres sites de nouvelles concernant la sécurité informatique. Certains, comme Loapi, disposent d'une architecture modulaire [21]. D'autres, comme TrojanDropper, se décomposent en plusieurs applications qui sont téléchargées par les précédentes [22]. Leurs cibles sont aussi diverses, ceux-ci peuvent être les média sociaux comme Tizi [23], les routeurs comme Switcher [24] ou les sites bancaires comme A2f8a [25]. L'environnement mobile ne fait pas exception à la tendance en sécurité informatique comme quoi l'erreur est surtout humaine, et que toutes les défenses du monde ne pourront rien faire si l'utilisateur se retrouve victime de *social engineering*. Cependant, il y a encore la possibilité d'améliorer la détection automatique des malwares palliant du mieux que possible au manque d'éducation ou d'attention des utilisateurs qui installeraient des applications malveillantes sur leurs appareils, puisque les défenses existantes sont encore loin d'être parfaites, et doivent toujours évoluer face aux nouvelles menaces.

2.1.2 Taxononomie

La taxonomie et comparaison qualitative des techniques d'analyses de sécurité des logiciels Android, publiée en juin 2017 par Sadeghi et al. [7], constitue notre point de départ dans l'étude des méthodes de détection de malware sur mobile existantes. En premier lieu, il faut bien préciser que cette étude, comme son nom l'indique, se cantonne à l'étude des méthodes applicables à Android. Bien que certaines s'appliquent aussi à d'autres plateformes, la majorité sont restreintes à Android et son architecture particulière. Cela s'explique par la

3. https://securelist.com/all/?category=912

part de marché majoritaire des téléphones intelligents utilisant ce système [26], comparativement à iOS, Windows Phone, Blackberry et les systèmes anciens ou minoritaires. De plus, les téléphones non intelligents ne sont pas pris en compte, puisque la majorité des malware s'attaquent aux smartphones. Dans le cas de notre propre projet, se réduire à l'étude des méthodes applicables à Android est parfaitement acceptable : la présence majoritaire de ce système est d'autant plus vraie dans les pays où notre partenaire Flex désire vendre ses services, et c'est aussi le système qu'ils utilisent actuellement. Son utilisation est aussi préférée par les chercheurs car c'est un système ouvert et répandu.

Les auteurs de la taxonomie traitent à la fois de la recherche en détection de malware et de celle qui s'attaque à la détection de vulnérabilité. Cette dernière catégorie ne représente que 26% des papiers et nous ne nous y intéresseront pas. Ils dressent de nombreux tableaux dont les catégories sont édifiées par la lecture des articles et de précédentes revues de littératures et taxonomies. Les études sont classées 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. Les méthodes basées sur l'analyse dynamique sont alors 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. Un dernier tableau, et non des moindres puisqu'il touche à la validité des papiers et des méthodes, classifie les articles selon la disponibilité (et donc la reproductibilité) du code source ou de l'outil développé, et sur l'évaluation de la méthode : S'agit-il d'une étude de cas ou d'un test effectué par des utilisateurs ? Si elle est empirique, d'oú viennent les applications testées ?

À partir de la classification effectuée et de la lecture des nombreux papiers, Sadeghi et al. dressent une liste de lacunes dans l'état de l'art qui devraient selon eux être creusées par de futurs articles :

— Il existe peu d'approches hybrides qui couplent l'analyse statique et dynamique, afin de tirer les fruits des deux types d'analyse, les lacunes de l'une pouvant être palliées par les forces de l'autre.

— Il serait bon d'analyser le système en ayant une vue d'ensemble de ce dernier plutôt que de regarder une application seule. De plus en plus de menaces se basent sur l'interaction entre plusieurs applications d'apparences anodines, mais malveillantes

lorsqu'elles coopèrent.

— Le code natif (C, C++) et le code externe chargé dynamiquement par l'application ne sont pas toujours considérés dans les analyses statiques, ce qui permet aux pirates d'y placer la partie malveillante du code. De plus, un travail de recherche est nécessaire pour contrer l'obfuscation de code, toujours plus utilisée par les auteurs de maliciels.
— Les ICC (Inter-Component Communication, c'est-à-dire les mécanismes de communication entre composants) sont un vecteur d'attaque très probant pour les attaquants, et nombre d'analyses se contente d'étudier les Intents, sans regarder du côté des Content Providers et des AIDL, eux aussi susceptibles de contenir des activités malveillantes.
— Les auteurs doivent rechercher toujours plus de précisions dans les méthodes développées, les faux positifs étant un gros problème de réputation et de fiabilité pour les méthodes fournies aux utilisateurs finaux.
— Pour la génération d'entrées lors d'analyses dynamiques, les auteurs se concentrent trop souvent sur le fuzzing (génération aléatoire d'input, qui se rapproche d'une méthode bruteforce) alors qu'il faut se rapprocher d'un comportement plus réaliste pour correctement explorer l'application et être plus malin que les hackers.
— Il est nécessaire pour les chercheurs de partager au maximum leurs benchmarks, leurs tests et le produit de leur recherche (code, outil), à la fois pour l'évaluation par les pairs et pour l'avancée de la recherche dans le domaine en général : les articles les plus cités sont toujours ceux qui fournissent leur code et permettent aux autres chercheurs de s'appuyer sur leur produit, voire d'améliorer directement la méthode développée.

2.1.3 Méthodes de détection principales

Un des apports de la taxonomie détaillée précédemment est aussi une liste d'outils de références provenant des articles les plus cités, qui nous servira à mieux comprendre les différentes directions prises dans la recherche dans le domaine de la détection de malwares sur Android. Il s'agit en effet de méthodes qui sont régulièrement reprises par les articles plus récents

TaintDroid [27] introduit et prototype une méthode basée sur l'analyse dynamique et très reprise par la suite, le taint tracking.

AppsPlayground [28] reprend le taint tracking de TaintDroid et développe une méthode intelligente de génération d'input et de parcours d'application pour l'analyse dynamique. Comme TaintDroid, il requiert une modification du système Android pour traquer les données via le taint tracking, et les tests ne sont donc effectués que sur émulateur, et non un téléphone réel.

Enck_ [29] provient d'une présentation de W. Enck sur la manière de faire l'analyse statique des applications Android. Cela souligne notamment quelques enjeux comme la décompilation du .apk ou du .dex (le bytecode qui s'exécute dans la machine virtuelle Android) en code source, avec son outil : ded, depuis remplacé par Dare [4]. Par la suite, il montre certaines des limitations du travail actuel, comme l'obfuscation non gérée ou l'échantillon restreint, et des pistes pour la suite, ce qui explique son nombre de citation important pour une simple présentation.

ComDroid [30] et Stowaway [31] se concentrent sur la détection de vulnérabilité par analyse statique.

DroidRanger [32] et Kirin [33] traitent de la détection de malware par analyse statique.

IccTA [34] et FlowDroid [35] traitent de la détection de malware et de vulnérabilité par analyse statique.

Certains de ses articles peuvent être assez vieux (2009 pour Kirin contre 2015 pour IccTA) mais ce sont les plus cités de la taxonomie, ce qui montre que leur influence sur les articles plus récents est colossale. Par exemple, TaintDroid a été cité 1563 fois entre sa publication et fin 2015, Stowaway 745 fois et Kirin 625 fois. En effet TaintDroid décrit une nouvelle méthode de détection des malwares par analyse dynamique utilisant le taint tracking. Ceci a été très repris pour l'analyse dynamique par exemple pour AppsPlayground (lui même cité 129 fois entre 2013 et 2015). Un des points forts des méthodes de cette liste est la disponibilité de l'application ou du code source : en dehors de DroidRanger, tous sont disponibles (d'après Sadeghi et al. au moment de l'écriture de la taxonomie), ce qui permet de construire de nouvelles méthodes améliorant directement les existantes.

2.2 Réduction du domaine d'études et enjeux consacrés

Une fois le domaine d'étude présenté dans son ensemble, nous pouvons remarquer que ce dernier est très large et qu'il sera difficile de proposer une nouvelle architecture à partir des informations précédentes. C'est pourquoi nous devons réduire ce domaine pour en étudier les derniers articles et déterminer les enjeux les plus prometteurs pour notre travail.

4. `http://siis.cse.psu.edu/dare/`

2.2.1 Analyse dynamique par apprentissage automatique sur périphérique mobile

L'objectif de notre travail étant d'améliorer les méthodes existantes, on ne peut évidemment pas se contenter de partir d'articles datant de 5 à 9 années pour identifier les requis pour notre propre méthode. Il s'agit donc maintenant d'étudier des articles très récents, innovants ou inspirés des méthodes de références, afin de voir quelles sont aujourd'hui les lacunes existantes dans le domaine, et où notre travail peut avoir un impact probant. Cependant, comme l'a montré l'étude de la taxonomie de Sadeghi et al. [7], le domaine de recherche en méthode de détection de malware sur Android est très large, et les méthodes, approches et implémentations très variées. Nous devons donc à ce stade faire des choix motivés dans la restriction du domaine d'étude, afin de considérer uniquement les articles pertinents parmi la grande quantité d'articles très récents de 2016 à 2018 (et inévitablement peu cités), pour définir des requis précis et permettre d'implémenter une méthode faisable apportant une contribution réaliste dans le cadre du mémoire de maîtrise.

Nous faisons le choix de nous restreindre aux méthodes d'analyse dynamique. En effet, les méthodes d'analyse statique se confrontent à de nombreux problèmes d'obfuscation, de code natif ou téléchargé, de maliciels à applications multiples et d'autres limitations. Chercher à contrer ces dernières tout en restant dans le cadre de l'analyse statique nous semble moins efficace que de creuser du côté de l'analyse dynamique et du monitoring, et les compétences requises ou qui pourraient être apprises nous semblent aussi moins intéressantes. De plus, la recherche en analyse statique paraît très aboutie et utilisée, là où de nombreuses améliorations peuvent encore être faites dans le cadre de l'analyse dynamique. Si de plus celle-ci prends place sur les appareils des utilisateurs finaux, elle est complémentaire aux analyses statiques (ou dynamiques sur émulateur) faites en amont pour générer les bases de données des antivirus ou filtrer les applications sur le Store.

En terme d'analyse dynamique, la question de l'environnement d'analyse se pose. De nombreuses méthodes décrites dans la littérature ont été évaluées à l'aide d'émulateurs. Ceci permet de modifier le framework Android pour faire une analyse sur certaines couches moins accessibles des appareils réels, et, avec une architecture infonuagique, de facilement faire des analyses de grande échelle, par exemple pour peupler la base de donnée d'un antivirus. Cependant, l'émulation apporte avec elle certains enjeux, dont le plus grand est l'évasion de la virtualisation et la détection par les maliciels de l'environnement factice. Ils risquent alors de ne pas révéler leur comportement suspect. Cela constitue une direction pour la recherche en analyse dynamique des malware. Mais la perspective de mener l'analyse avec

des appareils réels nous semble plus alléchante. D'une part, certains papiers montrent qu'il est tout à fait possible d'adapter les méthodes menées avec émulateurs pour les mener sur des appareils réels et éventuellement monter une architecture pour des tests à grande échelle. D'autre part, l'analyse dynamique suppose que l'application soit utilisée pour pouvoir révéler ses comportements malveillants. Les analyses semi-automatiques peuvent difficilement être mises à l'échelle, et les analyses automatiques requiert des algorithmes de parcours d'application. Nous pensons qu'une perspective plus proche du monitoring, en plaçant une solution de détection de maliciel faisant du monitoring sur l'appareil de l'utilisateur final, peut être un bon moyen de détecter les activités suspectes et les bloquer, pour ensuite classifier le maliciel dans une base de données permettant de le détecter avant même son utilisation par la suite.

Enfin, de nombreux efforts de recherche, notamment en analyse statique mais aussi en analyse dynamique, utilisent des techniques d'apprentissage automatique pour distinguer des comportements malveillants de comportements légitimes. Ceci apparaît donc comme un bon outil pour automatiser les processus et remplacer le jugement humain dans la classification des maliciels. Cependant, son utilisation est liée à la capacité de stockage et de calcul des machines impliquées. Les solutions qui peuvent tourner sur les appareils mobiles doivent restreindre leur utilisation de ce type de technique pour ne pas alourdir l'utilisation normale de l'appareil. D'autre part, l'utilisation d'une structure client-serveur basée sur l'infonuagique permet de décharger le mobile d'une partie des calculs et ainsi effectuer un traitement plus poussé avec des machines puissantes à distance pour une détection précise. Ceci requiert le plus souvent une connexion à internet, qui n'est pas garantie. Mais dans la mesure où l'utilisateur doit de toute manière avoir un accès à internet pour télécharger les applications, et parfois même pour les utiliser, ceci peut constituer une seconde couche d'analyse pour maximiser la précision du détecteur lorsque cela est possible. Enfin, bien que l'automatisation totale et l'intelligence artificielle paraissent porter de bons fruits, il peut être bon d'associer la détection d'anomalie par apprentissage automatique à un jugement humain lorsque la marge d'erreur est trop grande, afin d'apprendre de l'utilisateur final et non uniquement du comportement des applications, qui ont pu être modifiées par le pirate pour tromper la machine et biaiser son apprentissage.

Analyse dynamique

Les articles suivants proposent des méthodes d'analyses dynamiques intéressantes, tandis que les points forts et lacunes de ces méthodes seront détaillés dans la section suivante.

DroidDolphin [36] est plus vieux que les articles suivants, mais aussi plus cité, et se pose comme une référence pour ce qui est de la détection dynamique utilisant l'apprentissage

automatique. Les performances sont plus faibles (86.1 à 92.5% de détection) mais les échantillons utilisés sont nombreux (64000 applications dont la moitié de malveillantes) et celui-ci pose quelques bases pour la méthode. Par contre, l'expérimentation a été faite sur émulateur.

De la même manière, STREAM [37], auquel DroidDolphin est comparé, est plus régulièrement cité comme référence pour l'analyse dynamique utilisant l'apprentissage automatique.

SafeGuard [38] et ADroid [39] proposent des méthodes légères de monitoring et analyse dynamique en environnement réel. Ils ne demandent pas d'accès root mais sont incapables de tracer les appels au noyau et donc de détecter les élévations de privilèges.

MADAM [40] propose une architecture multi-niveau pour la détection de malware par analyse dynamique et apprentissage automatique directement sur le téléphone. C'est un des travaux les plus prometteurs à ce jour, mais il requiert à la fois l'installation de l'apk MADAM et celle de SuperUser (accès root) et Xposed (hooks et events).

L'article de Bhatia et al. [41] propose une analyse dynamique basé sur une installation sur machine virtuelle, une exploration par Monkey [5] et une extraction de features par strace, comme d'autres travaux avant lui.

T2Droid [5] roule en partie dans la zone sécurisée de ARM TrustZone, qui est inclut dans plusieurs téléphones de dernière génération comme système d'exécution en confiance afin de lancer le système puis les applications. Cela donne donc une vue d'ensemble du système qui est lui-même sécurisé, contrairement à des antivirus classiques. Cependant cela suppose d'avoir tout contrôle sur le matériel et accès à la zone sécurisé pour déployer le logiciel. Il est donc difficile voire impossible d'inclure un tel système dans un appareil lambda. Cet article présente plutôt des méthodes qui pourraient intéresser l'industrie et les fabricants, plutôt que l'utilisateur final directement.

DroidInjector [42] est une méthode d'analyse dynamique basé sur l'injection d'une librairie (celle de DroidInjector, via *ptrace_attach*) dans les processus des applications potentiellement malveillantes à monitorer. Ceci permet alors de récupérer de nombreuses informations sur le comportement de l'application afin de les envoyer au serveur distant pour analyse. Les informations récupérées de cette manière seraient selon les auteurs plus pertinentes que celles récupérées directement par d'autres méthodes d'analyse dynamique basées sur strace ou ptrace comme DroidTrace [43]. D'après eux, cette méthode fonctionne aussi bien sur appareil physique que sur émulateur. Ils ne fournissent cependant guère de résultats quantitatifs ou d'application utilisable, mais prévoient de déployer leur méthode sur les marchés Android pour la détection de malware.

5. `https://developer.android.com/studio/test/monkey.html`

Apprentissage automatique

Les articles suivant décrivent des approches basées sur l'apprentissage automatique et l'analyse statique. Il s'agit d'extraire des features à partir de l'application pour ensuite y appliquer diverses méthodes d'apprentissage automatique et ainsi détecter avec une grande précision de nouveaux malwares partageant les features des malwares du groupe de test mais non des programmes légitimes. Les features peuvent être extraites du manifest (et donc des permissions), des Intent, du code, etc. Les algorithmes de classifications peuvent être des méthodes d'ensemble, du Bayes naïf, des chaînes de Markov, etc. Puisque les méthodes ci-dessous sont basées sur l'analyse statique des APK, nous ne pouvons directement nous en inspirer, mais il est intéressant de voir en quoi les méthodes d'apprentissage automatique peuvent être adaptées au contexte de l'analyse dynamique, et les performances offertes par les méthode statiques peuvent aussi apporter de nouveaux points de comparaison. En effet, parmi les méthodes récentes d'analyse dynamique on peut aussi trouver l'utilisation de l'apprentissage automatique, mais nous avons remarqué dans notre étude que la plupart des travaux en analyse statique récents rivalisent plutôt au niveau des algorithmes de classifications utilisés qu'à celui des implémentations et des manières de récupérer les features, qui sont généralement plus diverses en analyse dynamique.

Par exemple MaMaDroid [44] utilise des chaînes de Markov pour des modèles comportementaux, tandis que PIndroid [45] prend en compte les permissions et Intent pour l'apprentissage automatique en analyse statique.

IntelliAV (2017-2018) [46] est un système de détection de malware *on-device* (sur l'appareil) et qui utilise l'analyse statique couplée à l'apprentissage automatique. L'application est déjà disponible sur le Play Store de Google, ce qui permet d'éventuels comparaisons de performances. A partir d'un set d'entraînement et de validation de 19722 applications de VirusTotal, dont 9664 malveillantes, les auteurs ont obtenu un taux de vrai positif de 92.5% et un taux de faux positif de 4.2% avec seulement 1000 attributs générés par l'entraînement, ce qui est beaucoup plus petits que les méthodes d'analyse statique basée sur l'apprentissage automatique hors-appareil. Ceci ne semble pas si bon que certaines méthodes de monitoring et analyse dynamique, mais il faut noter que la solution est aboutie, disponible et potentiellement complémentaire à de telles méthodes.

BAdDroIds, une application développée par Aonzo et al. [47] et disponible sur le Google Play Store, propose une méthode d'analyse statique basée sur l'apprentissage automatique, *on-device* et prétendument efficace : les auteurs indiquent que sa précision est autour de 99%. Lors de l'installation d'une nouvelle application, BAdDroIds extrait les permissions déclarées dans le manifeste et les appels à l'API Android d'après le code DEX. Afin de réduire les faux

résultats, lorsque l'algorithme de classification a une confiance inférieure à 70%, l'application laisse l'utilisateur choisir de classifier le programme installé comme malware ou non. Ceci étant dit, cette part de *Warning* est aussi petite que la part de fausses prédictions. L'algorithme choisi est linéaire et détaillé dans l'article, afin de garantir une utilisation minimale de ressource (comparativement aux autres algorithmes plus gourmands ou moins efficaces considérés). Les permissions et invocations d'AAPI (Android API) les plus importants pour le classificateur montrent que les malwares se traduisent au niveau données par un objectif principal de collecte de données et d'envoi de SMS. La méthode étant statique, il faut préciser que le modèle a été construit dans la meilleure expérience à partir de 12000 échantillons, hors de l'appareil et avant l'expérimentation avec les 1000 applications installées sur le LG Nexus 5 de test. La méthode d'extraction des features ne change pas, mais les auteurs relèvent alors que l'analyse d'une APK prends en moyenne 64,474 secondes et que le fichier stocké pèse 5539 KiB.

Cependant Melis et al. [48] montrent que les méthodes basées sur l'apprentissage automatique (dans cet exemple Drebin, une méthode d'analyse statique) sont sujettes à des contournements de la part des auteurs de maliciels. La détection de malware se voit effectivement dans un modèle adversarial puisque l'expert en sécurité cherche à détecter le maliciel du pirate qui cherche à lui cacher. Si le fonctionnement de l'algorithme d'apprentissage du détecteur est connu du développeur de malware, il va chercher à sortir des groupes établis par ce dernier ou flouer les limites entre application dangereuse et bénignes. Il faut donc prendre en compte ce modèle et les suggestions d'améliorations apportées par Melis et al. dans le cas de l'utilisation de l'apprentissage automatique pour la détection de maliciel.

Support pour la méthode

Les articles suivants montrent les intérêts d'une application des méthodes précédentes aux téléphones réels plutôt qu'aux émulateurs, comment cette implémentation peut être faite, et comment gérer la mise en commun des données récoltées sur les appareils. Ils ne permettent pas d'offrir une comparaison ou distinguer des lacunes, mais les points soulevés par ces articles pourraient être mis à profit dans l'élaboration de la nouvelle méthode.

EMULATOR vs REAL PHONE [49] propose un système permettant d'effectuer les analyses adaptées aux émulateurs sur des appareils cellulaires réels avec quelques adaptations, afin de faire cette étude à l'échelle sans s'encombrer des différentes limitations dûes aux émulateurs. Il se base sur Dynalog pour l'extraction de feature et Monkey pour l'exploration automatique des applications.

BareDroid [50] est cité par l'article précédent. Il permet de rapidement formater et préparer

un téléphone pour lancer l'analyse d'un nouveau programme tout en évitant d'introduire un bruit inutile à cause des applications testées précédemment.

CrowDroid [51] présente une méthode relativement simple pour la détection. Mais sa grande innovation est de proposer une architecture clients/serveur pour le crowdsourcing des informations récupérés sur chaque téléphone de chaque utilisateur final afin de créer la base de donnée en ligne.

La conférence proposée par Yoon et al. [52] fait un peu figure d'exception ici car le matériel supportant la méthode n'est pas un téléphone Android. Cependant, il met en exergue sur Raspberry Pi l'utilisation de l'apprentissage automatique pour classifier des comportements légitimes et malveillants à partir de la distribution des appels systèmes, une des méthodes prometteuses de l'analyse dynamique que nous pourrons adapter pour Android. La différence et l'apport principal face à d'autres articles classifiant grâce aux appels systèmes de cet article est le suivant : la distribution des appels systèmes ne caractérise pas l'application entière pour déterminer si cette dernière est malveillante ou non, mais différents comportements de l'application pour chaque cluster de syscall. A partir de là, les comportements malveillants et les anomalies peuvent être identifiés pou repérer les maliciels.

L'article de Xiao et al. [10] traite du compromis, pour une méthode de détection lourde en calculs, entre un processing en local (sur l'appareil de l'utilisateur) et en ligne (offloading sur serveur distant). En effet, le premier problème qui se pose est de savoir si l'utilisateur a un accès garanti à internet pour un tel offloading, mais même si ce problème est réglé, il faut savoir si le temps de traitement gagné par le serveur distant compense les délais de communication entre appareil et serveur. Leur exemple pourra donc être reproduit si l'on cherche à maximiser l'efficacité d'une méthode qui fera des calculs à la fois sur l'appareil et sur un serveur distant (par exemple si la connexion internet n'est pas garantie). De plus, ils développent une méthode de détection mettant en place cette technique pour optimiser sa détection à base de Q-Learning. Ils comparent alors le simple Q-Learning, DynaQ (qui utilise des simulations pour accélerer le Q-Learning), et PDS (pour post-decision state qui fait des hypothèses sur le futur) utilisé sur un appareil utilisateur pour apprendre à trouver le meilleur compromis sans connaître les mobiles compétiteurs et les caractéristiques du serveur de calculs. Ils trouvent que c'est le PDS qui donne les meilleures performances. Bien que l'article se concentre sur cette optimisation et non sur la méthode de détection elle-même (une collection de traces pour comparaison avec une signature par analyse statique), ces recherches peuvent contribuer à l'optimisation de toute méthode qui doit considérer des calculs sur serveur distant mais aussi potentiellement sur l'appareil.

L'article de Sen et al. [53] parle de la coévolution des malwares et des méthodes de détections

de ses derniers sur mobile, dans un schéma d'optimisation adversarial. Ils décrivent l'évolution des malwares, puis l'évolution des méthodes de détection, et enfin la coévolution des deux. Les auteurs utilisent ces connaissances pour déterminer une nouvelle méthode de détection des malwares existants mais aussi futurs dont on pourra s'inspirer. La méthode obtenue par programmation génétique (évoluée seule) ne détecte que 48.44% des malwares évolués et 42.86% des malwares coévolués, mais les deux méthodes obtenues par coévolution détectent 100% de tous ces malwares. Cependant, elles présentent des taux de faux positifs allant de 5 à 12.09%, ce qui n'est pas acceptable pour une méthode de détection de malware. Cependant l'idée est bonne pour éventuellement prévoir les futures évolutions des malwares et renforcer une méthode déjà aboutie. Il faut noter que les méthodes ont été testées sur le dataset bien connu de Drebin, mais aussi sur celui de PRAGuard, contenant 609 malwares seulement mais plus récents.

2.3 Lacunes des méthodes existantes

Dans cette section, nous analyserons plus en détails les articles les plus proches de notre domaine restreint, et qui présentent un outil très ressemblant à celui que nous souhaitons réaliser. L'objectif ici sera de trouver les lacunes de ces méthodes, pour identifier les besoins auxquels devra subvenir notre nouvelle méthode. La connaissance d'articles connexes pourra être utile pour présenter des pistes de solutions, par exemple à partir d'articles qui divergent un peu du domaine d'étude mais ne présentent pas les lacunes identifiées.

MADAM (2012-2018) se présente comme une des solutions récentes les plus abouties pour la détection de malware temps-réel sur mobile. Ce framework utilise l'apprentissage automatique pour reconnaître les maliciels. Il les classifie à partir de différents comportements malicieux observés à différents niveaux d'Android : noyau, application, utilisateur et package. L'ancienne version de l'article citée par Alzaylaee et al. ne proposait une validation que sur quelques malwares analysés sur émulateurs [49], mais la version améliorée de 2018 offre des expérimentations plus poussées [40]. 2800 maliciels de 125 familles différentes provenant de trois bases de données ont été testés sur un appareil réel, donnant un taux de détection de 96.9%. De plus, 9804 applications légitimes ont été testées pour montrer que le taux de faux positif est très bas : $2.8 * 10^{-5}$. Enfin, des tests supplémentaires montrent que l'overhead de performance induit est de 1.4% et la durée de batterie consommée de 4%. Néanmoins, pour permettre de telles performances, MADAM inclut à la fois le *Global Monitor* et le *Per-App Monitor* pour détecter les comportements anormaux des maliciels, à différents niveaux, dont le noyau, et requiert pour cela *X-posed Installer apk* et le chargement d'un module de MADAM par `insmod`. La gestion des tentatives d'élévation de privilèges est gérée par le biais

de *Superuser apk*. Ces trois éléments demandent des privilèges root sur le téléphone utilisé, et les auteurs de MADAM précisent donc que d'une part, leur solution n'est pas destinée au grand public mais cherche à prouver la force d'une telle approche (multi-niveaux, dynamique, sur l'appareil) et éventuellement faire en sorte que les constructeurs de systèmes pour les smartphones intègrent MADAM dans leur OS. Ceci constitue donc la lacune la plus flagrante de ce système qui semble, en dehors de cela répondre à toutes les exigences d'un tel système et s'inscrit parfaitement dans l'optique de notre travail.

SafeGuard (2017) fonctionne aussi sur l'appareil (*on-device*) mais est relié à un serveur externe [38]. Il surveille le comportement des applications en temps réel et peut les bloquer, comme MADAM et contrairement à AppsPlayground par exemple qui fait une analyse dynamique automatique sur émulateur pour provoquer les comportements malveillants afin de classer le malware, ou aux anti-virus existants qui comparent les signatures des applications installées avec celles des malwares connus. Mais comme dit plus tôt, il y a quand même un serveur qui gère les règles de blocage et avec lequel communique l'application SafeGuard. Une connexion internet permanente est donc requise.Cependant, contrairement à MADAM, SafeGuard ne requiert pas les hooks noyau pour fonctionner, car ce sont les core lib qui appellent SafeGuard : il n'est donc pas nécessaire d'avoir un smartphone rooté (sous-entend, avec accès root) pour utiliser SafeGuard. Cependant, cet avantage est contrebalancé par un inconvénient : n'étant pas capable d'observer le comportement des applications au niveau noyau, SafeGuard ne protège pas son utilisateur des élévations de privilèges. En ce qui concerne les autres maliciels, les performances de SafeGuard ne sont pas non plus décrites dans le papier, ce qui rend la comparaison difficile à moins d'effectuer des tests par nous même. Ainsi, il ne figure pas dans le tableau 2.1.

ADroid (2017) propose aussi une détection directement sur l'appareil [39]. L'idée est d'éviter les analyses statiques lourdes et non exemptes de défaut, les analyses dynamiques automatiques soit incomplète de par leur parcours automatique, soit non scalable de par l'implémentation sur téléphone, ou détectables par les malwares lorsqu'elles sont sur émulateur. De plus, l'apprentissage automatique créant un overhead trop important, on cherche à en implémenter une variante légère aux performances tout aussi bonnes. ADroid ne requiert pas d'accès root mais a besoin de nombreuses permissions pour effectuer le monitoring sur l'espace applicatif Android. Comme SafeGuard, les hypothèses d'ADroid ne sont pas valides dans le cas d'un appareil compromis, et il ne détecte pas les élévations de privilège. ADroid a d'une part une liste noire de signatures des malwares à la manière d'un antivirus classique, et d'autre part une liste blanche utilisateur. Pour les applications qui n'appartiennent à aucune des 2 listes, les comportements sont observés et enregistrés pour créer un vecteur comportemental, qui sera comparé au comportement *normal* : celui-ci est fait lors d'un éta-

lonnage initial de l'application (dès qu'elle est installé, en n'installant que des applications sûres). En deux semaines de test (installation d'applications légitimes et de malware) le taux de faux positif est de 5.9% et le taux de détection de 97%. La méthode légère employé à une performance en $O(1)$ plutôt qu'un $O(n)$ voire $O(n^2)$ pour des systèmes d'apprentissage automatique classiques. La consommation de batterie est cependant non négligeable car elle s'élève à 10%, ainsi que le processing qui prends 0.53s chaque 5s pour générer le vecteur. Enfin, la méthode d'étalonnage et la validation sont discutables. Cependant, le fonctionnement d'ADroid donne des idées intéressantes et peut être amélioré : effectuer du clustering sur le vecteur peut servir à réduire sa taille ainsi que les faux positifs, inclure les syscall dans le monitoring peut améliorer la classification, et ajouter une détection hors-appareil en complément pourrait améliorer les performances quand le réseau le permet.

T2Droid (2017) présente lui aussi une solution d'analyse dynamique, mais sa grande nouveauté est d'opérer dans ARM TrustZone [5]. Cela permet une protection suffisante pour l'antivirus, que les maliciels pourraient vouloir compromettre. Malheureusement, son implémentation sur des téléphones classiques est plus compliquée, ce qui rend sa potentielle réalisation encore moins probable que MADAM. Dans l'article de Yalew et al., T2Droid est implémenté sur une carte i.MX53 QSB, qui a pour particularité d'offrir un libre accès la *secure zone* de ARM TrustZone. Les performances sont alors excellentes (0.98 ou 0.99 pour les valeurs de détections et 0.02 pour le taux de faux positif) mais l'expérimentation n'a été faite que sur 160 applications et le comportement humain n'a évidemment été que simulé avec Monkey. Le point soulevé par cet article d'intégrer le système de détection à la *secure zone* n'est cependant pas à écarter pour améliorer les solutions existantes.

La méthode d'Abdelfattah Amamra [54], se propose d'enregistrer les appels systèmes, ce qui peut se faire sur un smartphone Android quelconque, afin d'identifier les anomalies dans le comportement des applications et classifier malwares et applications légitimes grâce à l'apprentissage automatique. Afin de raffiner cette solution, la méthode va filtrer et regrouper (abstraction) ces appels pour réduire leur nombre et augmenter leur représentation. L'algorithme de classification optimal choisi est la machine à vecteurs de support (SVM) et le tout, bien qu'encore améliorable, permet de bonnes performances : En utilisant toutes les techniques développées au cours de la thèse (raffinement des traces **et** pondération des anomalies), le système arrive à une précision allant de 96 à 100% pour un taux de faux positif allant de 6 à 0%, selon le niveau d'anomalie et le nombre de modèles différents d'appels systèmes pour l'abstraction. Cependant, l'ensemble de validation utilisé reste réduit et le nombre d'applications légitimes utilisées n'est pas forcément représentatif : différents tests sont effectués pour les différentes améliorations apportées, mais les auteurs considèrent au plus 170 applications bénignes (top téléchargement du Store) et 100 maliciels (provenants

de Contagio [6]). La consommation de batterie du classificateur n'a pas été étudiée, mais le temps d'exécution du SVM avec les traces raffinées se situe à 257 ms (complexité en $O(n^2)$) et occupe 9 MB d'espace ($O(n)$) pour la phase d'entraînement (la phase de test étant une simple comparaison).

La méthode de Xiao et al. [55], d'après leur connaissance, est la première méthode d'apprentissage automatique pour l'analyse dynamique de malware Android utilisant le LSTM, pour Long Short-Term Memory. Ce modèle de langage permet de prendre en compte un contexte plus large (c'est-à-dire une fenêtre temporelle plus grande) pour l'enregistrement des appels systèmes, car les auteurs ont remarqué que certaines des méthodes précédentes étaient limitées par cela. L'intérêt d'enregistrer les appels systèmes sous de longues séquences est que cela permet d'extraire un maximum d'informations reliées au contenu sémantique de ces séquences, pour bâtir un modèle contextuel plus complet. Cependant seules 2 modèles seront bâtis : 1 malveillant et 1 légitime. Les auteurs consolident cette construction à l'aide d'un score de similitude, définissant si le comportement d'une application s'approche d'un comportement légitime ou malveillant en le comparant aux modèles.

En analyse dynamique, les approches basées sur la collecte des appels systèmes via strace sont très répandues, comme le montrent les 2 systèmes précédents, ainsi que les méthodes moins efficaces ou moins testées de Bhatia et Kaushal [41], Chaba et al. [56] et le travail préliminaire de Das et al. [57].

Monet (2017) [58] implémente une architecture client-serveur utilisant des signatures représentant le comportements de l'application en cours d'exécution. L'intérêt d'une telle signature est que, contrairement aux signatures générées par l'analyse statique, elle permettrait de détecter les variantes d'un même malware (puisque elle est déduite de son comportement à l'exécution et non de son code). La signature est générée sur l'appareil puis envoyée au serveur pour comparaison avec la base de données. La signature est générée en plusieurs étapes : d'abord un graphe statique est fait, puis il est complété par l'analyse de l'application à l'exécution, et on lui ajoute ensuite un ensemble d'appels systèmes suspects. L'application client intercepte les appels à l'aide de hooks et de modules noyaux, ce qui peut poser problème pour l'implémentation de cette dernière sur un appareil quelconque. Les *root exploit* semblent pouvoir être détectés grâce aux appels systèmes suspects. La méthode ne semble par contre pas a priori capable de détecter des malwares totalement nouveaux, mais uniquement des variantes de malwares existants. L'article présente de très bonnes performances, mais il faut bien noter les limites des expériences effectuées : une expérience (celle qui donne la précision et le taux de aux positif) se base uniquement sur la détection des variantes de DroidKungfu

6. `https://contagiodump.blogspot.ca/`

(avec des applications légitimes dans le set), et les suivantes sont sur des malwares originaux manuellement transformés en variantes et sur la consommation de batterie et la performance de l'application client.

SAMADroid [59] est une nouvelle méthode hybride à 3 niveaux : elle combine l'analyse statique et dynamique, le client sur appareil et le serveur distant, et l'apprentissage automatique. Les auteurs ont entraîné leur algorithme à partir du dataset de Drebin [60], qui comporte 5560 malwares de 179 familles différentes, ainsi que la bagatelle de 123453 applications bénignes extraites de différents stores. Les performances annoncées par les développeurs de SAMADroid sont remarquables : en utilisant les meilleurs algorithme d'apprentissage automatique, l'analyse statique a une précision de 99.07% pour un TFP (taux de faux positif) de 0.03% et l'analyse dynamique une précision de 82.76% pour un TFP de 0.1%. Néanmoins, il est à noter que l'analyse statique ne peut se faire ici que via le serveur distant, les performances annoncées ne sont donc pas garantie sans connexion internet stable. Enfin, la base de donnée utilisée contient de très nombreuses applications légitimes mais parfois moins de malware que d'autres datasets plus récents que celui de Drebin, qui, datant de 2014, peut ne pas inclure certaines familles de malwares plus récentes.

ServiceMonitor [61] est une méthode d'analyse dynamique sur appareil réel. Le dataset utilisé contient 4034 malwares pris de la base de donnée de Drebin et 10370 applications légitimes du Store Android. Les expériences faites à partir de cette base de donnée montrent une précision de 96% pour un TFP de 4.4%. Sur l'appareil, les overhead sont pour le CPU 0.8% et la mémoire 2%. Cependant, ServiceMonitor requiert un accès root pour implémenter sur téléphone. Son algorithme est entraîné sur émulateur, puis testé sur appareil physique.

Wang et al. [62] proposent une nouvelle méthode de détection de malware basée sur deux observations : l'usage et les anomalies des applications. Cette méthode prétend détecter les nouveaux malwares (c'est-à-dire ceux qui n'existent pas encore) utilisant des zero-day (des vulnérabilités inconnues à ce jour). Il est bon de noter que malheureusement la méthode proposée fonctionne grâce à une sandbox incluses dans un framework appelé CuckooDroid, et non sur appareil réel. Pour utilisation sur appareil réel, les téléphones feront appel au service d'infonuagique contenant l'émulateur et récupèreront les résultats par la suite. Bien qu'utilisant un émulateur, la méthode proposée utilisant le *Dalvik Hooking* datant de 2015 est plus résistante à l'évasion des malwares que les méthodes référencées (DroidDolphin, CrowDroid). Elle permet de relever des features par analyse statique et dynamique de l'application testée. Cet ensemble passe ensuite dans le *misuse detector*, qui est un cluster linéaire de vecteurs de support permettant de reconnaître un malware catégorisé ou un nouveau membre d'une des familles connues. Si ce détecteur donne un résultat négatif, l'ensemble est alors analysé par

le *anomaly detector* qui est une machine à vecteurs de support à une classe : si l'application analysée s'éloigne du comportement catégorisé (les applications bénignes) d'une valeur supérieure au seuil, il est classifié comme nouveau malware utilisant une nouvelle vulnérabilité (*zero-day*). Le détecteur de malware connu qui a un taux de faux positif bas et le détecteur d'anomalie qui permet de déceler des zero-day mais a un haut taux de faux positif sont donc complémentaire et leur combinaison paraît comme une bonne idée. En effet les performances obtenues sont très bonnes, avec un taux de détection aussi haut que 98.76% pour un taux de faux positif de seulement 2.24% et un taux de faux négatif de 1.24%. La méthode est validée sur un ensemble de 5560 maliciels issus de la base de données de Drebin et 12000 applications bénignes provenant de marchés d'applications chinois et validées par VirusTotal.

Afin de comparer les résultats des prototypes considérés et la confiance qu'on peut leur attribuer, nous dressons le tableau 2.1 regroupant plusieurs des caractéristiques quantitatives décrites ci-dessus. À des fins de comparaison, nous faisons apparaître les résultats de IntelliAV et BAdDroIds, deux systèmes basés sur l'analyse statique mais largement testés et disponible sur le Google Play Store. Il est à noter que les deux premières colonnes (taux de détection et de faux positif) seront notre premier critère et ce que nous appellerons précision de la méthode, tandis que les colonnes suivantes (nombre de malwares et d'applications testés) concernent la validation de ces chiffres, avec un ensemble d'entraînement/de test conséquent ou non. Les deux dernières colonnes (overhead et consommation de batterie) permettent de comparer les performances en tant que telle du logiciel. On entend alors par performance l'impact du logiciel sur les composants du téléphone. Les informations sur la consommation de batterie seront données en pourcentage lorsque ceci est directement indiqué dans l'article étudié comme étant la batterie consommé par le logiciel sur une charge complète. L'*overhead* va correspondre à l'impact sur les composantes du téléphone en général (et inclus parfois la consommation de batterie dans son total). Dans le cas des méthodes d'analyse dynamique, nous considèrerons la mesure pertinente et donnée par les articles les plus intéressants comme étant aussi un pourcentage : il s'agit de la différence, selon que le logiciel d'analyse de logiciel malveillant est présent et actif sur l'appareil testé ou non, d'utilisation du processeur, de la mémoire vive, de la batterie et autres ressources du téléphone, en moyenne. Dans le cas des méthodes d'analyse statique (IntelliAV et BAdDroIds) cette mesure n'est pas pertinente car l'analyse induit généralement un coût fixe pour chaque application analysée, et nous gardons alors en secondes le temps requis pour effectuer une prédiction sur une application. Pour BadDroIds nous indiquons aussi l'espace requis en mémoire pour l'analyse. Les cases du tableau 2.1 remplies avec *N/A** signifient qu'une donnée potentiellement pertinente est fournie dans la littérature mais que nous n'avons pas modifié sa forme pour homogénéiser la colonne. Nous pensons donc qu'il serait pertinent pour une étude plus poussée de pouvoir

homogénéiser les données et combler ces cases, voire refaire les tests présentés dans les articles, si tant est que les logiciels cités soient disponibles au public.

Tableau 2.1 Comparaison des articles existants

Nom	Taux de détection (%)	Taux de faux positif (%)	Nombre de malware testés	Nombre total d'appli testées	Overhead	Consommation de batterie
MADAM	96.9	0.2	2800	12604	1.4%	4%
ADroid	97	5.9	480	720	N/A*	10%
T2Droid	98-99	2	80	160	N/A*	N/A
Amamra	96-100	0-6	100	200	N/A*	N/A
LSTM	96.6	9.3	3567	7103	N/A	N/A
Monet	99	0	3723	4223	7%	3%
SAMA-Droid	99.07/82.76	0.03/0.1	5560	129013	0.6%	N/A*
Service-Monitor	96	4.4	4034	14404	0.8-2%	N/A
Cuckoo Droid hybrid	98.76	2.24	5560	17560	N/A	N/A
IntelliAV	92.5	4.2	9664	19722	4-16 s	N/A
BAd-DroIds	98.9	0.6	7494	14988	64,67 s, 5539KiB	N/A

On peut observer grâce à ce tableau comparatif des quelques méthodes récentes étudiées plusieurs points intéressants. Tout d'abord, nous constatons que la plupart des méthodes ont un taux de détection bon, voire très bon : la méthode la plus fragile a déjà 92.5% et certaines méthodes vont jusqu'à 100%, d'après les auteurs. Il ne faut pas s'en étonner, car en général les prototypes ne présentant pas un taux de détection satisfaisant ne feront pas l'objet d'un article. Cela s'explique par le fait que c'est la mesure principale pour juger de la qualité des méthodes. C'est pourquoi pour une grande partie des articles nous sommes obligés d'aller consulter les autres colonnes pour connaître les véritables lacunes de la méthode et ce qui peut être substantiellement amélioré.

En premier lieu, il est important de remarquer le taux de faux positif : celui-ci peut rapidement invalider la méthode, quelque soit le taux de détection annoncé. En effet, il est tout à fait possible d'avoir 100% de détection dès lors qu'on considère toutes les applications comme malveillantes. Dès lors, le taux de faux positif sera aberrant, et la méthode ne sera pas du tout utilisable sur un appareil réel utilisé normalement. Il est bien évident que cette situation extrême est absente des articles choisis, mais certains taux de faux positifs comme 5.9% ou 9.3% menacent déjà la validité de la méthode. On cherchera idéalement à atteindre

un taux de détection de 100% avec un taux de faux positif de 0%, mais dans la réalité il s'agira souvent d'un compromis entre ces 2 valeurs. La distinction dans cette zone de floue pour la méthode qui oblige à faire ce compromis peut cependant être confiée à l'utilisateur : Lorsque le jugement de la machine ne permet plus de distinguer un malware d'une application légitime, l'utilisateur peut parfois en être capable, ou au moins comprendre les risques liés à l'installation d'un logiciel non identifiable.

Cependant, ces derniers chiffres n'ont de valeur que lorsqu'ils sont associés avec les suivants : le nombre d'applications légitimes et malveillantes testées. Il faudrait même idéalement préciser lesquels (ou au moins leur source), notamment pour vérifier la présence de toutes les familles de malware connues, surtout les récentes, et la popularité des applications légitimes choisies. En effet, une méthode qui fonctionne sur des malwares vieux ou spécifiques uniquement pourrait ne pas être applicable aux innovations des pirates informatiques. Une méthode qui ne considère que quelques applications légitimes parmi les plus connues est-elle capable de distinguer une application moins populaire d'un logiciel malveillant ? Le taux de faux positif indiqué est-il représentatif ? La méthode scientifique voudrait que nous puissions tester toutes les méthodes sur une même base de donnée la plus complète possible. Quelques articles ont effectivement testés leurs méthodes sur les mêmes bases, mais cela nous obligerait alors à utiliser cette dernière, qui peut parfois être datée. Pour effectuer les tests nous-mêmes sur une base récente, il faudrait disposer du code source ou au moins de l'application, ce qui est malheureusement rarement le cas pour les méthodes récentes. Enfin, si l'on considère que la méthode développée doit être capable de distinguer non pas seulement des malwares connus (ce que toutes font déjà ou presque) mais aussi de futurs malwares non encore classifiés, la méthode de validation doit être appropriée : un des exemples vu dans la littérature serait d'entraîner un classificateur sur des malwares datant d'avant une date précise, puis de la tester sur une base contenant des malwares plus récents que cette date. Certaines méthodes sont aussi plus adaptées aux nouveaux malwares, utilisant des *zero day*, comme celles à base de détection d'anomalie, mais ont leurs propres défauts, notamment un haut taux de faux positif.

Les colonnes suivantes découlent d'un second aspect à considérer pour l'usage en conditions réelles de la méthode. Il s'agit des performances de l'application sur l'appareil. En effet, une telle application peut, si elle n'est pas optimisée ou dépend de beaucoup de données et calculs, vite se ressentir à l'utilisation de l'appareil. Il faut donc tenir compte de son impact sur le CPU et la RAM (ou bien sur les délais ajoutés) et la batterie. Malheureusement, certains articles n'indiquent pas ce type de donnée, parfois parce que la méthode n'est testée que sur émulateur. De plus, les données sont fournies de manière différente suivant les articles, ce qui peut poser problème pour la comparaison, surtout lorsque les chiffres indiqués sont

dépendants de l'appareil de test. Dans tous les cas, des valeurs trop hautes peuvent vraiment handicaper l'utilisateur et doivent être réduites.

CHAPITRE 3 REQUIS ET ARCHITECTURE

Une fois la revue de littérature établie, nous pouvons définir un ensemble de requis pour la construction de notre propre méthode. Nous décrirons donc en premier lieu ces requis. A partir de ces requis, il sera possible de déterminer une architecture idéale pour, nous l'espérons, une précision et une performance optimale tout en respectant les contraintes de l'environnement du projet. Les détails de l'implémentation des différentes itérations du prototype seront ensuite décrites. Nous décrirons le prototype moins ambitieux qui sera développé dans ce mémoire, et nous montrerons les premières étapes de son développement, avec notamment la récupération et la mise en forme des données pour notre étude d'apprentissage automatique.

3.1 Requis idéaux pour l'architecture

L'étude des articles précédente a permis de mettre en lumière plusieurs lacunes récurrentes, que notre propre prototype devra corriger. La liste suivante détaille ces dernières :

- Précision
  - Taux de détection : Puisque le but premier du système est de détecter les malwares, nous cherchons à maximiser le taux de détection, sans quoi son utilisation est inutile.
  - Taux de faux positif : Le taux de détection précédent doit nécessairement être comparé au taux de faux positif engendré par le système de détection. Si celui-ci est trop grand, l'utilisateur ne pourra plus se fier au prototype, qui risque de bloquer des applications légitimes trop souvent pour être utilisable.
  - Validation : Certains des articles prometteurs ne proposent pas une validation assez aboutie de leur prototype, ce qui ne permet pas une assez grande assurance pour l'utilisateur finale, ou une comparaison correcte avec les autres articles.
- Facilité d'intégration aux systèmes existants : Dans le cadre d'une utilisation à grande échelle par de nombreux utilisateurs lambdas, la facilité d'intégration du système de détection est une des conditions nécessaires. Idéalement, nous souhaitons qu'un utilisateur quelconque puisse installer l'application sur tout smartphone Android à jour à partir du store officiel.
  - Accès root requis : Plusieurs solutions parmi les plus prometteuses utilisent des hooks au niveau du noyau, généralement pour construire un modèle plus précis et détecter les tentatives d'élévations de privilège : On utilise généralement les applications SuperUser et Xposed à cet effet. L'accès root n'est pas inclus par

défaut sur un smartphone Android neuf, et l'utilisateur lambda ne peut donc pas installer une telle application.
    - Modification du matériel requise : D'autres systèmes vont encore plus loin en demandant des smartphones ou des cartes spéciales et intègrent leur système directement à une partie du matériel accessible au fabriquant mais pas à l'utilisateur. Ainsi leur prototype ne peut être présent que sur du nouveau matériel et avec l'accord du fabriquant pour une intégration dans leurs nouveaux modèles, ce qui suggère une intégration à très long terme.
    - Accès internet requis : Parfois, les systèmes proposés se composent à la fois d'une application client installée sur le téléphone intelligent et d'un serveur distant pour les traitement lourds. L'utilisateur devrait pouvoir être protégé en tout temps par le système, sans avoir besoin d'une connexion internet permanente. Néanmoins, on peut décemment supposer que l'utilisateur aura accès à internet au moins lors de l'installation de nouvelles applications, et il est possible de proposer un système à performances réduites en hors-ligne.
- Performance
    - Overhead et utilisation de la batterie : Néanmoins, le système doit être assez léger pour être utilisé sur smartphone sans empêcher l'utilisation normale de son propriétaire, en réduisant de trop la durée de vie de la batterie ou en augmentant trop les temps de réponse de l'appareil.
- Non-détection des zero day, anomalies ou nouveaux malwares n'appartenant pas à une famille connue. Un cas récurrent dans la littérature est la non-détection des élévations de privilège : Il semble que plusieurs des prototypes soient incapables de détecter certaines actions des malwares lorsque ceux-ci accèdent au noyau de l'appareil alors qu'eux-mêmes n'y ont pas accès. Il faut donc trouver un compromis donnant des permissions supérieures ou égales au système de détection qu'aux logiciels malveillants que l'utilisateur est capable d'installer. Il est à noter que seulement une partie des élévations de privilège accèdent au noyau, et que celles-ci reposent sur l'exploitation de vulnérabilité dans le système Android même. Cependant, les méthodes de détection doivent être capables de prévenir les autres types d'élévations de privilèges au niveau applicatif, puisqu'elles peuvent être introduites par les applications [63].

D'après l'étude de ces lacunes, nous allons pouvoir définir un ensemble de conditions nécessaires et de variables que nous pouvons améliorer dans notre propre architecture. Cependant il va falloir sélectionner des améliorations qu'il est possible d'apporter dans le cadre de la maîtrise. Nous pourrons par la suite comparer les performances de la nouvelle architecture avec celles décrites dans les articles de la littérature.

3.2 Identification des avenues de recherche et solutions possibles

Le récapitulatif du chapitre précédent a permis de déterminer 3 grandes avenues de recherche que nous pourrions creuser pour développer notre propre architecture. Il s'agit de grands axes qui proposent de combler les lacunes existantes, et qui chacun vont dans une direction différente. Nous les présentons ci-dessous :

Tout d'abord, l'un des points faibles de la majorité des méthodes de la littérature, bien que souvent caché, est l'impossibilité de garantir une détection des futurs malwares. En effet, de nombreuses méthodes s'appuient sur les malwares existants (et préalablement détectés !) pour construire leur modèle de prédiction, ce qui ne leur permet généralement que de détecter des logiciels similaires. L'une des méthodes intéressantes pour détecter un malware nouveau dans le cas où la reconnaissance échoue est la détection d'anomalie. Cependant cette dernière comporte un haut taux de faux positif. On rencontre donc aujourd'hui des méthodes hybrides afin d'allier le meilleur des deux mondes pour laisser passer un minimum de logiciels malveillants entre les mailles du filet de l'antivirus. Il faut donc prévoir une flexibilité de la méthode si on veut profiter de ces deux techniques : par exemple, lorsque une application n'a pas pu être classée avec une certitude suffisante comme malware ou bénigne, on l'analyse avec le détecteur d'anomalie. Il est aussi possible, dans une optique d'amélioration continue de demander l'avis de l'utilisateur lorsque la méthode ne peut pas garantir la prédiction avec assez de certitude. De plus, ce type d'implémentation est potentiellement plus léger et s'associe bien avec un modèle de base d'apprentissage crowdsourcée.

Ensuite, un aspect récurrent des méthodes sur cellulaire précédentes est la difficulté de communier précision et performances pour certaines méthodes, notamment basées sur l'apprentissage automatique. En effet, les méthodes qui obtiennent le plus de précision (un maximum de malwares détectés et un minimum de faux positif) demandent très souvent de lourds calculs et ont donc de mauvaises performances (grande consommation de batterie et temps de latence induits par le CPU du téléphone). Certains compromis existent avec des méthodes légères mais aux performances approchants les méthodes lourdes. Il est aussi possible, dans le cadre d'un plus gros projet, de mettre en place des architectures hybrides client-serveur. Le client dispose d'une méthode de détection légère et le serveur, qui est plus performant, de méthodes plus précises. Il faut alors considérer le problème de l'offloading client-serveur, c'est-à-dire de savoir quand il est plus efficace d'envoyer les données au serveur (en tenant compte de la bande passante disponible et de l'activité du serveur, dans le cas d'un serveur partagé entre tous les utilisateurs) et quand il est plus efficace d'effectuer tous les calculs sur le téléphone. Bien sûr, dans sa forme la plus primitive, le dispositif hybride permet d'effectuer les calculs en local lorsque l'appareil est hors-ligne, et sur un serveur partagé sinon. Une amé-

lioration possible pour répondre à ce problème de disponibilité des réseaux est l'utilisation des réseaux mobiles pour envoyer des données via le canal USSD par exemple.

Enfin, il est possible avec les bons accords commerciaux et les partenariats industriels d'intégrer directement la méthode de détection au matériel qui sera fourni. Nous rappelons que cette maîtrise s'inscrit dans le cadre du projet ATISCOM qui vise à développer une architecture multi-service centrée autour du paiement mobile. On peut imaginer que les terminaux de vente sont produits et personnalisés par l'entreprise partenaire et que nous pouvons utiliser des méthodes innovantes, comme T2Droid basé sur TrustZone d'ARM, ou l'accès aux modules de sécurité hardware des téléphones de nouvelle génération pour des calculs cryptographiques accélérés. Dans une moindre mesure, on peut aussi considérer des méthodes qui requièrent l'accès root sur l'appareil voire une modification du firmware, dans le cas où le terminal est directement distribué avec le client pour l'architecture ATISCOM.

On remarque avec l'analyse précédemment effectuée des articles récents qu'il y a peu d'innovation pour pallier aux lacunes existantes. En général, il s'agit surtout d'améliorer les méthodes existantes et de chercher de meilleur compromis. Les méthodes qui semblent se démarquer au niveau de la précision et de la performance vont bien souvent être des méthodes hybrides, voire hybrides à plusieurs niveaux. Cela semble être une des meilleures voies pour développer un système robuste et utilisable en production. Nous pouvons nous inspirer de méthodes existantes et chercher à avoir la plus grande flexibilité possible et les meilleurs compromis dans un modèle hybride afin que chaque aspect de la méthode compense les lacunes d'un autre. L'objectif est alors de maximiser la précision et la prévoyance de notre détection sans pour autant sacrifier la performance ou la praticabilité.

Pour la suite de ce mémoire, nous chercherons donc à développer une solution optimale centrée sur la flexibilité et l'adaptabilité de la méthode. Il s'agira de mettre en place un modèle client-serveur, le client ayant un fonctionnement hybride capable de détecter les malware avec différents degrés d'efficacité et des performances variables selon qu'il est hors-ligne, sur réseaux mobiles ou en ligne, au mieux de ses capacités. Nous allons maintenant étudier les contraintes et variables de la définition des requis de cette nouvelle méthode. L'avenue de recherche précédemment décrite choisie ici est donc la dualité client-serveur et la gestion de l'offloading, étant donné que les contraintes qui entourent le projet peuvent être incompatibles avec l'utilisation du hardware ou noyau du téléphone, et la détection d'anomalies.

3.3 Conditions nécessaires et variables à optimiser

En partant sur cette voie, nous pouvons tout de même définir quelques requis pour la méthode, qui permettront d'encadrer et limiter notre travail, tout en offrant une certaine garantie pour la suite du projet et l'industrialisation du prototype.

A partir de l'étude précédente, nous pouvons définir des conditions nécessaires que devra respecter notre nouvelle architecture :

- Le système de détection doit pouvoir être installé sur tout smartphone Android neuf disposant d'un accès à internet par le biais du Store officiel, sans modification du matériel ou du logiciel de base. La version exacte sera à préciser avec les partenaires industriels, néanmoins pour le prototype, on peut se baser sur la version d'API 15, correspondant à la version 4.0.3 d'Android, surnommée IceCreamSandwich, puisque Android Studio indique qu'environ 100% des appareils roulent cette version ou une plus récente.
  - Ceci exclut donc l'installation du logiciel par le fabricant, par exemple pour l'inclure dans des parties protégées du système comme la TrustZone d'ARM. Tout cela permettra d'inclure le projet dans tout type d'architecture avec aucune modification du matériel requise pour le système de détection.
  - On considère aussi que l'appareil n'est pas *rooté*, et que les permissions de l'utilisateur sont celles par défaut. Ceci pour garantir une compatibilité plus grande encore. Cependant, il faut bien réaliser que de nombreuses méthodes parmi les plus performantes et notamment celles d'analyse dynamiques existants dans la littérature reposent sur cet accès. Il faut donc s'attendre à des performances suboptimales dans le cas d'une analyse dynamique. Si les conditions du projet viennent à changer et qu'un accès root sera requis pour l'architecture dans son ensemble, il semble important de préciser que le prototype de détection peut très certainement être amélioré en tirant parti de cela.
- L'accès à internet n'est nécessaire que pour l'installation de l'application. On ne peut pas considérer que l'utilisateur bénéficie d'un accès à internet stable et garanti une fois cette dernière installée, et elle doit donc pouvoir fonctionner sans par la suite. On remarquera néanmoins que dans beaucoup de cas, la prédiction se fera dès l'installation d'une nouvelle application voire avant dans le cas d'une archive .apk disponible sur l'appareil. Il y a alors de grandes chances que l'utilisateur ait accès à internet à ce moment. Les cas où cela n'est pas applicable sont a priori :
  - Une détection très longue par le biais par exemple de monitoring du comportement de l'application sur une durée étendue. Cela présente les risques que l'application

effectue des actions malveillantes sans que nous puissions l'arrêter puisque nous voulons les observer. Cela se rapproche aussi de la détection d'anomalie que nous allons laisser de côté pour ce travail.

- L'utilisateur reçoit la nouvelle application par un autre biais, par exemple transfert via un autre téléphone ou un ordinateur par connexion directe ou des réseaux de courte portée, comme WiFi (non relié à internet) ou Bluetooth. Bien souvent l'application sera visible en tant qu'APK sur l'appareil utilisateur avant d'être installée.
- Ces cas restent minoritaires mais dans le cadre d'un projet de sécurité, ne peuvent être simplement ignorés. Il faut cependant mesurer les efforts requis pour couvrir ces cas. Une solution de facilité si la méthode de détection ne peut s'effectuer hors-ligne est d'avertir l'utilisateur que la prédiction n'a pas pu être effectué en hors-ligne et lui suggérer de retrouver un accès à nos serveurs avant d'installer une application suspecte non analysée.

— Le système de détection doit avoir une interface simple et être non-intrusif pour l'utilisateur, privilégiant l'automatisation au jugement humain. En effet, bien qu'il soit possible d'intégrer le jugement humain dans le système pour corriger ou confirmer certaines prédictions en cas d'incertitude, nous préférons éviter ceci. L'appel à l'utilisateur (qui est, a priori, non qualifié) va potentiellement réduire sa confiance dans le système et aussi le déranger pendant ses activités (dans le cas où il n'installe que des applications légitimes, car sinon, on préfère l'interrompre et le protéger) ce qui peut baisser son appréciation du logiciel voire le désinstaller.

Les variables, qui sont quantifiables et que nous chercherons à optimiser, sont alors les suivantes :

- Précision
  - Un taux de détection des maliciels s'approchant ou dépassant 97% est notre objectif, puisque on cherche à obtenir une méthode supérieure ou égale à ADroid et MADAM en terme de performances et gestion des contraintes.
  - Cependant, le taux de faux positif est également à prendre en compte. Un taux trop grand invalide les résultats de la méthode : nous devons trouver le meilleur compromis entre haut taux de détection et faible taux de faux positif, idéalement pour atteindre ceux de MADAM, 96.9% et 0.2%.
- Performance
  - Même avec un taux de détection extrêmement bon, la performance constitue une contrainte si elle n'est pas optimisée : puisque l'utilisateur doit être capable de se servir de son appareil sans intrusion de l'application, celle-ci ne doit pas non plus

ralentir les autres applications de manière visible. Les temps de calcul acceptables sont variables selon la présence de l'application. Nous devons trouver un moyen de comparer cette performance aux autres méthodes.
    - La consommation de batterie ne doit pas être prise non plus à la légère. Une consommation trop importante va réduire la durée d'utilisation de l'appareil et jouer sur la propension de l'utilisateur à conserver l'application de détection.
- Validation
    - Nous devons maximiser le nombre d'applications testées par notre système : Les 19722 applications analysées par IntelliAV ont plus de poids en validation que les 720 d'ADroid.
    - La quantité et la variété de malwares parmi ces échantillons doit aussi être maximisée, tout en conservant une part importante d'applications légitimes afin de ne pas perturber les résultats : dans le cas d'ADroid, 66% des applications analysées sont malveillantes, alors qu'entre 22 et 50% le sont pour les validations de grande échelle. Dans la réalité, la proportion de malware parmi les applications légitimes est réduite.

3.4 Construction du prototype de base

En premier lieu, nous nous attacherons à implémenter un prototype d'une méthode de détection de base, la plus modulaire possible, pour tester différentes hybridations et implémenter des algorithmes basés sur la flexibilité. Ce prototype de base pourra être inspiré de méthodes existantes.

Les modules dont nous aurons besoin pour tester la flexibilité sont :

- Une méthode de détection sur serveur
    - Autonome
    - Recevant des entrées d'un client mobile
- Une méthode de détection sur client
    - Autonome
    - Communiquant des informations au serveur distant
- Un système de communication client-serveur flexible
    - Communication Android-Serveur (Linux) par IP
    - Communication par les réseaux mobiles
        - Interface application-SIM
        - Passerelle serveur-réseaux mobiles
        - Traitement adapté des informations sur serveur et client

Les différents systèmes de détection peuvent être de même type ou non, éventuellement hybride. Le serveur a l'avantage d'avoir une meilleure capacité de calcul mais aussi de stockage d'une base de donnée, ce qui le rend plus performant pour les méthodes basées sur l'apprentissage automatique, mais aussi adaptées à la reconnaissance de signature. Dans tous les cas, il peut facilement supporter les méthodes d'analyse statique tandis que les méthodes d'analyse dynamique ne peuvent être faites entièrement sur le serveur qu'en utilisant un émulateur.

Cependant, puisque nous avons à disposition un client, il est possible d'y appliquer des méthodes d'analyse dynamique plus réalistes. Il est aussi possible d'appliquer des reconnaissances de signature ou des extractions de caractéristiques pour l'analyse statique mais il peut connaître les limitations suivantes : espace de stockage pour la base de malwares connus trop petit, et capacités de calcul trop basses pour l'apprentissage automatique.

Dans un premier temps, on mettra en place des méthodes existantes efficaces et connues qu'on modifiera par la suite afin de les adapter au prototype, voire remplacera par des méthodes plus adaptées à la communication et aux forces de chaque machine.

Pour la communication client-serveur, on pourra s'inspirer de travaux existants pour la communication IP, mais il faudra peut être implémenter nous-même la communication sur les réseaux mobiles. Notre première approche sera d'utiliser les USSD.

Un schéma grossier de l'architecture envisagée est présenté en figure 3.1. Il représente d'une part l'appareil mobile de l'utilisateur final, équipé d'une méthode de détection simple, et d'autre part le serveur distant, équipé d'une méthode de détection plus performante. Ces deux entités sont reliées par les réseaux mobiles et internet, puisque le serveur se trouve dans le nuage. Cela résume donc les 3 challenges à relever pour la construction de l'architecture imaginée : d'une part la méthode de détection sur périphérique mobile, d'autre part la méthode de détection sur serveur, et enfin la communication entre ses deux entités qui doit être optimisée et balancée en temps réel pour garantir la précision et la performance de la détection dans tout environnement.

3.5 Méthode de détection sur périphérique mobile

Dans un premier temps, en dehors de quelques tests pour vérifier la faisabilité des méthodes sur serveur et des communications client-serveur, nous nous concentrerons sur les méthodes pour périphérique mobile hors-ligne. En effet, les méthodes les plus simples à développer ne sont généralement pas les plus gourmandes en ressources. On fait alors l'hypothèse que tous les calculs et le stockage peuvent être gérés sur l'appareil même, et qu'envoyer des instructions au serveur pour qu'il s'occupe de ces derniers ne fera que ralentir l'exécution.

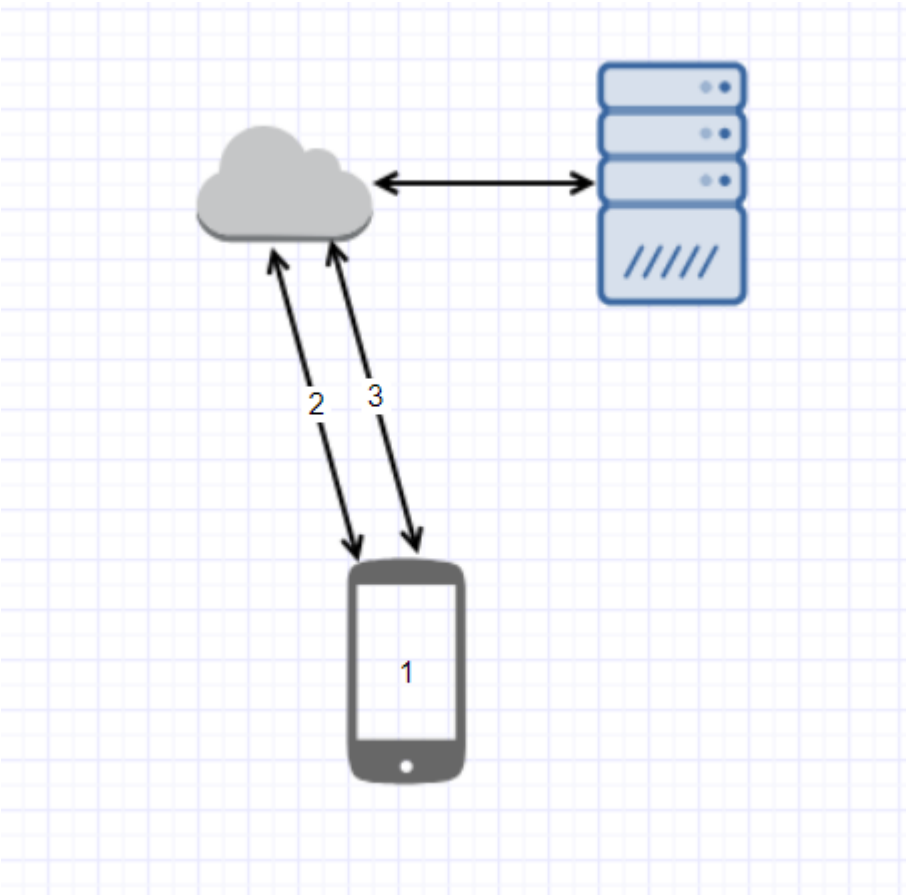


Figure 3.1 Schéma simplifié de l'architecture client-serveur

3.5.1 Abandon de l'analyse dynamique

La méthode utilisée actuellement est de type statique, car après étude des possibilités techniques pour l'analyse dynamique, nous réalisons que les différentes options disponibles en l'absence de code source utilisable ne respectent pas les contraintes ou pourraient faire l'objet d'une maîtrise à eux seuls. En effet, il est difficile dans le système Android de tracer le comportement d'une application, par exemple via les appels systèmes, sans effectuer une de ces deux modifications que nous nous interdisons : obtenir les droits administrateurs sur l'appareil afin de monitorer une application spécifique, qui sera roulée avec un utilisateur différent de notre antivirus, ou modifier l'application à observer (comme pour le *taint tracking*). Cela serait assez faisable cependant sur un serveur grâce à l'émulation, malgré les problèmes que cela peut impliquer. Mais lorsqu'il s'agit d'effectuer l'analyse sur l'appareil même en respectant nos contraintes, la seule solution à notre connaissance est de développer une application de *sandboxing* afin de lancer l'application à analyser à travers cette dernière et ainsi pouvoir tracer tous ses appels. Cela représente de nombreux enjeux de développement Android et est susceptible de rencontrer les mêmes barrières que l'émulation sur serveur.

3.6 Description de la méthode

La méthode d'analyse statique utilisée repose sur deux choses. D'une part, les permissions du système Android, que nous pouvons extraire du Manifest.xml des applications installées sur l'appareil ou des archives jar ou apk d'applications visibles sur le système de fichiers. D'autre part, un modèle d'apprentissage automatique construit grâce à une grande base de données d'applications malveillantes et légitimes, et du classificateur naïf de Bayes pour l'apprentissage supervisé.

3.6.1 Collection des caractéristiques pour l'apprentissage automatique

D'après le site officiel du développement Android, il existe 151 permissions du systèmes Android, qui sont les permissions pré-existantes que les développeurs peuvent déclarer dans le fichier Manifest afin de permettre l'interaction de leur application avec différentes parties du système. De nombreux articles de recherche en détection de logiciel malveillant utilisent les similitudes dans les permissions des malwares pour les caractériser et les détecter. En effet, les permissions déclarées dans le Manifest permettent d'avoir une vue d'ensemble des parties du système auxquelles l'application analysée peut (et vraisemblablement veut) accéder. Par exemples, de nombreuses applications malveillantes demandent la permission d'envoi de SMS (en général pour texter des numéros surtaxés et ainsi voler de l'argent aux victimes), alors

que peu d'applications légitimes le font, et sont généralement reconnaissables. La combinaison de certaines permissions peut aussi donner un indice sur des comportements suspects. Cependant, même si les permissions demandées donnent un indice sur le comportement de l'application et lui imposent un certain cadre (pas d'envoi de SMS sans permission l'autorisant, en général), elles ne sont pas obligatoirement symptomatiques du comportement de cette dernière. Ainsi, une permission demandée n'est pas forcément exploitée, ce qui peut mener à brouiller la classification entre applications légitimes et malveillantes par abus des demandes de permission (ce qui est déconseillé pour tout développeur honnête). De plus, cette méthode ne détecte pas les logiciels malveillants qui coopèrent, que cela soit avec d'autres applications du même marché, entre des versions consécutives d'une même application, ou via une application téléchargée par le malware initial. Enfin, comme de nombreux articles le font remarquer, l'envoi de SMS et l'accès à internet sont symptomatiques des malwares simples existants aujourd'hui, ce qui produit beaucoup de faux positifs avec les applications de messagerie, par exemple. Notre système actuel n'échappe malheureusement pas à ces critiques.

Pour obtenir les permissions des applications utilisées pour la base de donnée, nous utilisons l'outil multi-plateforme en ligne de commande `aapt dump permissions <app.apk>`. Nous comparons ensuite ces permissions avec la liste des permissions officielles du système Android pour obtenir un vecteur de booléens indiquant la présence ou l'absence des permissions. Nous indiquons aussi si l'application est un malware ou une application bénigne, dans le cas de la construction du modèle. Sur le téléphone, la classe PackageManager permet de retrouver le manifeste et les permissions associées aux applications installées ou à des archives lisibles sur le système de fichier. Une représentation schématique de l'entraînement de notre modèle est montrée en figure 3.2 et son utilisation sur le téléphone en figure 3.3.

3.6.2 Base de données

Nos recherches ont permis de trouver plusieurs bases de données de logiciels malveillants disponibles en libre accès ou seulement aux chercheurs sur le web. Nous avons collecté celles de theZoo [1] et contagio [2] qui sont parmi les plus récentes, pour tester notre modèle. Afin de le construire, nous avons pu récupérer une base de donnée bien plus importante (mais aussi un peu plus datée) : il s'agit des quelques 5560 logiciels malveillants provenant de 179 familles différentes de la base de Drebin [60]. Ces bases ne fournissent malheureusement pas de logiciels légitimes pour construire une base équilibrée. En effet, il faut pour bâtir notre

1. http://thezoo.morirt.com/
2. https://contagiodump.blogspot.ca/

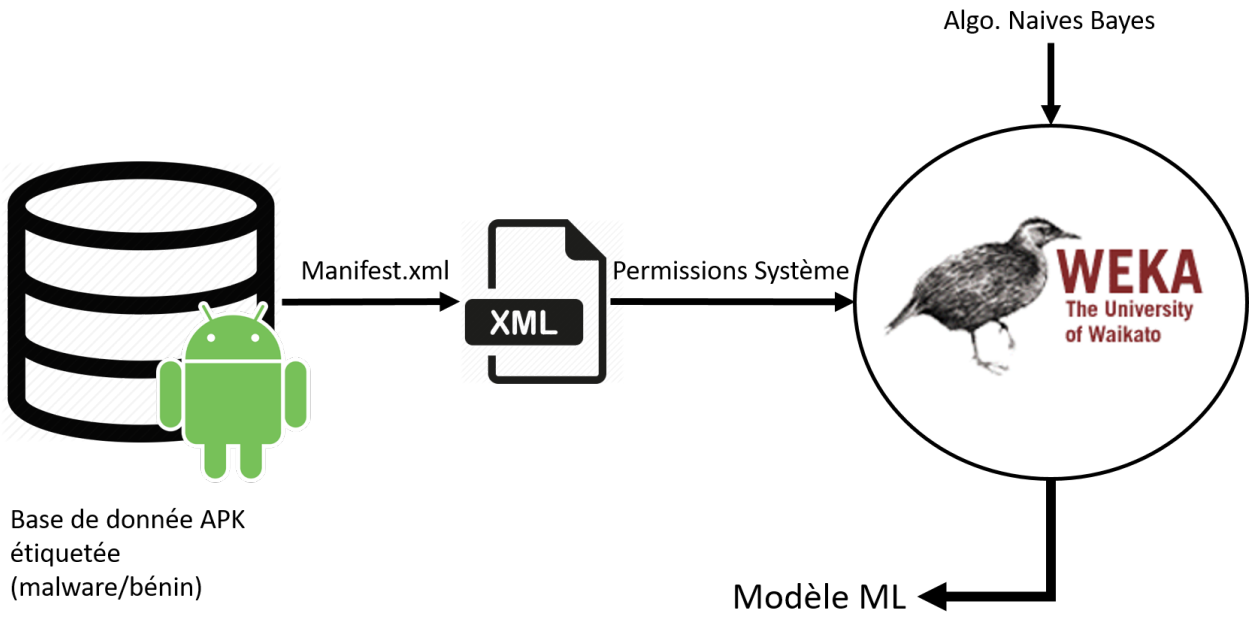


Figure 3.2 Représentation schématique de l'entraînement du modèle

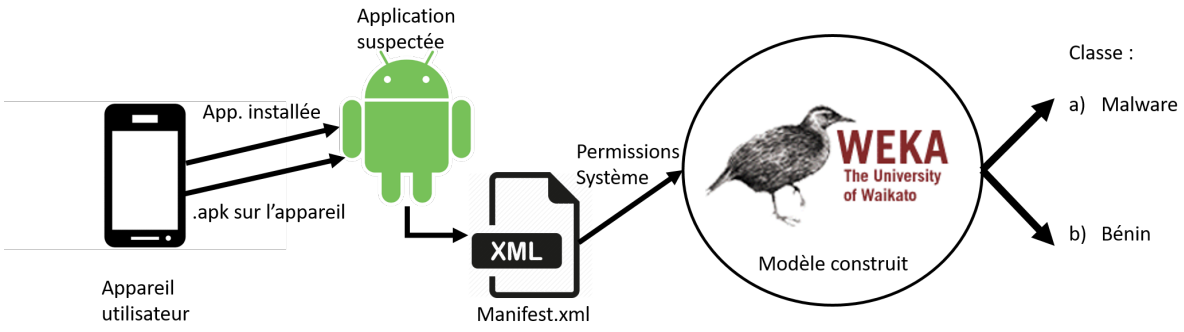


Figure 3.3 Représentation schématique de la prédiction par le modèle

modèle au moins autant de logiciels bénins que de malwares puisque dans le nature il y a une proportion très basse de malware par rapport aux autres applications, malgré leur grand nombre. Nous avons donc téléchargé environ 27000 applications depuis le Google Play Store [3], en prenant le top 500 de chaque catégorie pendant l'été 2018. Nous avons pour cela exploité les outils open-source suivants : gplaycli [4] et google-play-crawler [5]. Les seules garanties que ces applications ne sont effectivement pas des malwares sont le Bouncer de Google cité plus haut, dont le taux de détection nous est inconnu, et l'avis favorable du public concernant ces applications (pour qu'elles aient atteint le top 500 d'une catégorie parmi quelques millions d'applications), qui est toujours susceptible de se tromper, malgré son nombre important. Nous n'avons à ce jour pas fait de double vérification par des antivirus concurrents, comme la base VirusTotal.

Pour effectuer la construction du modèle, nous utilisons le logiciel Weka. En effet ce dernier est écrit en Java et dispose en plus d'une interface graphique et ligne de commande d'une API pour l'utiliser depuis un programme Java. Il est donc possible de retirer le code de l'interface graphique du programme et inclure l'archive Java ainsi générée dans un projet Android pour effectuer des prédictions à partir du modèle. Le modèle est néanmoins construit au préalable sur le serveur à ce jour. Pour cela, nous avons écrit plusieurs scripts Bash permettant d'extraire les permissions de nos 32000 applications, les comparer aux permissions systèmes pour avoir le vecteur de booléen, et y ajouter notre connaissance de leur classe (malware ou légitime). Tout cela est écrit dans un fichier de type ARFF, qui correspond à un fichier de donnée exploitable par Weka. Le processus est schématisé en figure 3.2.

Dans un second temps, pour garantir une meilleure précision de nos résultats, nous bâtissons 2 nouveaux ensembles à partir des échantillons précédemment récupérés. Ces choix seront expliqués dans la partie suivante d'optimisation du classificateur. Tout d'abord, avant de construire les nouveaux ensembles nous vérifions que chaque application est effectivement un package apk ou jar valide dont on peut déterminer les permissions via le Manifest.xml. Cela enlève très peu d'applications du Play Store mais quelques-unes de Drebin et beaucoup de contagio et theZoo. Nous construisons alors les deux ensembles. Le premier est un ensemble de test constitué des malwares de contagio, plus récents, qui forment un total de 251 malwares valides, et autant d'applications légitimes choisies au hasard parmi les 27965 téléchargées précédemment. Le second est l'ensemble d'entraînement, composé des 5304 malwares de Drebin et tout autant d'applications bénignes du Play Store choisies au hasard parmi les 27714 restantes.

3. https://play.google.com/store
4. https://github.com/matlink/gplaycli/
5. https://github.com/Akdeniz/google-play-crawler/

3.7 Construction des ensembles pour Weka

Dans cette section, nous détaillerons plus en détail les outils, les scripts et la méthodologie utilisée pour collecter les applications mobiles et fabriquer nos ensembles de données utilisés par la suite pour entraîner et tester nos algorithmes d'apprentissage automatique.

3.7.1 Collection d'applications

Comme expliqué auparavant, les applications utilisées dans ce projet proviennent de trois sources distinctes :

— Drebin [6]
— contagio [7]
— Google Play Store [8]

Chacune de ces sources a nécessité différentes procédures pour le téléchargement efficace et contrôlé des applications requises.

Pour la base offerte par les chercheurs de Drebin, nous avons simplement téléchargé des archives à partir de liens fournis gracieusement, et désarchivé les applications pour l'étape suivante.

Pour les malwares partagés sur contagio, nous avons obtenu le mot de passe des archives de la part de la personne maintenant le dépôt et téléchargé l'ensemble des liens disponibles sur la page avec l'outil *wget*.

Pour les applications disponibles sur le Play Store, nous avons dû faire preuve de plus de créativité. Puisqu'il a fallu se limiter à un certain nombre d'échantillons en première approche, et pour garantir un certain degré de légitimité des applications, nous avons considéré uniquement les 500 applications les plus populaires de chacune des catégories existantes sur le site.

Grâce à l'outil google-play-scraper [9] basé sur NodeJS, nous avons pu récupérer les informations de toutes les applications suscitées. Le code permettant cette opération est donné en figure 3.4 et un échantillon du JSON obtenu pour une seule application est montré en figure 3.5. Nous traitons le résultat avec les outils *grep* et *awk* pour ne conserver que les noms exacts des paquets en supprimant les doublons. La commande effectuée est présentée en figure 3.6.

6. https://www.sec.cs.tu-bs.de/ danarp/drebin/index.html
7. http://contagiomobile.deependresearch.org/index.html
8. https://play.google.com/store
9. `https://github.com/Akdeniz/google-play-crawler/`

```
var gplay = require('google-play-scraper');
var i;
for(var key in gplay.category){
  for(i=0; i<6; i++){
  gplay.list({
        category: key,
        collection: gplay.collection.TOP_FREE,
        num: 100,
        start: i*100
      })
    .then(console.log, console.log);
  }
}
```

Figure 3.4 Utilisation de google-play-scraper

```
{ url: 'https://play.google.com/store/apps/details?id=com.
   turbochilli.hoppyfrog2',
    appId: 'com.turbochilli.hoppyfrog2',
    title: 'Hoppy Frog 2 - City Escape',
    summary: 'The original endless hopper sequel to \'Hoppy
       Frog\' is back.',
    developer: 'Turbo Chilli',
    developerId: 'Turbo+Chilli',
    icon: '//lh3.googleusercontent.com/n-Q69KJgt2ad3nOQ2
       JMZBQydU0HqC1y6uArW6AYZJ3yn_DR2IMSEzCLzVcnW6Aeicw=w340',
    score: 4.5,
    scoreText: ' Rated 4.5 stars out of five stars ',
    priceText: '',
    free: true },
```

Figure 3.5 Exemple des informations d'une application récupérées avec le script en figure 3.4

```
grep -Po "(?<=appId: ')(.+)(?=',)" | awk '!x\[$0]++'
```

Figure 3.6 Commande bash pour conserver les noms des paquets d'application uniquement

A partir du fichier listant les 26907 noms d'applications distincts, nous utilisons gplaycli [10]. Le programme utilise sa propre clé d'API pour Google et permet de télécharger n'importe quelle application à partir du nom du paquet. Le script mettant en oeuvre cet outil est donné en figure 3.7.

Une fois les applications téléchargées, nous avons procédé à une vérification supplémentaire de nos données afin de construire des ensembles sains. En effet, malgré la provenance des applications, nous avons dû vérifier que les fichiers présentés (surtout dans le cas des bases de malware) étaient bien des applications qui pouvaient être analysées et caractérisées par la méthode prévue. Pour cela, nous avons simplement utilisé l'outil d'extraction des permissions sur tous nos échantillons, et nous avons conservé uniquement les fichiers qui ne provoquaient pas d'erreur à son utilisation. L'outil s'appelle *aapt*, pour Android Asset Packaging Tool, et dispose d'une fonction appelée *dump permissions* qui prend un paquet en argument et liste les permissions données par son fichier Manifest.xml. Un script pour ce tri est donné en exemple en figure 3.8.

3.7.2 Création des ensembles

Puisque nous avons décidé que les caractéristiques identifiant les applications, en dehors de leur classe (à prédire) seront les permission systèmes Android présentes dans le Manifest, nous devons extraire ces dernières pour convertir nos dossiers remplis d'APK et autres JAR en fichier texte caractérisant la présence des permissions. La liste des permissions systèmes Android à ce jour, telle que donnée sur le site officiel [11] est redonnée dans la liste présentée en figure A.1.

Afin de créer les ARFF (fichier de donnée utilisé par Weka), nous devons décider d'un format et obtenir de nos paquets complets d'applications des instances pour Weka respectant ce format. Pour rappel, nous choisissons de représenter chacune des applications avec le vecteur

10. https://github.com/matlink/gplaycli/
11. https://developer.android.com/reference/android/Manifest.permission

```
1 #!/bin/bash
2 FILE=downloadList26907.txt
3 DIR=/home/arfoub/Projects/malware-databases/google-play-store-scrap-27k/
4
5 while read p; do
6     gplaycli -d $p -f $DIR
7 done <$FILE
```

Figure 3.7 Utilisation de gplaycli

```
#!/bin/bash

filedir=$1
faildir=$3
succdir=$2

for file in $filedir/*; do
  if aapt dump permissions "$file"; then
    mv $file $2
  else
    mv $file $3
  fi
done
```

Figure 3.8 Vérification des applications par aapt

suivant : 151 attributs numériques égaux à 0 ou 1, représentant respectivement la présence ou non de chacune des permissions systèmes dans le Manifest.xml de l'application, dans l'ordre donné en figures A.1 et A.2, suivi d'un attribut de classe égal à *malware* ou *benign* pour indiquer si le logiciel est malveillant. La première ligne du fichier ARFF (après les commentaires indiquant son origine) sera donc `@RELATION apk` pour indiquer le type de données, les 151 suivantes seront de la forme `@ATTRIBUTE ACCEPT_HANDOVER NUMERIC` en remplaçant `ACCEPT_HANDOVER` par chacune des permissions, puis on aura la ligne `@ATTRIBUTE class malware,benign` et enfin, après l'en-tête `@DATA` indiquant la fin des attributs et le début des données étiquetées, des lignes semblable à

```
1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,
0,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,0,1,1,1,1,1,benign
```

qui sont générées à l'aide du script présenté en figure 3.9.

A un certain point dans notre optimisation du chapitre 4, nous devons changer les ensembles de données pour tenir compte de la certitude de l'algorithme quant à ses prédictions. Nous remplaçons alors `@ATTRIBUTE class malware,benign` par `@ATTRIBUTE IS_MALWARE NUMERIC` et nous remplaçons la valeur `malware` par 100 et la valeur `benign` par -100 à chaque ligne suivant `@DATA` à l'aide d'un éditeur de texte, comme Notepad++ ou VIM.

```
1  #!/bin/bash
2  LEGITDIR=/home/arfoub/Projects/malware-databases/google-play-store-scrap-27k
3  MALWDIR=/home/arfoub/Projects/malware-databases/malware-samples
4  PERMFILE=android_system_permissions.txt
5  LEGITFILE=legitfile.txt
6  MALWFILE=malwfile.txt
7  ARFFFILE=all_samples.arff
8
9  makeline(){ #pass two args : first is filename and second is 'malware' or '
       benign'
10   aapt dump permissions "$FILE" | grep -Po "(?<=name='android.permission.)(\S
       +)(?=')" > $LEGITFILE
11   while read p; do
12     grep -Fxq $p $LEGITFILE #check if $p string exists in $LEGITFILE, returns
         0 or 1 to $?
13     echo -n $?,
14   done <$PERMFILE
15   echo $2
16 }
17
18 for FILE in $LEGITDIR/*
19 do
20   makeline "$FILE" benign >> $ARFFFILE
21 done
22
23 for DIR in $MALWDIR*
24 do
25   for FILE in $DIR/*
26   do
27     makeline "$FILE" malware >> $ARFFFILE
28   done
29 done
```

Figure 3.9 Création du fichier ARFF

CHAPITRE 4 OPTIMISATION ET RESULTATS

Une fois les données collectées et la méthode générale définie, il nous reste la partie pratique du travail. Il s'agit d'une part de construire et optimiser un classificateur effectif d'apprentissage automatique, et d'autre part de l'implémenter sous Android. Par la suite, nous effectuerons des mesures et des tests afin de déterminer le meilleur algorithme à choisir pour classer les applications. L'application Android subira alors plusieurs itérations, d'une part par l'amélioration de son coeur, la classe de détection, et d'autre part par l'ajout de fonctionnalités mettant en oeuvre cette dernière. Nous décrirons donc certaines parties du code de l'application actuelle. Enfin, l'architecture complète comprenant une méthode implémentée côté serveur et une communication client-serveur sera envisagée.

4.1 Optimisation de la classification

Comme dit précédemment, la méthode est assez simple et définie, mais laisse beaucoup de place à l'optimisation. D'une part, la collecte de données permet toujours d'obtenir un modèle plus fidèle. D'autre part, de nombreux algorithmes d'apprentissage automatique sont disponibles sur Weka, et peuvent être comparés. Ils peuvent aussi être optimisés à l'aide d'un choix précis de leurs paramètres, et du format des données en entrée.

4.1.1 Prototype initial

En premier lieu nous avons bâti le modèle avec le classificateur naïf de Bayes. Cela prend environ 50 secondes sur un ordinateur portable de moyenne gamme datant de 2016. Pour tester le modèle sans ensemble de test (nous ne fournissons ici qu'un ensemble d'entraînement), Weka propose la vérification croisée : le modèle est alors entraîné sur 90% des données et testé sur les 10% restant, et cela est fait 10 fois afin d'avoir testé le modèle sur toutes les données en l'ayant entraîné sur toutes les autres. Avec notre base actuelle et l'algorithme choisit, les performances du modèle en vérification croisée sont les suivantes :

- Précision (instances correctement classées) : 92.4486%
- Taux de faux positif (applications bénignes incorrectement classées) : 0.217
- Taux de faux négatif (malware non détectés) : 0.047

4.1.2 Choix des ensembles

C'est ainsi que nous avons pu développer le premier prototype, le premier modèle de prédiction étant nécessaire pour développer une application fonctionnelle. Cependant, afin d'optimiser la précision des prédictions et garantir ces résultats, nous avons dans un second temps expérimenté avec plusieurs algorithmes inclus par défaut dans Weka (tableau 4.1.3) et les nouveaux ensembles d'échantillons équilibrés. En effet, nous avons réalisé qu'il est généralement préférable en apprentissage automatique d'utiliser des ensembles équilibrés, c'est-à-dire que chaque classe est représentée par un nombre proche d'échantillons. Il existe des cas où il est cependant préférable d'utiliser des ensembles représentatifs, c'est-à-dire présentant la même proportion d'échantillons de chaque classe que dans la réalité. Ceci est vrai lorsqu'il est plus important de détecter la classe majoritaire que la classe minoritaire. Dans notre cas, il est plus important de détecter les malwares, qui sont la classe minoritaires, donc nous utiliserons les ensembles équilibrés. Ceci a été testé avec la validation croisée de Weka (tableau 4.1) : même si la précision globale est meilleure sur l'ensemble représentatif (il faut noter que la proportion n'est pas si représentative puisqu'il s'agit de 27/5), le nombre de malwares non détectés est très important par rapport à la validation croisée sur l'ensemble équilibré, sur lequel on a néanmoins une précision globale un peu plus basse.

Tableau 4.1 Performances globales en validation croisée pour Bayes naïf : ensemble représentatif contre équilibré

Ensemble	27k - 5k	5k - 5k
Correctly Classified Instances	31016	9483
Correctly Classified Instances (%)	93.23	89.39
Kappa statistic	0.743	0.788
Mean absolute error	0.075	0.114
Root mean squared error	0.240	0.288
Relative absolute error (%)	28.01	22.98
Root relative squared error (%)	65.60	57.51
Total Number of Instances	33269	10608

On peut voir sur les tableaux comparatifs 4.1, 4.2 et la figure 4.1 que même si la précision globale (table 4.1) semble meilleure avec l'ensemble dit représentatif comptant environ 27000 applications bénignes pour 5000 malwares, les résultats détaillés racontent une autre histoire. Les matrices de confusion de la figure 4.1 montrent bien que le classificateur entraîné sur l'ensemble équilibré est plus à même de détecter les malwares même si, en proportion, plus d'applications bénignes sont mal classées. Cependant la détection des applications bénignes constitue notre second critère après les malwares, et la différence n'est pas assez importante

Tableau 4.2 Performances détaillées en validation croisée pour Bayes naïf : ensemble représentatif contre équilibré

Ensemble	27k - 5k			5k - 5k		
Class	malware	benign	average	malware	benign	average
TP Rate	0.768	0.964	0.932	0.85	0.938	0.894
FP Rate	0.036	0.232	0.201	0.062	0.15	0.106
Precision	0.8	0.956	0.931	0.932	0.862	0.897
Recall	0.768	0.964	0.932	0.85	0.938	0.894
F-Measure	0.783	0.96	0.932	0.889	0.898	0.894
MCC	0.743	0.743	0.743	0.791	0.791	0.791
ROC Area	0.964	0.964	0.964	0.964	0.964	0.964
PRC Area	0.836	0.993	0.968	0.96	0.967	0.963

a	b	<= classified as
4071	1233	a = malware
1020	26945	b = benign

(a) Représentatif

a	b	<= classified as
4510	794	a = malware
331	4973	b = benign

(b) Equilibré

Figure 4.1 Matrices de confusion en validation croisée pour Bayes naïf : ensemble représentatif contre équilibré

pour justifier qu'on prenne ici ce critère en compte devant le nombre de malwares correctement classés. Nous utiliserons donc pour la suite des ensembles équilibrés uniquement.

4.1.3 Choix d'un classificateur

Après avoir défini les nouveaux ensembles, nous avons creusé plus avant les possibilités de Weka. Il s'agit ici d'étudier les performances d'autres classificateurs, plus variés, parfois plus complexes et potentiellement plus précis. Avec les algorithmes plus complexes, on peut maintenant connaître des problèmes de performances si les calculs sont trop coûteux.

Le premier tableau présenté en 4.1.3 décrit les performances générales (précision et taux de faux positifs) de chacun des algorithmes inclus par défaut sur Weka et utilisable sur nos ensembles avec classes. C'est-à-dire que chaque instance est soit bénigne, soit un malware, alors qu'on pourrait aussi définir un score numérique montrant la certitude de l'algorithme quand à une telle classification de l'instance, comme nous le ferons par la suite. Il est à noter que tous les algorithmes présentés dans ce tableau sont utilisés avec la configuration par défaut fournie par Weka 3.8.2. Il est évident que nous n'avons pas pris le temps de jouer avec tous les hyperparamètres de chacun des classificateurs, étant donné leur nombre à ce

stade.

— La précision globale correspond au pourcentage d'applications correctement classées par rapport à l'ensemble des applications testées.
— Le taux de faux positif des malwares correspond au nombre d'applications bénignes classées comme malwares divisé par le nombre d'applications bénignes au total.
— Le taux de faux positif des applications bénignes est l'équivalent pour les malwares. Il est donc représentatif du nombre de malwares qui ne seront pas détectés.

Tableau 4.3 – Comparaison des classificateurs de Weka

Nom	Entraînement			Test		
	Précision globale	Taux FP Malware	Taux FP Bénin	Précision globale	Taux FP Malware	Taux FP Bénin
NaiveBayes	89.39%	0.062	0.15	90.64%	0.036	0.151
NaiveBayes Multinomial	89.50%	0.062	0.15	91.83%	0.036	0.151
NaiveBayes Updateable	89.39%	0.08	0.13	90.64%	0.072	0.092
NaiveBayes Updateable	89.39%	0.062	0.15	90.64%	0.036	0.151
Logistic	94.57%	0.066	0.042	93.82%	0.064	0.06
Multilayer Perceptron	93.66%	0.075	0.051	89.64%	0.016	0.191
Simple Logistic	94.65%	0.064	0.043	94.42%	0.06	0.052
SMO	94.61%	0.068	0.04	94.22%	0.064	0.052
Voted Perceptron	93.67%	0.057	0.069	91.04%	0.044	0.135
SGD	93.85%	0.041	0.082	94.62%	0.06	0.048
Random Forest	96.00%	0.031	0.049	91.63%	0.024	0.143
Random Tree	94.59%	0.053	0.056	91.24%	0.036	0.139
J48	94.82%	0.039	0.065	91.24%	0.028	0.147
BayesNet	90.60%	0.025	0.163	89.24%	0.008	0.207
NaiveBayes Multinomial Updateable	89.50%	0.08	0.13	91.83%	0.072	0.092
IBk	95.53%	0.041	0.048	90.64%	0.048	0.139
KStar	95.38%	0.025	0.067	90.64%	0.032	0.155
LWL	81.68%	0.259	0.107	73.71%	0.255	0.271

Tableau 4.3 – Comparaison des classificateurs de Weka

Nom	Entraînement			Test		
	Précision globale	Taux FP Malware	Taux FP Bénin	Précision globale	Taux FP Malware	Taux FP Bénin
Decision Stump	81.68%	0.259	0.107	73.71%	0.255	0.271
LMT	95.35%	0.041	0.052	92.03%	0.02	0.139
REPTree	94.55%	0.045	0.064	88.65%	0.052	0.175
Decision Table	92.73%	0.111	0.034	93.82%	0.1	0.024
Jrip	94.20%	0.043	0.073	89.44%	0.044	0.167
OneR	81.68%	0.259	0.107	73.71%	0.255	0.271
PART	95.18%	0.041	0.055	90.64%	0.155	0.094
Attribute Selected Classifier	91.90%	0.139	0.023	92.83%	0.124	0.02
Bagging	94.91%	0.047	0.055	91.43%	0.028	0.143
Classification Via Regression	94.39%	0.058	0.054	89.04%	0.044	0.175
Filtered Classifier	94.74%	0.05	0.055	91.83%	0.02	0.143
Iterative Classifier Optimizer	93.63%	0.056	0.071	88.25%	0.064	0.171
LogitBoost	93.63%	0.056	0.071	88.25%	0.064	0.171
Multiclass Classifier	94.57%	0.066	0.042	93.82%	0.064	0.06
Multiclass Classifier Updateable	93.85%	0.041	0.082	94.62%	0.06	0.048
Randomizable Filtered Classifier	93.39%	0.065	0.067	88.84%	0.064	0.159
Random Subspace	94.55%	0.038	0.071	91.43%	0.028	0.143
AdaBoostM1	91.50%	0.066	0.104	92.63%	0.076	0.072
Random Committee	95.80%	0.035	0.049	91.43%	0.028	0.143

Le tableau 4.1.3 met en valeur la supériorité de quelques-uns des algorithmes que nous avons ensuite étudié en plus détail. Il s'agit des classificateurs suivants (avec leur configuration exacte) :

— Simple Logistic (`weka.classifiers.functions.SimpleLogistic -I 0 -M 500 -H`

`50 -W 0.0`)

— SMO : `weka.classifiers.functions.SMO -C 1.0 -L 0.001 -P 1.0E-12 -N 0 -V -1 -W 1 -K ""weka.classifiers.functions.supportVector.PolyKernel -E 1.0 -C 250007"" -calibrator ""weka.classifiers.functions.Logistic -R 1.0E-8 -M -1 -num-decimal-places 4""`

— SGD : `weka.classifiers.functions.SGD -F 0 -L 0.01 -R 1.0E-4 -E 500 -C 0.001 -S 1`

— Multi Class Classifier Updateable : `weka.classifiers.meta.MultiClassClassifierUpdateable -M 0 -R 2.0 -S 1 -W weka.classifiers.functions.SGD - -F 0 -L 0.01 -R 1.0E-4 -E 500 -C 0.001 -S 1`

Nous avons alors dressé les matrices de confusion (figure 4.2) de chacun de ces classificateurs et déterminé que SGD et MultiClassClassifierUpdateable (MCCU) étaient les plus précis. Il est à noter, comme le montre la configuration de chacun des algorithmes, que ce dernier est basé sur SGD (qui est lui aussi parmi les meilleurs), ce qui explique que leurs performances soient identiques. Dans les deux cas les classificateurs sont *updateable*, c'est-à-dire qu'on peut leur ajouter des données après la construction du modèle sans avoir à tout reconstruire, puisque le MCCU doit être basé sur un classificateur qui peut être mis à jour, en l'occurence ici le SGD. L'unique différence est que le second est MultiClass, c'est-à-dire qu'il permet de traiter des datasets avec plus de deux classes. Ce n'est actuellement pas le cas donc il n'y a effectivement aucune différence entre les deux classificateurs. Cependant, on peut imaginer des cas de figure où on aurait 3 classes malwares, benign et unsure, par exemple. Nous verrons par la suite que plutôt qu'augmenter le nombre de classe, on peut utiliser un attribut numérique pour gérer la certitude des prédictions.

a	b	<= classified as
238	13	a = malware
15	236	b = benign

(a) Simple Logistic

a	b	<= classified as
238	13	a = malware
16	235	b = benign

(b) SMO

a	b	<= classified as
239	12	a = malware
15	236	b = benign

(c) SGD

a	b	<= classified as
239	12	a = malware
15	236	b = benign

(d) Multi Class Classifier Updateable

Figure 4.2 Matrices de confusion des meilleurs classificateurs sur l'ensemble de test

4.1.4 Seconde approche : méthodes de régression

Comme indiqué précédemment, un des inconvénients de notre approche actuelle est de ne pas gérer les cas incertains ou permettre d'introduire un jugement de l'utilisateur. Ceci constitue donc une des motivations pour la seconde approche avec méthodes de régression. L'autre motivation est tout simplement d'essayer les autres algorithmes disponibles sur Weka, qui sont des algorithmes de régression et donc non compatibles avec notre dataset à classes, puisqu'ils requièrent un attribut numérique. Nous avons donc dupliqué les datasets d'entraînement et de tests précédemment utilisés et avons remplacé l'attribut `CLASS` par un attribut numérique `IS_MALWARE`, situé entre -100 (pour les applications bénignes) et 100 (pour les malwares). Les algorithmes suivants, une fois entraînés, effectuent leur prédiction en attribuant un score `IS_MALWARE` aux échantillons testés, et c'est à nous de faire un second traitement pour décider quel score est suffisant pour garantir notre prédiction.

Le tableau 4.4 montre les performances des algorithmes testés. Contrairement aux classificateurs, on n'a pas de nombre précis de prédictions correctes ou de matrice de confusion puisque les prédictions peuvent très bien donner -13, 0 ou 56 et non 100 ou -100 comme les échantillons étiquetés. Il faut donc se référer au coefficient de corrélation, situé entre 0 et 100%, qui indique si les prédictions sont proches de la réalité, et les différentes valeurs d'erreurs entre réalité et prédiction. Le temps est encore une fois à prendre en compte pour l'usage sur téléphone de certains algorithmes.

Tableau 4.4 – Comparaison des algorithmes de régression avec dataset à classe numérique

Algorithme	Ensemble	Correlation Coefficient	Mean absolute error	Root mean squared error	Relative absolute error (%)	Root relative squared error (%)	Time (s)
Linear Regression	Training	0.873	36.578	48.772	36.578	48.772	6.73
	Testing	0.850	37.729	52.902	37.729	52.902	0.9
Multilayer Perceptron	Training	0.955	11.0169	29.513	11.016	29.513	1259.67

Tableau 4.4 – Comparaison des algorithmes de régression avec dataset à classe numérique

Algorithme	Ensemble	Correlation Coefficient	Mean absolute error	Root mean squared error	Relative absolute error (%)	Root relative squared error (%)	Time (s)
	Testing	0.915	16.708	40.669	16.708	40.669	0.13
Simple Linear Regression	Training	0.641	58.894	76.742	58.894	76.742	0.05
	Testing	0.474	69.245	90.415	69.245	90.415	0.01
SMO Reg	Training	0.781	33.457	65.492	33.457	65.492	6058.1
	Testing	0.682	43.688	79.356	43.688	79.356	0.01
IBK	Training	0.968	6.218	24.937	6.218	24.937	43.04
	Testing	0.906	14.784	42.401	14.784	42.401	2.13
Kstar	Training	0.955	11.199	29.727	11.199	29.727	2941.16
	Testing	0.898	19.106	44.188	19.106	44.188	156.53
LWL	Training	0.654	57.44	75.617	57.448	75.617	1821.66
	Testing	0.492	67.694	89.258	67.694	89.258	197.43
Decision Stump	Training	0.6411	58.894	76.742	58.894	76.742	0.23
	Testing	0.474	69.245	90.415	69.245	90.415	0.01
M5P	Training	0.929	15.621	36.924	15.621	36.924	2.61

Tableau 4.4 – Comparaison des algorithmes de régression avec dataset à classe numérique

Algorithme	Ensemble	Correlation Coefficient	Mean absolute error	Root mean squared error	Relative absolute error (%)	Root relative squared error (%)	Time (s)
	Testing	0.892	21.566	45.623	21.566	45.623	0.01
Random Forest	Training	0.9657	8.4154	26.040	8.415	26.040	7.13
	Testing	0.918	16.399	39.904	16.399	39.904	0.02
Random Tree	Training	0.968	6.218	24.937	6.218	24.937	0.15
	Testing	0.909	13.822	42.119	13.822	42.119	0.01
REP Tree	Training	0.931	13.209	36.344	13.209	36.344	0.99
	Testing	0.888	19.533	46.420	19.533	46.420	0.01
Bagging	Training	0.941	12.911	33.7637	12.911	33.7637	6.12
	Testing	0.904	18.201	43.112	18.200	43.112	
Additive Regression	Training	0.833	43.294	55.422	43.294	55.422	1.84
	Testing	0.812	47.653	59.289	47.653	59.289	0.01
Random Committee	Training	0.968	6.218	24.937	6.218	24.937	1.4
	Testing	0.917	15.312	39.984	15.312	39.984	0.01
Randomizable Filtered Classifier	Training	0.967	6.293	25.086	6.293	25.086	5.54

Tableau 4.4 – Comparaison des algorithmes de régression avec dataset à classe numérique

Algorithme	Ensemble	Correlation Coefficient	Mean absolute error	Root mean squared error	Relative absolute error (%)	Root relative squared error (%)	Time (s)
	Testing	0.864	18.144	51.010	18.144	51.010	0.289
Random SubSpace	Training	0.919	29.376	42.098	29.3768	42.0983	3.65
	Testing	0.872	33.517	50.466	33.517	50.466	0.02
Regression By Discretization	Training	0.941	11.281	33.588	11.281	33.588	16.36
	Testing	0.905	16.188	42.843	16.188	42.843	0.82
Decision Table	Training	0.932	13.086	36.174	13.086	36.174	33.71
	Testing	0.893	18.626	45.202	18.626	45.202	0.01
M5Rules	Training	0.926	15.921	37.740	15.921	37.741	18.18
	Testing	0.871	22.938	49.590	22.938	49.590	0.01

Afin de pouvoir comparer ces résultats aux méthodes de classification, nous avons étudié plus avant les prédictions des algorithmes ayant les meilleurs coefficients de corrélation sur l'ensemble de test : Random Forest, ainsi que Random Committee et Random Tree. On se passe d'étudier le Multilayer Perceptron pour l'instant étant donné qu'il demande beaucoup de ressources pour être calculé et n'a pas pour autant de meilleures performances que Random Forest, mais il faut garder en tête que son étude serait intéressante, avec la possibilité de faire varier les hyperparamètres pour customiser le réseau de neurones.

La méthode la plus simple et naïve consiste à convertir le `IS_MALWARE` prédit en un binaire : si la prédiction numérique est strictement inférieure à 0, on considère que la prédiction est

`BENIGN`, et si la prédiction est supérieure ou égale à 0, on considère qu'elle est `MALWARE`. Comme on pourrait s'y attendre, les résultats ne sont pas idéaux avec ce seuil de 0 : on obtient 41 erreurs de prédiction pour Random Forest, 42 pour Random Committee et 44 pour Random Tree. C'est notamment encore supérieur aux 27 (12 + 15) erreurs pour 502 échantillons qu'on avait en figure 4.2 pour les classificateurs. Nous testons alors différents seuils situés entre -90 et 90 par tranche de 10. Les résultats sont bien plus prometteurs sur certains seuils, les plus petits nombres d'erreurs dans ce cas étant 19 pour Random Tree avec un seuil de -20 ou -30, 18 pour Random Committee avec un seuil de -20 et 17 pour Random Forest avec un seuil de -30. Dans l'optique d'un modèle plus complexe qui étudie les incertitudes sur les prédictions, on pourrait définir non pas un mais deux seuils pour l'algorithme. L'idée serait alors d'avoir une marge située entre les deux seuils telle que l'on considère comme incertaine toute prédiction dont la valeur obtenue se situe dans la marge. Une piste de travail pour raffiner le prototype serait la recherche d'une marge optimale pour les incertitudes, et l'indication à l'utilisateur que notre prédiction n'est pas assez précise pour une application tombant dans cette marge, mais on se cantonnera pour l'instant à l'usage d'un simple seuil qui donne de très bons résultats. En effet, dans notre prototype actuel, l'ajout d'une troisième catégorie n'a pas encore d'usages intéressants.

Par la suite, nous détaillerons comment nous avons intégré ces différents modèles de plus en plus précis et complexes dans une application mobile pour l'utilisateur final.

4.2 Prototype d'application de démonstration

Afin de démontrer un usage pratique de ce modèle pour la détection des malwares sur l'appareil d'un utilisateur lambda, nous avons développé une application Android de test intégrant la méthode et montrant par là-même qu'elle s'installe sans modifications de l'appareil. Nous présenterons dans cette section une vue d'ensemble de ses différentes versions suivi d'une explication détaillée du code et de la structure de l'application actuelle.

4.2.1 Vue d'ensemble

La première version était une preuve de concept pour le développement Android et la communication client serveur (figure 3.1) : la «détection» était simplement un compte-rendu du nombre d'antivirus sur VirusTotal qui détectaient le malware placé dans le dossier Documents comme effectivement un malware. Pour obtenir cela nous avons intégré une classe permettant d'avoir le haché MD5 du fichier, et une classe pour faire une requête HTTP à l'API de VirusTotal et renvoyer la réponse à l'activité. Une autre fonction que nous avons

ajouté est l'envoi du haché à notre propre serveur, qui s'occupera lui-même de communiquer avec VirusTotal et de répondre à l'application Android. Ceci a été fait dans le but de montrer le bon fonctionnement de la communication client-serveur. Bien évidemment, cette méthode n'apporte pas une précision très grande et repose entièrement sur un service tiers. C'est pourquoi nous avons développé le modèle expliqué plus haut pour l'intégrer à l'application.

Dans la seconde version de l'application, l'affichage et la gestion des activités a été améliorée pour proposer deux modes de sélection de l'application (au lieu du chemin de fichier hardcodé en premier lieu). L'activité principale (figure 4.3) permet de choisir entre scanner un fichier disponible dans l'arborescence du téléphone, ou scanner une application déjà installée.

La première activité (figure 4.5) permet donc d'afficher l'arborescence (figure 4.6) ou d'écrire le chemin et presser Scanner (figure 4.7).

La seconde (figure 4.8) montre une liste des applications installées et permet d'en cocher une ou plusieurs avant de presser Scanner (figure 4.9) ou Tout Scanner (figures 4.10 et 4.11). Le résultat des prédictions manuelles sont affichés dans une boîte de dialogue.

Une autre fonctionnalité de l'application est le scan automatique des applications nouvellement installées. Le programme détecte toute nouvelle installation grâce aux BroadcastReceiver d'Android et scan le package installé. La prédiction apparaît sous forme de notification après quelques secondes (figure 4.4).

Dans tous les cas, si la sélection est valide le PackageManager nous permet de récupérer les permissions du Manifest de l'application, les comparer aux permissions systèmes, construire un ARFF personnalisé et effectuer la prédiction de Weka avec le modèle existant, chargé sur le téléphone. Ce processus est décrit dans le schéma en figure 3.3.

4.2.2 Classe de prédiction

En premier lieu, il est important de préciser que dans les morceaux de code présentés en annexe, nous avons retiré des commentaires, des fonctions redondantes, ainsi que les entêtes du fichier (commentaires, `package` et `import` multiples). Il faut cependant ajouter que la copie du code sur Android Studio pousserait ce dernier à proposer de les ajouter. Nous ajouterons à ce préambule que les fichiers `xml`, qu'ils s'agissent du `Manifest.xml` ou des *layout* (les dispositions des écrans et objets affichés), ne sont pas donnés non plus. En effet le `Manifest.xml` sera presque généré automatiquement par la résolution automatique des erreurs d'Android Studio, et les *layout* sont soit très simples, soit disponibles via les sources citées.

Notre classe de prédiction, Prediction.java, dont le code réduit est donné en figure B.1, fait

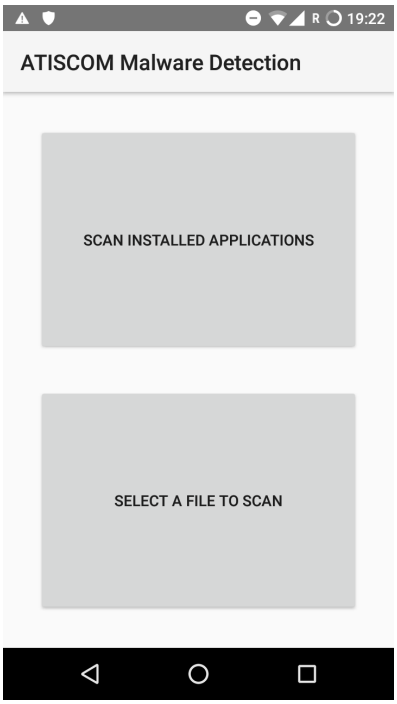


Figure 4.3 Activité principale

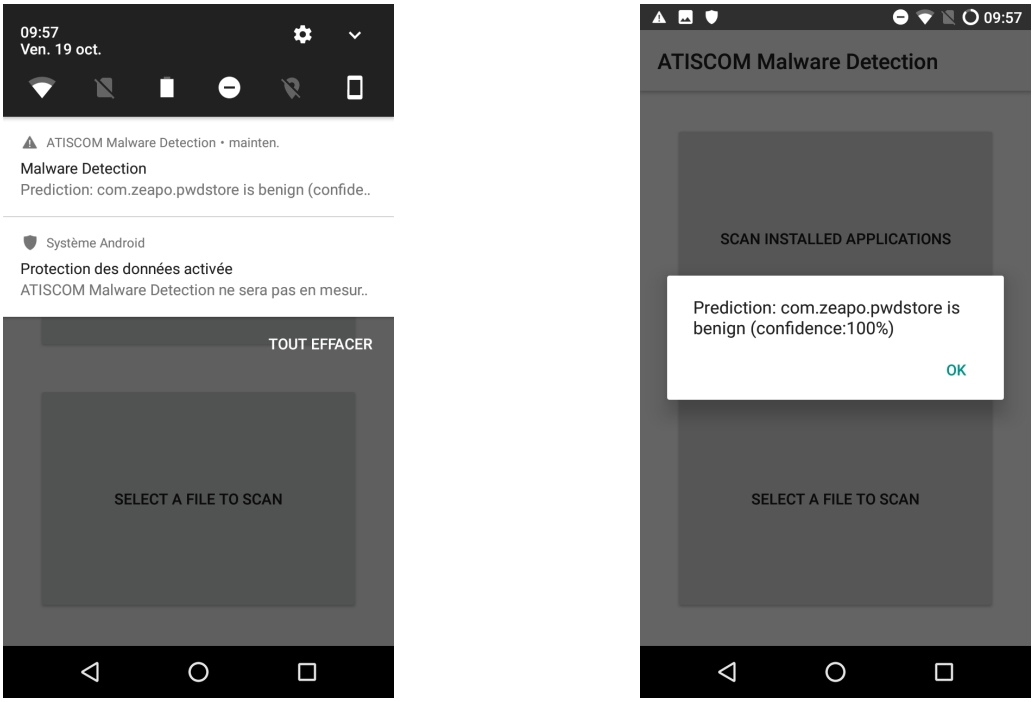


Figure 4.4 Prédiction sur une application nouvellement installée

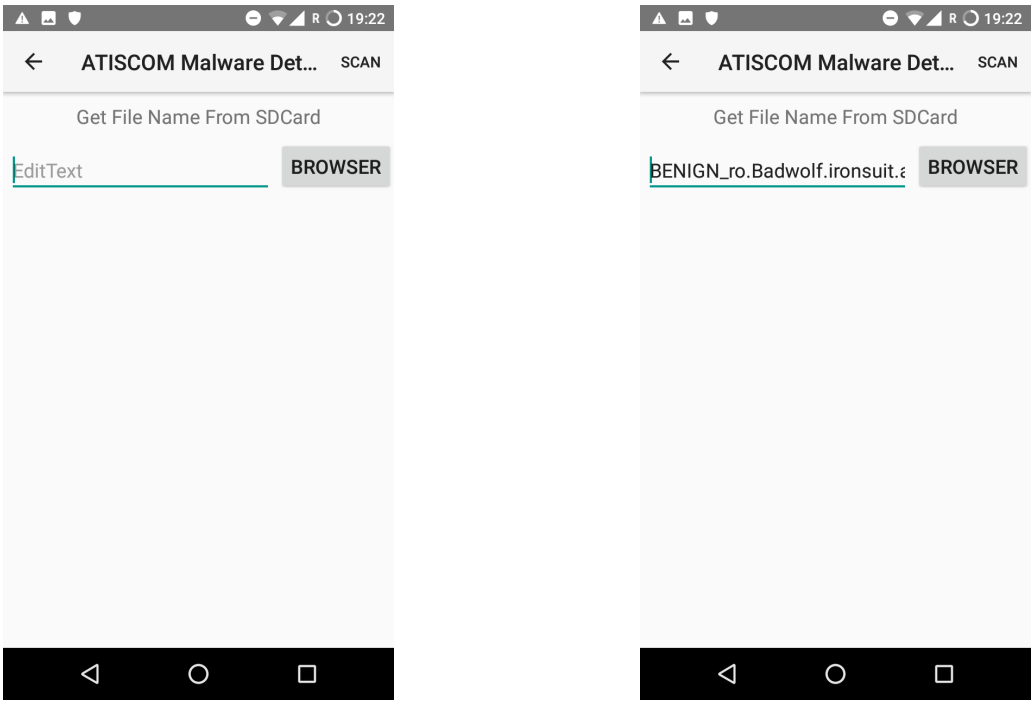


Figure 4.5 Activité de scan des fichiers locaux

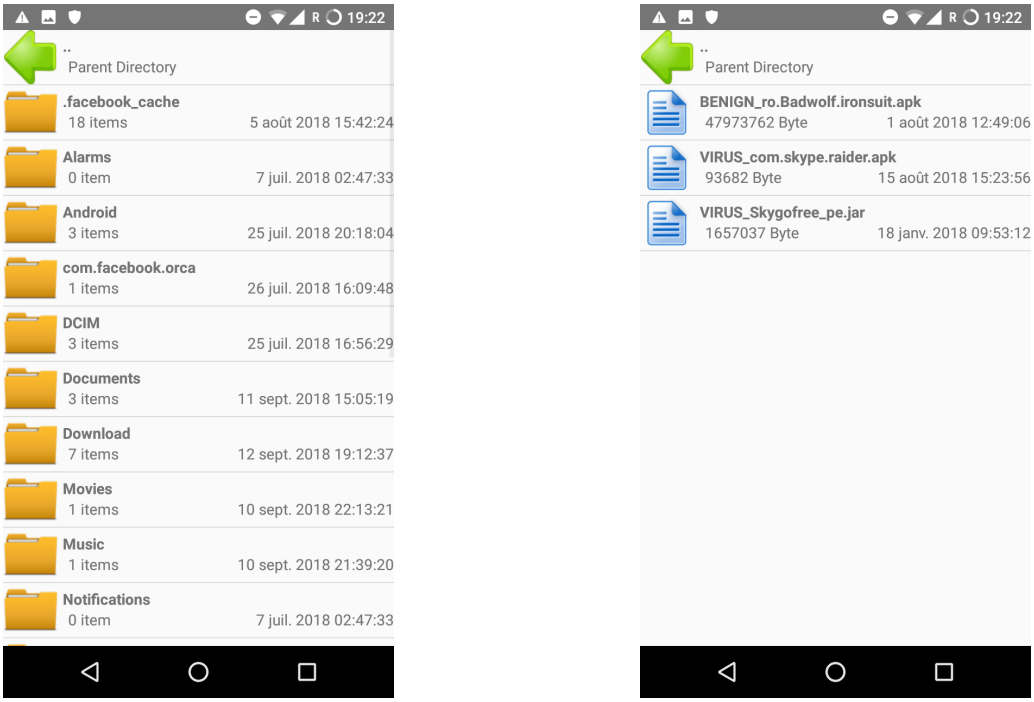


Figure 4.6 Explorateur de fichiers

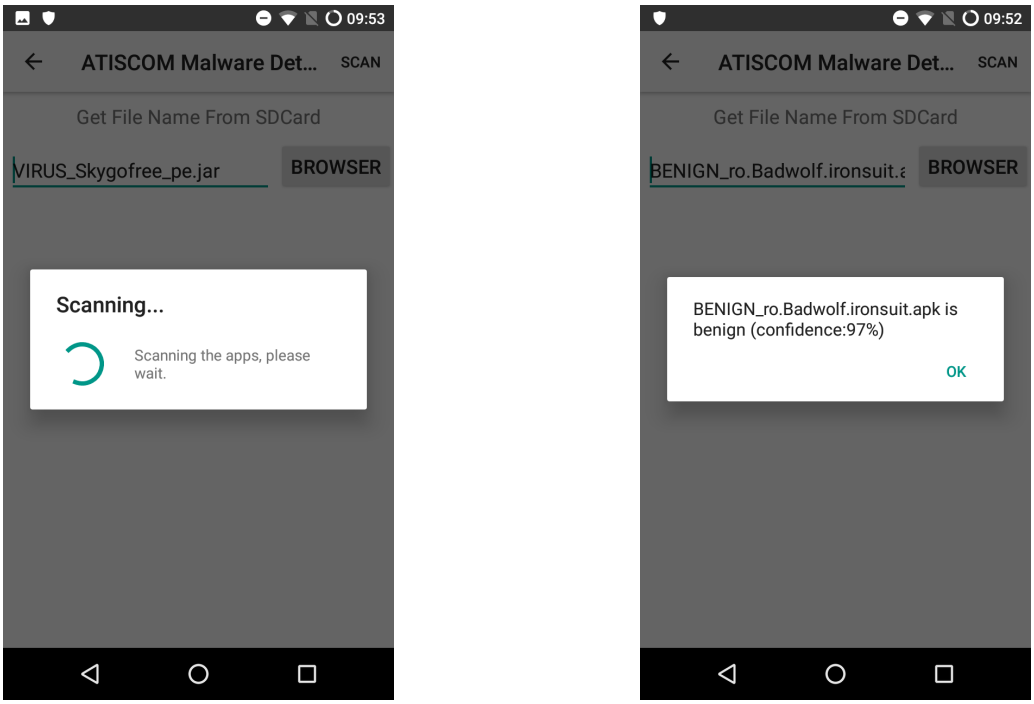


Figure 4.7 Résultats de scans de fichiers locaux

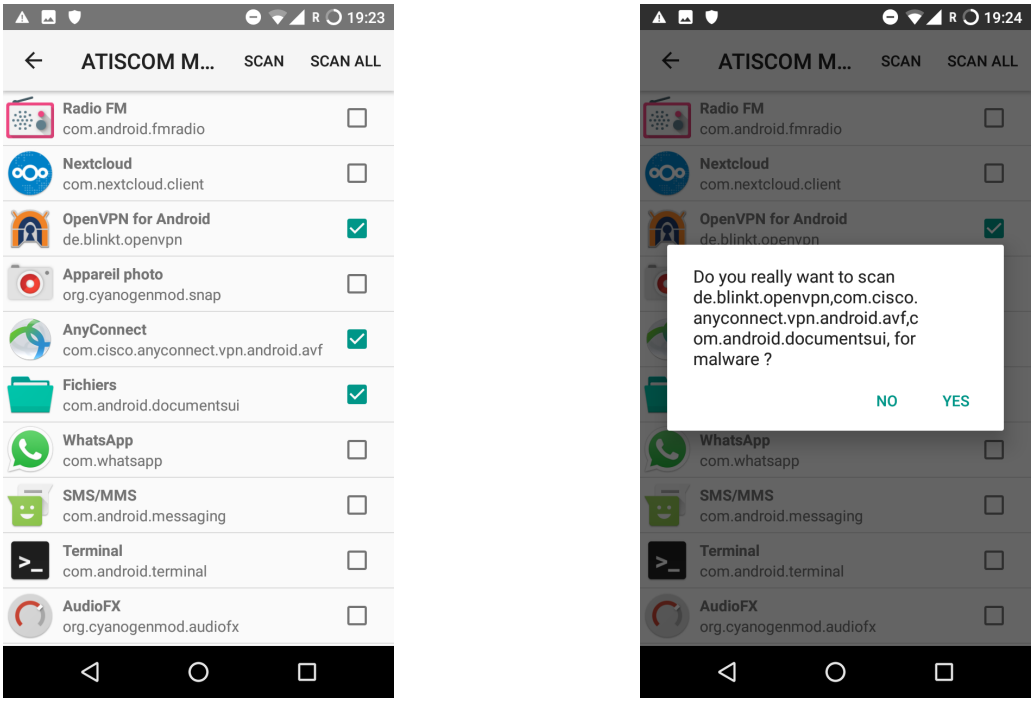


Figure 4.8 Sélection et confirmation de scan multiple des applications installées

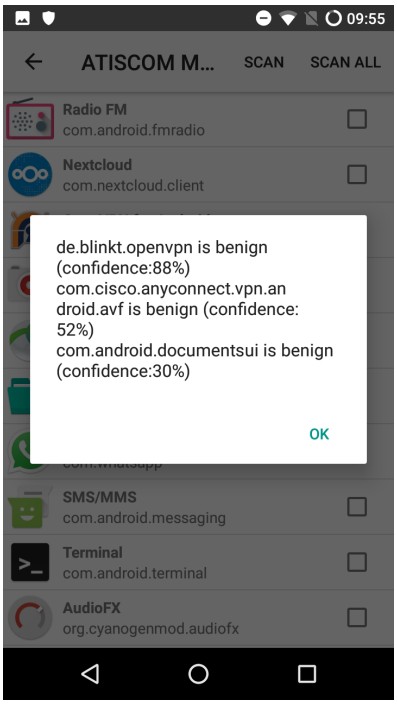


Figure 4.9 Résultats de scan multiple

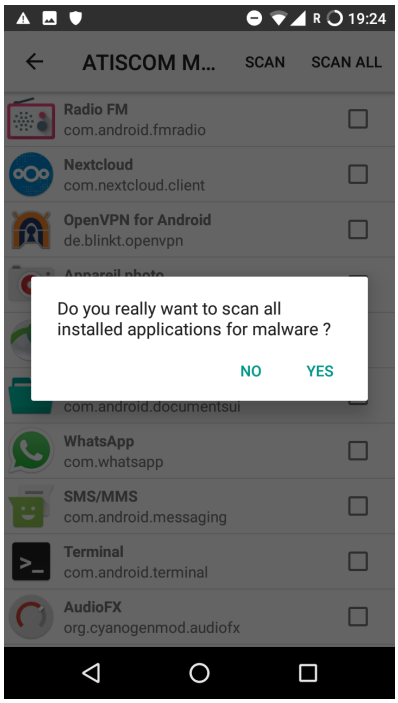
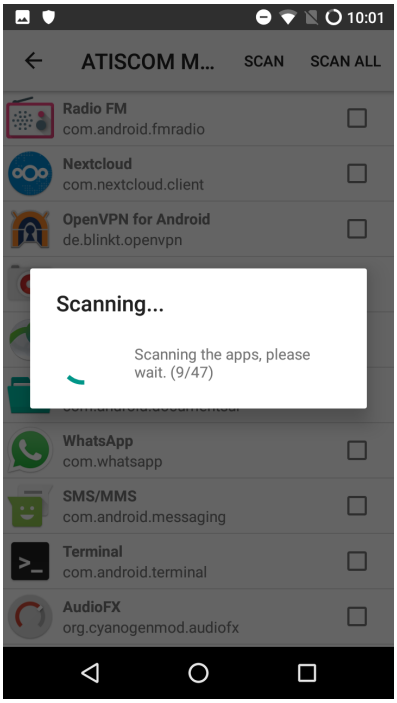


Figure 4.10 Confirmation de scan de toutes les applications installées



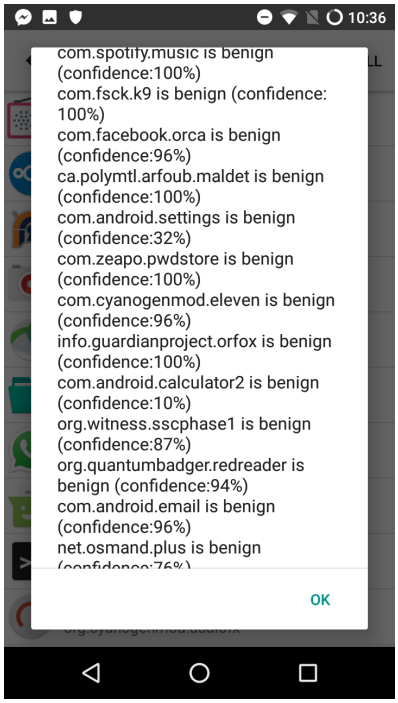


Figure 4.11 Résultats du scan de toutes les applications

le lien entre l'API Java de Weka, détaillée sur `https://waikato.github.io/weka-wiki/` et les activités et services de notre application.

Pour inclure Weka dans l'application, nous avons téléchargé une archive Java `weka-stable-3.8.1-SNAPSHOT.jar` du programme Weka sans les composantes d'interface graphique incompatibles avec Android, sur `https://github.com/MeiyappanKannappa/Weka-Android`. Bien que cette version ne soit pas la dernière de Weka en date (qui sont 3.8.3 en stable et 3.9.3 en développement), il s'agit d'une des versions stables, ce qui nous a épargné le travail de modification de la source pour l'inclure au projet. L'archive Java est déclarée comme une dépendance dans notre projet Android.

Les différentes itérations de notre projet ont fait que cette classe contient plusieurs fonctions redondantes ou inutiles, notamment les cas de test et les fonctions basées sur l'ancien classificateur (et non l'algorithme de régression), que nous ne montrons pas dans ce document. De plus, en raison de la construction initiale du programme, la majorité des fonctions peuvent prendre en argument, en plus du contexte de l'activité, une donnée du type File (fichier) ou ApplicationInfo (extrait d'un paquet Android). Nous ne montrons ici que ces dernières, mais il faut noter que dans le cas où le fichier considéré est un paquet d'application (seul cas de figure valide), il suffit d'utiliser la classes PackageManager pour récupérer l'instance de PackageInfo associée au fichier, avec le code présenté en figure 4.12.

en considérant que `context` est le contexte de l'activité, que nous récupérons pour la majorité des fonctions, `fileFullPath` est le chemin du fichier indiqué, et `PackageManager.GET_PERMISSIONS` permet d'obtenir les permissions dans le PackageInfo. Ceci peut être passé en argument des fonctions présentées en figure B.1 avec de légères adaptations.

La majorité des fonctions de cette classe utilitaire seront statiques, puisqu'elles sont amenées à être appelées directement depuis les activités sans avoir à instancier `Prediction`. Celle qui sera appelée en général est `NumApkPred`, qui retourne un `String` indiquant le résultat de la prédiction, en prenant comme argument une donnée de type `ApplicationInfo` et `Context`. Le contexte sera en général celui de l'activité qui appelle la fonction, ou simplement de l'application, puisqu'il est utile dans les autres méthodes pour appeler le `PackageManager`. Les

```
1 PackageManager pm = context.getPackageManager();
2 PackageInfo packageInfo = pm.getPackageArchiveInfo(fileFullPath,PackageManager
      .GET_PERMISSIONS);
```

Figure 4.12 Code Java pour obtenir ApplicationInfo depuis le fichier APK

informations de l'application seront utiles pour obtenir son nom et en déduire ses permissions via ledit gestionnaire de paquets.

Comme on peut le voir, la méthode charge la ressource (incluse dans l'application) brute correspondant au modèle pré-entraîné par Weka. Il s'agit d'un fichier binaire généré par l'explorateur de Weka à la suite de nos expérimentations, et que les classes importées depuis le JAR Weka peuvent désérialiser et exploiter. Par la suite, la méthode appelle `NumArffPred` qui permet d'effectuer la prédiction sur une donnée ARFF (comme les fichiers d'ensembles de données pour Weka) avec le modèle, après avoir généré ledit ARFF grâce à `MakeNumArff`, qui contiendra comme unique instance l'application donnée en argument.

La texte résultant de `NumApkPred` est déterminé par la méthode `IsMalware` qui détermine à partir de l'entier fourni par la méthode de prédiction (compris en conditions normales entre -100 et 100) la classe de l'application : `malware` pour les valeurs supérieures ou égales au seuil choisi et `benign` sinon, le cas d'erreur étant -9999. Il redonne aussi en pourcentage le rapport entre la différence de cette valeur et seuil et la différence optimum-seuil, comme *confidence*, c'est-à-dire notre indice de certitude pour la prédiction. Il sera donc de 100% si la prédiction retourne 100 ou -100, et 0 si elle retourne la valeur choisie pour seuil.

`MakeNumArff` a besoin de la méthode `GetPermissions` qui elle-même appelle la méthode `GetPermAll` pour obtenir le vecteur de chaînes de caractère correspondant aux permissions déclarées dans le manifeste de l'application analysée. Cela, comme dit précédemment, est simplement fait grâce à une instanciation du `PackageManager` et un appel à sa méthode `getPackageInfo`. `GetPermAll` permet de mettre en forme ce résultat par traitement du texte. Nous faisons ensuite appel à `ComparePerm` avec ces permissions et celles du système, déclarées précédemment, pour avoir un vecteur de 0 et 1 suivi d'une valeur arbitraire de `IS_MALWARE` comme expliqué plus haut. La classe `Attribute` de Weka permet de déclarer chacun des attributs sur Java, comme nous l'avions fait plus tôt dans le fichier .arff pour la construction des ensembles : d'abord les 151 permissions données avant, puis l'attribut numérique `IS_MALWARE`. L'`ArrayList` d'attributs ainsi formées permet de générer une instance de la classe `Instances` qui est l'équivalent du fichier ARFF. Nous lui ajoutons alors une ligne de données grâce au vecteur de 0 et 1 suivis de la valeur arbitraire.

`NumArffPred` prend ensuite cet objet `Instances` et instancie aussi un `Classifier` à partir du fichier de modèle. La méthode de Weka `classifyInstance` nous permet alors d'effectuer la prédiction sur la première ligne de donnée de notre «ARFF », c'est-à-dire celle caractérisant l'application analysée.

4.2.3 Activités

Le code de l'activité principale MainActivity est donnée en figure B.2. Il s'agit d'une simple fenêtre avec des boutons permettant d'aller vers les deux types de détection manuels actuellement implémentés : via un explorateur de fichiers locaux ou via la liste des applications installées.

Pour l'activité de détection des fichiers locaux, nous utilisons les classes FileBrowserActivity (figure B.3), FileChooser (figure B.4), Item (figure B.5) et FileArrayAdapter (figure B.6) inspirées de `https://custom-android-dn.blogspot.com/2013/01/create-simple-file-explore-in-android.html`.

Pour l'activité de détection des applications installées, nous utilisons les classes InstalledAppActivity (figure B.7) et ApplicationAdapter (figure B.8) inspirés de `https://stackoverflow.com/questions/19533098/how-to-list-all-installed-applications-with-icon-and-check-box-in-android`.

Avec l'inclusion de la méthode de régression pour prédire avec une certitude indiquée la classe des applications, nous avons aussi augmenté les temps de traitement. Pour le confort de l'utilisateur, nous avons donc implémenté un écran de chargement indiquant que l'opération est bel est bien en cours même pendant les traitement les plus longs. Le contraire pourrait en effet lui faire croire que l'application est bloquée et il la fermerait sans attendre le résultat. A cet effet, nous introduisons un morceau de code tiré d'une classe LoadingScreen tirée de `http://www.41post.com/4588/programming/android-coding-a-loading-screen-part-1` dans les activités qui effectuent des scans (figures B.3 et B.7). La prédiction est toujours lancée à l'intérieur d'une `AsyncTask`. La classe `Prediction` est appelée dans la méthode associée `doInBackground` pendant que s'affiche en premier plan, via `onProgressUpdate`, un cercle de chargement et le décompte du nombre d'applications scannées (dans le cas du scan multiple de InstalledAppActivity). Le message de dialogue (`AlertDialog` construit à l'aide de sa méthode statique `Builder`) montre les résultats de la prédiction en premier plan une fois que tout ceci est fini, grâce à `onPostExecute`.

4.2.4 Détection automatique des nouvelles applications installées

La détection automatique des nouvelles applications installées et l'affichage de la prédiction obtenue se fait grâce de la classe AppInstalledReceiver donnée en figure B.9.

Il est très important de noter que le fonctionnement de cette classe requiert plusieurs déclarations dans le `AndroidManifest.xml` de notre application. Il s'agit des lignes suivantes de code présentées en figure 4.13.

```
<receiver
        android:name=".AppInstalledReceiver"
        android:enabled="true"
        android:exported="true">
        <intent-filter android:priority="100">
                <action android:name="android.intent.action.PACKAGE_ADDED" />
                <data android:scheme="package" />
        </intent-filter>
</receiver>
```

Figure 4.13 Code à placer dans AndroidManifest.xml pour la détection automatique

Celles-ci permettent de déclarer que notre application reçoit les `Intent` en `Broadcast` de type `PACKAGE_ADDED`, qui sont envoyés à toutes les applications qui veulent les recevoir à chaque fois qu'une application est installée sur l'appareil (ceci inclus aussi les mises à jour et les remplacements). Cependant, il faut préciser dans le manifest ou le fichier de *build* de *Gradle* sur Android Studio que la version d'API Android de notre application est inférieure à 26 (voire 25), ce qui correspond aux versions 8.0 et 7.0 d'Android pour que cela fonctionne. En effet, comme indiqué sur `https://developer.android.com/about/versions/oreo/background#broadcasts`, leur utilisation a été ensuite limitée car ils causaient une consommation de ressources de la part de nombreuses applications à chaque broadcast. Il est donc à noter qu'une méthode tout aussi efficace se passant de tels broadcast pour surveiller les nouvelles installations serait appréciable pour que le projet soit plus facile à maintenir dans le futur.

Lorsqu'une nouvelle installation est détectée, la méthode `onReceive` s'exécute et passe dans le `try` car l'`Intent` reçu est de la forme attendue dans ce cas. Elle appelle alors la classe `Prediction` comme les activités citées précédemment, pour obtenir une prédiction sur la nouvelle application, et l'affiche dans une notification. Pour afficher la notification, il est nécessaire d'appeler `createNotificationChannel` qui va créer le canal sur lequel sera envoyé la notification.

La méthode `Builder` de `NotificationCompat` permet de définir la notification, et d'inclure un `PendingIntent` dedans. Cet `Intent` est en attente et lorsque l'utilisateur cliquera sur la notification, notre `Intent notificationIntent` sera lancé, et il fera ouvrir la classe principale. Il est à noter que grâce à la méthode `putExtra` de l'`Intent` nous incluons la valeur de la prédiction. Lorsqu'elle est ouverte, l'activité principale détecte si cela a été fait via un `Intent` contenant une telle donnée et, si c'est le cas, affichera le résultat de la prédiction comme dans les autres activités de détection manuelle. Dans tous les cas, le texte de la prédiction est aussi

visible, au moins en partie, dans le texte de la notification.

4.3 Méthode de détection sur serveur

L'amélioration graduelle de la précision de la méthode grâce aux changements d'algorithme et à l'ajout de données s'accompagne néanmoins d'une potentielle augmentation des temps de calcul. En effet, dans le cas du premier prototype avec l'algorithme naïf de Bayes, la prédiction de la classe d'une application était quasiment instantanée. Cependant, la précision du modèle était inférieur à 90%. Avec le modèle de régression basé sur Random Forest qui est notre meilleur prototype, la précision approche 100%. C'est en tout cas un fait si on considère les applications bénignes et les malwares, bien que peu nombreux, disponibles sur notre téléphone. Par contre, faire une prédiction prend 10 à 15 seconde pour chaque application sur le téléphone de test, ce qui est beaucoup plus que dans le cas précédent, et devient très remarquable lorsqu'on essaie par exemple de détecter tous les malwares installés sur le téléphone d'un coup. Même si les dernières versions de l'application montrent une image de chargement pendant que le calcul est effectué, cela représente néanmoins jusqu'à 690 secondes (soit 11.5 minutes) pour scanner les 46 applications de notre MotoG LTE (1è génération). Cela peut donc devenir une nuisance pour l'utilisateur ou même paraître peu pratique voire inutilisable, notamment dans le cas où l'utilisateur pense que l'application a planté et ferme cette dernière. Et c'est sans considérer la consommation de mémoire et de batterie, que nous avons décidé de ne finalement pas mesurer dans ce travail, puisque ces valeurs de performance ne prendront un véritable sens que dans les futures étapes du projet. Au point où l'on en est, il nous suffit de remarquer que le temps de traitement à lui seul va forcer une complexification de l'architecture : on ne va pas la réduire à la simple méthode de détection sur le client avant de comparer le prototype aux autres articles sur le plan de la performance.

La solution, qui sera au moins décrite en théorie et potentiellement développée en partie, consiste à mettre en place une communication client-serveur entre le téléphone Android et un serveur doté d'une méthode de détection. La méthode pourra être la même si celle du téléphone est déjà précise : l'avantage du serveur est d'avoir accès à des composants plus puissants pour traiter les informations plus vite que le téléphone. Dans notre cas, la simple méthode d'apprentissage automatique basée sur les permissions des APK est utilisable aussi bien sur Android que sur un serveur doté de Java. Elle de plus assez simple à porter, il suffit pour nous de réutiliser la classe Prediction.java et de lui ajouter une interface adaptée.

La majeure partie du travail va alors consister à développer les classes client et serveur pour la communication par socket sur Java, puis à détecter la bande passante et optimiser

les communications. L'enjeu est alors de déterminer de manière suffisamment précise s'il vaut mieux, étant donné une certaine bande passante, envoyer les données au serveur pour traitement distant (*offloading*) ou faire le traitement en local sur le téléphone.

La première chose à savoir est la complexité de chacune des étapes de la détection, et le poids des données résultantes à chaque étape. Cela permet de savoir quelles sont les meilleures étapes à offloader. En première approche, sans faire les tests et à partir d'une intuition naïve, nous considérons que l'extraction des permissions et la comparaison avec les permissions systèmes pour obtenir un vecteur binaire sont deux opérations relativement simples, peu gourmandes (simple appel pour les permissions, puis 151 boucles sur les 151 permissions au maximum pour vérifier leur présence) et réduisent drastiquement la taille des données : à partir de l'APK complète (plusieurs méga octets de données, comprenant code et ressources) on obtient un simple vecteur binaire de longueur 151. On effectuera donc toujours cette opération sur le téléphone, et si besoin est, la partie prédiction faisant appel au modèle pré-entraîné Weka sera effectuée à distance. Cette dernière opération renvoie un simple entier compris entre -100 et 100, donc encore une fois le poids des données pour la communication est très faible.

CHAPITRE 5 CONCLUSION

5.1 Synthèse des travaux

Le travail précédent constitue les premières étapes dans l'élaboration d'un système de détection de logiciels malveillants pour environnement mobile pour l'architecture ATISCOM. Nous avons en premier lieu donné une vue d'ensemble du domaine qui permettra à l'équipe du projet d'avoir une base solide de références pour aborder les différentes branches de la détection de malware sur Android. Cette vue d'ensemble comprend d'une part les références aux malwares mobiles de l'actualité, et d'autre part une étude approfondie de la taxonomie des méthodes d'analyse de programme pour les logiciels Android de Sadeghi et al. [7]. Grâce à cette vue d'ensemble, nous avons pu déterminer que le sous-domaine le plus prometteur serait l'analyse dynamique sur l'appareil de l'utilisateur, et développé l'analyse sur cette branche de la détection. Nous y avons détaillé plusieurs articles récents et comparé leurs résultats en terme de précision et performance, pour révéler leurs lacunes de manière quantitative mais aussi nous donner des points de comparaison pour les futurs travaux. On remarque par exemple que les taux de détection (précision) approchent très souvent 99% dans les articles considérés, avec un taux de faux positif allant jusqu'à 0.1 voire moins. Il est aussi intéressant de considérer que les bases de données utilisées pour entraîner les algorithmes vont jusqu'à considérer la bagatelle de 135000 applications. On remarquera aussi que les détails sur la performance sont plus difficilement appréciables et comparables que la simple précision, lorsqu'on dresse un tableau des résultats de chaque méthode.

A partir des lacunes identifiées dans les travaux existants, qui sont multiples mais récurrentes, nous avons pu déterminer un ensemble de requis à respecter pour développer une méthode optimale. Il s'agit en premier lieu d'utiliser l'apprentissage automatique comme la plupart des articles avant nous, pour obtenir en un temps record des règles assez fiables avec un très grand nombre de données. D'autre part, nous avons pu remarquer que les travaux hybrides sont bien souvent les plus remarquables. Loin d'offrir une grande innovation, ils permettent une grande versatilité et offrent les meilleurs performances pour la meilleur précision, car les lacunes d'une de leurs facettes sont compensées par les forces d'une autre. C'est le cas en analyse dynamique sur appareil de MADAM [44] et en hybride statique/dynamique et client/serveur de SAMADroid [59], le projet le plus prometteur à ce jour. Nous avons aussi identifié le problème de la fiabilité des résultats, surtout dans le cas de la détection de malwares encore inconnus, et qui semble requérir des méthodes de détection d'anomalie,

qui peuvent elles aussi être couplées au reste. Nous avons donc décrit l'architecture vers laquelle nous pensons diriger notre projet. Il s'agit d'une architecture client-serveur mettant en oeuvre une ou plusieurs méthodes de détection complémentaires. Le serveur permettra d'une part de maintenir la base de données des échantillons et de mettre à jour les modèles d'apprentissage machine construits, et d'autre part de traiter les calculs les plus lourds à la place du client sur Android pour permettre d'accélérer les traitements. Cela se fera grâce à un lien client-serveur optimisé en temps réel pour choisir quels calculs décharger en fonction du débit des réseaux disponibles. Mais dans le cadre de cette maîtrise, c'est majoritairement le système de détection autonome pour le client qui est décrit. En effet, avant de pouvoir implémenter le système complet, il est nécessaire d'avoir une méthode fonctionnelle la plus précise possible sur le mobile. Notre choix, par souci de simplicité, et d'utiliser l'analyse statique des permissions systèmes requise par l'application en première approche. Après la description des moyens de récupérer et traiter cette information sans entrer dans les détails de l'apprentissage automatique, nous donnons une description des ensembles de données collectés et formés dans le cadre du projet, qui totalisent environ 27000 applications légitimes et 6000 malwares.

Une fois l'architecture établie, la méthode principale théorisée et les données récoltées, nous avons décrit l'implémentation et l'optimisation de cette méthode. En premier lieu, puisque celle-ci est basée sur l'apprentissage machine, nous avons dû effectuer quelques choix de données et d'algorithmes. Nous avons d'abord décrit l'implémentation et les performances de notre prototype initial utilisant le Bayes naïf entraîné avec la totalité de nos données. Une étude plus approfondie de ces résultats et des tests supplémentaires ont montré que nos résultats étaient plus probants avec un ensemble d'entraînement et de test équilibré, ce qui nous a obligé à réduire le nombre d'applications légitimes utilisées, faute de pouvoir obtenir d'autres échantillons de logiciels malveillants à l'heure actuelle. Par la suite, nous comparons l'ensemble des algorithmes de classification proposés par Weka, pour déterminer lequel est le plus adapté à notre problème. La comparaison n'est nullement théorique, nous avons simplement regardé la précision et les autres caractéristiques de chacun des modèles entraînés et testés sur nos ensembles. Parmi Simple Logistic, SMO et SGD, l'étude approfondie des matrices de confusion nous indique que SGD offre les meilleures résultats. Sa précision globale sur notre ensemble d'entraînement n'est cependant que de 94,42%, ce qui reste bas devant les articles étudiés en première partie. Nous choisissons alors d'essayer les méthodes de régression pour deux raisons : la première est qu'il s'agit d'algorithmes différents que nous pourrons eux aussi tester, et la seconde est que traiter la classe de l'application comme un nombre compris entre -100 et 100 au lieu d'un binaire équivalent à *benign* ou *malware* permet une plus

grande flexibilité pour la combinaison avec d'autres méthodes et l'indication d'un indice de certitude à l'utilisateur. Cela permettrait aussi éventuellement de construire et maintenir le modèle avec plus de précision, dans le cas où l'on veut inclure de nouvelles données. Nous reformatons donc nos ensembles de données pour les adapter à ce nouveau traitement. La comparaison des résultats des différents algorithmes de régression disponible avec Weka sur nos nouveaux ensembles d'entraînement et de tests permet de relever plusieurs résultats très prometteurs. Plusieurs algorithmes ont un coefficient de corrélation supérieur à 96,5% sur l'ensemble d'entraînement, comme Random Forest, Random Tree ou Random Committee. Si nous comparons cette donnée avec la précision globale des algorithmes de classification, il y a déjà une amélioration. Mais en regardant plus attentivement les prédictions de ces algorithmes sur notre ensemble de test, nous pouvons voir que ces derniers sont capables de détecter 100% des malwares lorsqu'on choisit un seuil très naïf pour déterminer la classe de l'application (valeur positive pour malware et valeur négative pour application bénigne). Nous choisissons donc d'utiliser l'algorithme Random Forest dans notre dernier prototype. Nous décrivons alors l'implémentation du prototype sous la forme d'une application Android disposant de plusieurs fonctionnalités faisant appel au modèle construit pour prédire les classes et indiquer l'indice de certitude à partir de la valeur numérique obtenue. Les premières fonctionnalités sont le scan manuel de fichiers sur le téléphone et le scan simple, multiple ou total des application installées. La dernière fonctionnalité est le scan automatique de toute application installée, cas de figure le plus utilisable si l'on considère l'inclusion du prototype à l'architecture de paiement mobile ATISCOM.

5.2 Limitations de la solution proposée

Les limitations de la solution proposée, puisque il s'agit d'une première approche à inclure dans un travail plus complet, sont tout de même nombreuses. Parmi les limitations évidentes, on peut compter le fait que la méthode ne fonctionne que sur Android, puisque nous nous sommes limité à ce cas. Cependant, cette décision est justifiée et dans le cadre du projet il est fort possible que l'on n'ait pas à considérer d'autres cas. Le prototype développé à ce jour est évidemment fonctionnel sur mobile uniquement, sans l'appui du serveur qui avait été prévu en partie 3. Or nous voyons les premières limitations évidentes de ceci à la fin de la partie 4 : la méthode optimale déterminée par nos comparaisons et optimisations, basée sur le Random Forest, a un temps de calcul beaucoup plus important que la Naive Bayes du début, et ce temps de traitement devient visible pour l'utilisateur avec pas moins de 10 à 15 secondes d'attente pour effectuer une prédiction sur une seule application. Il faut ajouter à cela que sans l'appui d'un serveur, le projet n'est guère maintenable, et ses

performances sont figées dans le temps. En effet, l'application construite n'est pas évolutive à l'heure actuelle, et on peut supposer que plus les malwares seront récents, plus leur détection effective sera hors de portée de notre modèle construit avec les données d'aujourd'hui. Associé à cela reste toujours le problème des nouveaux malwares. Peut-on supposer que de nouveaux types de logiciels malveillants soient détectés par un modèle construit à partir de l'étude de malwares préexistants ? Il y a une chance pour que cela soit vrai, dans la mesure où les permissions critiques d'Android ne changent pas, mais là encore notre application pourrait devenir obsolète avec les mises à jour du système. Il y a encore d'autres facteurs qui peuvent mettre en doute les résultats obtenus par notre modèle : la quantité de données utilisées pour l'entraînement, bien que non négligeable, fait encore pâle figure face à certains articles du domaine étudiés précédemment, et il est important de relever que la majorité d'entre elles ont déjà 2 à 4 an d'ancienneté. De plus, il est bon de rappeler que dans le cas des malwares, nous avons fait confiance à d'autres experts et chercheurs du domaine pour leur identification préalable, ce qui est très probablement notre meilleure chance en dehors d'une très fastidieuse ré-identification par nos propres moyens (et encore, ceux-ci sont très probablement plus limités que la communauté dans son ensemble). Par contre, dans le cas des applications jugées légitimes que nous avons téléchargé en très grand nombre sur les favoris du Google Play Store, il faut savoir que nous n'avons aucune réelle garantie, mais simplement l'hypothèse que des applications aussi connues et répandues ne pourraient être des malwares sans avoir été détectées avant par des experts touchés par leur visibilité grandissante.

5.3 Améliorations futures

Nous espérons cependant que la continuation de notre travail par les membres du projet ATISCOM et son intégration à la plateforme dans son ensemble permettront de répondre à la plupart de ces lacunes.

En ce qui concerne la quantité de malwares testés, de nombreuses autres base de données accessibles gratuitement ont d'ores et déjà été identifiées et requièrent simplement une demande d'accès, un téléchargement et un traitement des données pour l'ajouter à nos ensembles. Nous citerons en particulier VirusShare [1] dont l'accès nous a déjà été gracieusement fourni et qui contient au moins 11080 malwares Android ajoutés avant 2013, sans compter les plus récents. Un travail pourra être fait pour rechercher des données fiables en ce qui concerne les applications bénignes de l'ensemble d'entraînement et de test ainsi que la récupération des ensembles utilisés par d'autres auteurs à fins de comparaison. Il est à noter que Drebin constitue déjà une base utilisée par de nombreux auteurs.

1. `https://virusshare.com/`

Pour l'architecture client-serveur, notre prototype Android dans une version plus datée contient déjà un commencement de code réseau, et la classe Java utilisée pour la prédiction à partir du modèle peut être facilement réutilisée dans un programme Java complet pour le serveur. La première partie du travail futur serait donc de coder l'interface réseau côté serveur avant de pouvoir améliorer la communication client-serveur et ajouter de nouvelles méthodes de détection. Ces méthodes pourraient être statiques basées sur de plus nombreuses caractéristiques des applications, ou dynamiques avec l'usage d'une machine virtuelle côté serveur ou d'un accès système côté client.

L'usage du serveur permettra aussi de réunir les données de plusieurs clients et envisager une solution de maintenance du serveur et des modèles d'apprentissage automatique. Il est envisageable d'avoir un expert dans le projet ATISCOM qui alimente les ensembles et construits de nouveaux modèles, ou alors il serait possible d'inclure les résultats des prédictions effectuées par le système dans les nouveaux ensembles.

RÉFÉRENCES

[1] A. S. Jawale et J. S. Park, "A Security Analysis on Apple Pay," dans *2016 European Intelligence and Security Informatics Conference (EISIC)*, août 2016, p. 160–163.

[2] Y. Wang, C. Hahn et K. Sutrave, "Mobile payment security, threats, and challenges," dans *2016 Second International Conference on Mobile and Secure Services (MobiSecServ)*, févr. 2016, p. 1–5.

[3] A. Gharib et A. Ghorbani, "DNA-Droid : A Real-Time Android Ransomware Detection Framework," dans *Network and System Security*, ser. Lecture Notes in Computer Science. Springer, Cham, août 2017, p. 184–198. [En ligne]. Disponible : https://link.springer.com/chapter/10.1007/978-3-319-64701-2_14

[4] Y. Wang et Y. Alshboul, "Mobile security testing approaches and challenges," dans *2015 First Conference on Mobile and Secure Services (MOBISECSERV)*, févr. 2015, p. 1–5.

[5] S. D. Yalew, G. Q. Maguire, S. Haridi et M. Correia, "T2droid : A TrustZone-Based Dynamic Analyser for Android Applications," dans *2017 IEEE Trustcom/BigDataSE/ICESS*, août 2017, p. 240–247.

[6] C. Shepherd, G. Arfaoui, I. Gurulian, R. P. Lee, K. Markantonakis, R. N. Akram, D. Sauveron et E. Conchon, "Secure and Trusted Execution : Past, Present, and Future - A Critical Review in the Context of the Internet of Things and Cyber-Physical Systems," dans *2016 IEEE Trustcom/BigDataSE/ISPA*, août 2016, p. 168–177.

[7] A. Sadeghi, H. Bagheri, J. Garcia et S. Malek, "A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software," *IEEE Transactions on Software Engineering*, vol. 43, n°. 6, p. 492–530, juin 2017.

[8] L. Bordoni, M. Conti et R. Spolaor, "Mirage : Toward a Stealthier and Modular Malware Analysis Sandbox for Android," dans *Computer Security ESORICS 2017*, ser. Lecture Notes in Computer Science. Springer, Cham, sept. 2017, p. 278–296. [En ligne]. Disponible : https://link.springer.com/chapter/10.1007/978-3-319-66402-6_17

[9] H. Ruan, X. Fu, X. Liu, X. Du et B. Luo, "Analyzing Android Application in Real-Time at Kernel Level," dans *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, juill. 2017, p. 1–9.

[10] L. Xiao, Y. Li, X. Huang et X. Du, "Cloud-Based Malware Detection Game for Mobile Devices with Offloading," *IEEE Transactions on Mobile Computing*, vol. 16, n°. 10, p. 2742–2750, oct. 2017.

[11] Z. Cheng, X. Chen, Y. Zhang, S. Li et Y. Sang, “Detecting Information Theft Based on Mobile Network Flows for Android Users,” dans *2017 International Conference on Networking, Architecture, and Storage (NAS)*, août 2017, p. 1–10.

[12] M. Eslahi, M. Yousefi, M. V. Naseri, Y. M. Yussof, N. M. Tahir et H. Hashim, “Cooperative network behaviour analysis model for mobile Botnet detection,” dans *2016 IEEE Symposium on Computer Applications Industrial Electronics (ISCAIE)*, mai 2016, p. 107–112.

[13] S. Garg, S. K. Peddoju et A. K. Sarje, “Network-based detection of Android malicious apps,” *International Journal of Information Security*, vol. 16, n°. 4, p. 385–400, août 2017. [En ligne]. Disponible : https://link.springer.com/article/10.1007/s10207-016-0343-z

[14] Android Open Source project, “ART and Dalvik,” https://source.android.com/devices/tech/dalvik/, accessed : 2018-03-05. [En ligne]. Disponible : https://source.android.com/devices/tech/dalvik/

[15] M. Kumar, “Dynamic Analysis tools for Android Fail to Detect Malware with Heuristic Evasion Techniques,” https://thehackernews.com/2014/05/dynamic-analysis-tools-for-android-fail.html, accessed : 2018-03-05.

[16] “Mobile Security | Mobile App Security | Lookout, Inc.” https://www.lookout.com/, accessed : 2018-01-18.

[17] “MI:RIAM,” https://www.wandera.com/miriam/, accessed : 2018-01-18.

[18] “Technology - z9 Mobile Threat Defense | Zimperium,” https://www.zimperium.com/technology, accessed : 2018-01-18.

[19] “Mobile Protection, Enterprise Mobile Security,” https://www.skycure.com/, accessed : 2018-01-18.

[20] “HTC, Secured by D4.” [En ligne]. Disponible : https://cog.systems/htc-secured-by-d4/

[21] “Loapi Android Trojan Does All Sorts of Bad | SecurityWeek.Com,” http://www.securityweek.com/loapi-android-trojan-does-all-sorts-bad, accessed : 2018-01-18.

[22] T. Seals, “Sneaky Multi-Stage Android Malware Spreads Banking Trojans in Google Play,” https://www.infosecurity-magazine.com:443/news/sneaky-multistage-android-malware/, nov. 2017, accessed : 2018-01-19.

[23] “Android Malware Steals Data from Social Media Apps | SecurityWeek.Com,” http://www.securityweek.com/android-malware-steals-data-social-media-apps, accessed : 2018-01-18.

[24] "Switcher : Android joins the attack-the-router club," https://securelist.com/switcher-android-joins-the-attack-the-router-club/76969/, accessed : 2018-01-18.

[25] T. Seals, "Android Trojan Targets 200+ Global Financial Apps," https://www.infosecurity-magazine.com:443/news/android-trojan-targets-200-global/, janv. 2018, accessed : 2018-01-18.

[26] R. Cozza, I. Durand et A. Gupta, "Market share : Ultramobiles by region, os and form factor, 4q13 and 2013," *Gartner Market Research Report*, 2014.

[27] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel et A. N. Sheth, "TaintDroid : an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, nº. 2, p. 5, 2014.

[28] V. Rastogi, Y. Chen et W. Enck, "AppsPlayground : automatic security analysis of smartphone applications," dans *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 2013, p. 209–220.

[29] W. Enck, D. Octeau, P. D. McDaniel et S. Chaudhuri, "A Study of Android Application Security." dans *USENIX security symposium*, vol. 2, 2011, p. 2.

[30] E. Chin, A. P. Felt, K. Greenwood et D. Wagner, "Analyzing Inter-application Communication in Android," dans *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA : ACM, 2011, p. 239–252. [En ligne]. Disponible : http://doi.acm.org/10.1145/1999995.2000018

[31] A. P. Felt, E. Chin, S. Hanna, D. Song et D. Wagner, "Android Permissions Demystified," dans *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA : ACM, 2011, p. 627–638. [En ligne]. Disponible : http://doi.acm.org/10.1145/2046707.2046779

[32] Y. Zhou, Z. Wang, W. Zhou et X. Jiang, "Hey, you, get off of my market : detecting malicious apps in official and alternative android markets." dans *NDSS*, vol. 25, 2012, p. 50–52.

[33] W. Enck, M. Ongtang et P. McDaniel, "On Lightweight Mobile Phone Application Certification," dans *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA : ACM, 2009, p. 235–245. [En ligne]. Disponible : http://doi.acm.org/10.1145/1653662.1653691

[34] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau et P. McDaniel, "IccTA : Detecting Inter-component Privacy Leaks in

Android Apps," dans *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA : IEEE Press, 2015, p. 280–291. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?id=2818754.2818791

[35] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau et P. McDaniel, "FlowDroid : Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," dans *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA : ACM, 2014, p. 259–269. [En ligne]. Disponible : http://doi.acm.org/10.1145/2594291.2594299

[36] W.-C. Wu et S.-H. Hung, "DroidDolphin : A Dynamic Android Malware Detection Framework Using Big Data and Machine Learning," dans *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, ser. RACS '14. New York, NY, USA : ACM, 2014, p. 247–252. [En ligne]. Disponible : http://doi.acm.org/10.1145/2663761.2664223

[37] B. Amos, H. Turner et J. White, "Applying machine learning classifiers to dynamic Android malware detection at scale," dans *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, juill. 2013, p. 1666–1671.

[38] E. S. Jeong, I. S. Kim et D. H. Lee, "SafeGuard : A Behavior Based Real-time Malware Detection Scheme for Mobile Multimedia Applications in Android Platform," *Multimedia Tools Appl.*, vol. 76, n°. 17, p. 18 153–18 173, sept. 2017. [En ligne]. Disponible : https://doi.org/10.1007/s11042-016-4189-1

[39] A. Ruiz-Heras, P. García-Teodoro et L. Sánchez-Casado, "ADroid : anomaly-based detection of malicious events in Android platforms," *International Journal of Information Security*, vol. 16, n°. 4, p. 371–384, août 2017. [En ligne]. Disponible : https://link.springer.com/article/10.1007/s10207-016-0333-1

[40] A. Saracino, D. Sgandurra, G. Dini et F. Martinelli, "MADAM : Effective and Efficient Behavior-based Android Malware Detection and Prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, n°. 1, p. 83–97, janv. 2018.

[41] T. Bhatia et R. Kaushal, "Malware detection in android based on dynamic analysis," dans *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*, juin 2017, p. 1–6.

[42] W. Fan, Y. Sang, D. Zhang, R. Sun et Y. Liu, "DroidInjector : A process injection-based dynamic tracking system for runtime behaviors of Android applications," *Computers & Security*, vol. 70, p. 224–237, sept. 2017. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S0167404817301207

[43] M. Zheng, M. Sun et J. C. S. Lui, “DroidTrace : A ptrace based Android dynamic analysis system with forward execution capability,” dans *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, août 2014, p. 128–133.

[44] L. Onwuzurike, E. Mariconti, P. Andriotis, E. De Cristofaro, G. Ross et G. Stringhini, “MaMaDroid : Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version),” *arXiv:1711.07477 [cs]*, nov. 2017, arXiv : 1711.07477. [En ligne]. Disponible : http://arxiv.org/abs/1711.07477

[45] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen et Y. Rahulamathavan, “PIndroid : A novel Android malware detection system using ensemble learning methods,” *Computers & Security*, vol. 68, p. 36–46, juill. 2017. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S0167404817300640

[46] M. Ahmadi, A. Sotgiu et G. Giacinto, “IntelliAV : Building an Effective On-Device Android Malware Detector,” *arXiv:1802.01185 [cs]*, févr. 2018, arXiv : 1802.01185. [En ligne]. Disponible : http://arxiv.org/abs/1802.01185

[47] S. Aonzo, A. Merlo, M. Migliardi, L. Oneto et F. Palmieri, “Low-Resource Footprint, Data-Driven Malware Detection on Android,” *IEEE Transactions on Sustainable Computing*, p. 1–1, 2017.

[48] M. Melis, D. Maiorca, B. Biggio, G. Giacinto et F. Roli, “Explaining Black-box Android Malware Detection,” *arXiv:1803.03544 [cs, stat]*, mars 2018, arXiv : 1803.03544. [En ligne]. Disponible : http://arxiv.org/abs/1803.03544

[49] M. K. Alzaylaee, S. Y. Yerima et S. Sezer, “EMULATOR vs REAL PHONE : Android Malware Detection Using Machine Learning,” dans *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, ser. IWSPA ’17. New York, NY, USA : ACM, 2017, p. 65–72. [En ligne]. Disponible : http://doi.acm.org/10.1145/3041008.3041010

[50] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel et G. Vigna, “BareDroid : Large-Scale Analysis of Android Apps on Real Devices,” dans *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA : ACM, 2015, p. 71–80. [En ligne]. Disponible : http://doi.acm.org/10.1145/2818000.2818036

[51] I. Burguera, U. Zurutuza et S. Nadjm-Tehrani, “Crowdroid : behavior-based malware detection system for android,” dans *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, p. 15–26.

[52] M. K. Yoon, S. Mohan, J. Choi, M. Christodorescu et L. Sha, “Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded

System," dans *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, avr. 2017, p. 191–196.

[53] S. Sen, E. Aydogan et A. I. Aysan, "Coevolution of Mobile Malware and Anti-Malware," *IEEE Transactions on Information Forensics and Security*, vol. 13, n°. 10, p. 2563–2574, oct. 2018.

[54] A. Amamra, "Anomaly detection system using system calls for android smartphone system," Thèse de doctorat, École de technologie supérieure, 2015.

[55] X. Xiao, S. Zhang, F. Mercaldo, G. Hu et A. K. Sangaiah, "Android malware detection based on system call sequences and LSTM," *Multimedia Tools and Applications*, p. 1–21, sept. 2017. [En ligne]. Disponible : https://link.springer.com/article/10.1007/s11042-017-5104-0

[56] S. Chaba, R. Kumar, R. Pant et M. Dave, "Malware Detection Approach for Android systems Using System Call Logs," *arXiv:1709.08805 [cs]*, sept. 2017, arXiv : 1709.08805. [En ligne]. Disponible : http://arxiv.org/abs/1709.08805

[57] P. K. Das, A. Joshi et T. Finin, "App behavioral analysis using system calls," dans *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS) : MobiSec*, 2017.

[58] M. Sun, X. Li, J. C. S. Lui, R. T. B. Ma et Z. Liang, "Monet : A User-Oriented Behavior-Based Malware Variants Detection System for Android," *IEEE Transactions on Information Forensics and Security*, vol. 12, n°. 5, p. 1103–1112, mai 2017.

[59] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song et H. Yu, "SAMADroid : A Novel 3-Level Hybrid Malware Detection Model for Android Operating System," *IEEE Access*, vol. 6, p. 4321–4339, 2018.

[60] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon et K. Rieck, "Drebin : Effective and Explainable Detection of Android Malware in Your Pocket." Internet Society, 2014. [En ligne]. Disponible : https://www.ndss-symposium.org/ndss2014/programme/drebin-effective-and-explainable-detection-android-malware-your-pocket/

[61] M. Salehi et M. Amini, "Android Malware Detection using Markov Chain Model of Application Behaviors in Requesting System Services," *arXiv:1711.05731 [cs]*, nov. 2017, arXiv : 1711.05731. [En ligne]. Disponible : http://arxiv.org/abs/1711.05731

[62] X. Wang, Y. Yang, Y. Zeng, C. Tang, J. Shi et K. Xu, "A Novel Hybrid Mobile Malware Detection System Integrating Anomaly Detection With Misuse Detection," dans *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*, ser. MCS '15. New York, NY, USA : ACM, 2015, p. 15–22. [En ligne]. Disponible : http://doi.acm.org/10.1145/2802130.2802132

[63] M. Rangwala, P. Zhang, X. Zou et F. Li, “A taxonomy of privilege escalation attacks in Android applications,” *International Journal of Security and Networks*, vol. 9, n°. 1, p. 40, 2014. [En ligne]. Disponible : http://www.inderscience.com/link.php?id=59327

ANNEXE A Ressources pour la construction des ensembles

1 ACCEPT_HANDOVER
2 ACCESS_CHECKIN_PROPERTIES
3 ACCESS_COARSE_LOCATION
4 ACCESS_FINE_LOCATION
5 ACCESS_LOCATION_EXTRA_COMMANDS
6 ACCESS_NETWORK_STATE
7 ACCESS_NOTIFICATION_POLICY
8 ACCESS_WIFI_STATE
9 ACCOUNT_MANAGER
10 ADD_VOICEMAIL
11 ANSWER_PHONE_CALLS
12 BATTERY_STATS
13 BIND_ACCESSIBILITY_SERVICE
14 BIND_APPWIDGET
15 BIND_AUTOFILL_SERVICE
16 BIND_CARRIER_MESSAGING_SERVICE
17 BIND_CARRIER_SERVICES
18 BIND_CHOOSER_TARGET_SERVICE
19 BIND_CONDITION_PROVIDER_SERVICE
20 BIND_DEVICE_ADMIN
21 BIND_DREAM_SERVICE
22 BIND_INCALL_SERVICE
23 BIND_INPUT_METHOD
24 BIND_MIDI_DEVICE_SERVICE
25 BIND_NFC_SERVICE
26 BIND_NOTIFICATION_LISTENER_SERVICE
27 BIND_PRINT_SERVICE
28 BIND_QUICK_SETTINGS_TILE
29 BIND_REMOTEVIEWS
30 BIND_SCREENING_SERVICE
31 BIND_TELECOM_CONNECTION_SERVICE
32 BIND_TEXT_SERVICE
33 BIND_TV_INPUT
34 BIND_VISUAL_VOICEMAIL_SERVICE
35 BIND_VOICE_INTERACTION
36 BIND_VPN_SERVICE
37 BIND_VR_LISTENER_SERVICE
38 BIND_WALLPAPER
39 BLUETOOTH
40 BLUETOOTH_ADMIN
41 BLUETOOTH_PRIVILEGED
42 BODY_SENSORS
43 BROADCAST_PACKAGE_REMOVED
44 BROADCAST_SMS
45 BROADCAST_STICKY
46 BROADCAST_WAP_PUSH
47 CALL_PHONE
48 CALL_PRIVILEGED
49 CAMERA
50 CAPTURE_AUDIO_OUTPUT
51 CAPTURE_SECURE_VIDEO_OUTPUT
52 CAPTURE_VIDEO_OUTPUT
53 CHANGE_COMPONENT_ENABLED_STATE
54 CHANGE_CONFIGURATION
55 CHANGE_NETWORK_STATE
56 CHANGE_WIFI_MULTICAST_STATE
57 CHANGE_WIFI_STATE
58 CLEAR_APP_CACHE
59 CONTROL_LOCATION_UPDATES
60 DELETE_CACHE_FILES
61 DELETE_PACKAGES
62 DIAGNOSTIC
63 DISABLE_KEYGUARD
64 DUMP
65 EXPAND_STATUS_BAR
66 FACTORY_TEST
67 FOREGROUND_SERVICE
68 GET_ACCOUNTS
69 GET_ACCOUNTS_PRIVILEGED
70 GET_PACKAGE_SIZE
71 GET_TASKS
72 GLOBAL_SEARCH
73 INSTALL_LOCATION_PROVIDER
74 INSTALL_PACKAGES

Figure A.1 Permissions système Android (1/2)

```
INSTALL_SHORTCUT
INSTANT_APP_FOREGROUND_SERVICE
INTERNET
KILL_BACKGROUND_PROCESSES
LOCATION_HARDWARE
MANAGE_DOCUMENTS
MANAGE_OWN_CALLS
MASTER_CLEAR
MEDIA_CONTENT_CONTROL
MODIFY_AUDIO_SETTINGS
MODIFY_PHONE_STATE
MOUNT_FORMAT_FILESYSTEMS
MOUNT_UNMOUNT_FILESYSTEMS
NFC
NFC_TRANSACTION_EVENT
PACKAGE_USAGE_STATS
PERSISTENT_ACTIVITY
PROCESS_OUTGOING_CALLS
READ_CALENDAR
READ_CALL_LOG
READ_CONTACTS
READ_EXTERNAL_STORAGE
READ_FRAME_BUFFER
READ_INPUT_STATE
READ_LOGS
READ_PHONE_NUMBERS
READ_PHONE_STATE
READ_SMS
READ_SYNC_SETTINGS
READ_SYNC_STATS
READ_VOICEMAIL
REBOOT
RECEIVE_BOOT_COMPLETED
RECEIVE_MMS
RECEIVE_SMS
RECEIVE_WAP_PUSH
RECORD_AUDIO
REORDER_TASKS
REQUEST_COMPANION_RUN_IN_BACKGROUND
REQUEST_COMPANION_USE
            _DATA_IN_BACKGROUND
REQUEST_DELETE_PACKAGES
REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
REQUEST_INSTALL_PACKAGES
RESTART_PACKAGES
SEND_RESPOND_VIA_MESSAGE
SEND_SMS
SET_ALARM
SET_ALWAYS_FINISH
SET_ANIMATION_SCALE
SET_DEBUG_APP
SET_PREFERRED_APPLICATIONS
SET_PROCESS_LIMIT
SET_TIME
SET_TIME_ZONE
SET_WALLPAPER
SET_WALLPAPER_HINTS
SIGNAL_PERSISTENT_PROCESSES
STATUS_BAR
SYSTEM_ALERT_WINDOW
TRANSMIT_IR
UNINSTALL_SHORTCUT
UPDATE_DEVICE_STATS
USE_BIOMETRIC
USE_FINGERPRINT
USE_SIP
VIBRATE
WAKE_LOCK
WRITE_APN_SETTINGS
WRITE_CALENDAR
WRITE_CALL_LOG
WRITE_CONTACTS
WRITE_EXTERNAL_STORAGE
WRITE_GSERVICES
WRITE_SECURE_SETTINGS
WRITE_SETTINGS
WRITE_SYNC_SETTINGS
WRITE_VOICEMAIL
```

Figure A.2 Permissions système Android (2/2)

ANNEXE B Code de l'application Android

```
1  public class Prediction {
2
3          private static String[] systemPerms = {
4        \\liste des 151 permissions
5      }
6          private static int model = R.raw.apk_perm_5_5_randomforestreg;
7          private static InputStream modelFile;
8      private static int threshold = -30; // C]-100;100[
9
10         public static String NumApkPred (ApplicationInfo appInfo, Context
               context){
11             modelFile = context.getResources().openRawResource(model);
12             int result = NumArffPred(MakeNumArff(appInfo, context));
13             String prediction = IsMalware(result);
14             return prediction;
15         }
16
17         private static String IsMalware(int intPred){
18             String cl = "";
19             float confidence;
20             if(intPred == -9999){
21                 return "ERROR - Can't make prediction";
22             }else{
23                 if (intPred>=threshold){
24                     cl = "malware";
25                     confidence = 100*(intPred-threshold)/(100-threshold);
26                 }else{
27                     cl = "benign";
28                     confidence = 100*(intPred-threshold)/(-100-threshold);
29                 }
30
31                 return cl + " (confidence:"+confidence+"%)";
32             }
33         }
34
35         public static String[] GetPermissions(ApplicationInfo app, Context
               context) {
36             PackageManager pm = context.getPackageManager();
37             PackageInfo packageInfo;
38             try{
```

```
39              packageInfo = pm.getPackageInfo(app.packageName,
                    PackageManager.GET_PERMISSIONS);
40
41          } catch (PackageManager.NameNotFoundException e) {
42              packageInfo = null; // /!\ I guess
43              e.printStackTrace();
44          }
45          return GetPermAll(packageInfo);
46      }
47
48      static private String[] GetPermAll(PackageInfo packageInfo){
49          String[] requestedPermissions = packageInfo.requestedPermissions;
50          int newTableSize = 0;
51          if(requestedPermissions == null) requestedPermissions = new String
                [0];
52          for(String perm : requestedPermissions) {
53              if(perm.startsWith("android.permission.")) newTableSize++;
54          }
55          String[] sysPerm = new String[newTableSize];
56          int index = 0;
57          for(String perm : requestedPermissions) {
58              if(perm.startsWith("android.permission.")){
59                  sysPerm[index] = perm.substring(19);
60                  index++;
61              }
62          }
63          return sysPerm;
64      }
65
66      private static double[] ComparePerm(String[] actualSysPerm, String[]
            apkPerm){
67          double[] result = new double[actualSysPerm.length+1];
68          for(int i = 0; i < actualSysPerm.length; i++){
69              boolean permFound = false;
70              for(String perm : apkPerm){
71                  permFound = permFound || (actualSysPerm[i].equals(perm));
72              }
73              result[i] = permFound ? 0 : 1;  //puts 0 for ARFF if
                    permission found, or 1 if not found
74          }
75          result[actualSysPerm.length] = 0; //fills the double[] with "0"
                for malware by default but this has no influence on actual
                prediction
76          return result;
```

```
77          }
78
79          //this one with numeric attribute for is_malware instead of a binary
                class (MakeArff)
80          private static Instances MakeNumArff (ApplicationInfo appInfo, Context
                context) {
81              int length = systemPerms.length;
82              ArrayList<Attribute> atts = new ArrayList<>(length+1);
83              for(String perm : systemPerms){
84                  atts.add(new Attribute(perm,0));
85              }
86              atts.add(new Attribute("IS_MALWARE"));
87              Instances dataRaw = new Instances("TestInstances",atts,0);
88              double[] instanceValue1 = ComparePerm(systemPerms, GetPermissions(
                    appInfo, context));
89              dataRaw.add(new DenseInstance(1.0, instanceValue1));
90        return dataRaw;
91          }
92
93          private static int NumArffPred (Instances data) {
94        int result = -9999;
95              try {
96                  //avoid arrayindexoutofbounds
97                  if (data.classIndex() == -1)
98                      data.setClassIndex(data.numAttributes() - 1);
99                  Classifier cls = (Classifier) weka.core.SerializationHelper.
                        read(modelFile);
100                 double value = cls.classifyInstance(data.instance(0));     //in
                        this case we have ARFF with only one instance
101                 result = (int) value;
102             } catch (Exception e) {
103                 Log.v("Prediction(ownARFF)", e.toString());
104                 result = -9999;
105             }
106             return result;
107         }
108 }
```

Figure B.1 Prediction.java : Classe Java pour effectuer les prédictions avec Weka

```
1 public class MainActivity extends AppCompatActivity {
2     private static final int MY_PERMISSIONS_REQUEST_READ_EXTERNAL_STORAGE = 0;
3     static {
```

```
4           System.loadLibrary("native-lib");
5       }
6
7       @Override
8       protected void onCreate(Bundle savedInstanceState) {
9           super.onCreate(savedInstanceState);
10          setContentView(R.layout.activity_main);
11          Toolbar myToolbar = (Toolbar) findViewById(R.id.my_toolbar);
12          setSupportActionBar(myToolbar);
13
14          if (ContextCompat.checkSelfPermission(this, Manifest.permission.
                READ_EXTERNAL_STORAGE)
15                  != PackageManager.PERMISSION_GRANTED) {
16              ActivityCompat.requestPermissions(this,
17                      new String[]{Manifest.permission.READ_EXTERNAL_STORAGE},
                            MY_PERMISSIONS_REQUEST_READ_EXTERNAL_STORAGE);
18          }
19          //bundle and dialog for receiver from notification on new install of
                other app
20          Bundle extras = getIntent().getExtras();
21          if(extras != null && extras.getString("prediction") != null){
22              String prediction = prediction =  extras.getString("prediction");
23
24              AlertDialog.Builder builder = new AlertDialog.Builder(this);
25              //builder.setMessage(appInfo.packageName + " is " +  result)
26              builder.setMessage(prediction)
27                      .setPositiveButton(R.string.ok, new DialogInterface.
                            OnClickListener() {
28                          public void onClick(DialogInterface dialog, int id) {
29                          }
30                      });
31              AlertDialog sd = builder.create();
32              sd.show();
33          }
34      }
35
36      public void scanInstalled(View view){
37          Log.v("MainAct","open activity to scan installed app");
38          Intent intent = new Intent(this, InstalledAppActivity.class);
39          startActivity(intent);
40      }
41
42      public void findApk(View view){
43          Log.v("MainAct","open activity to find apk file");
```

```
44          Intent intent = new Intent(this, FileBrowserActivity.class);
45          startActivity(intent);
46      }
47  }
```

Figure B.2 MainActivity.java : Classe Java pour l'activité principale

```
1  public class FileBrowserActivity extends AppCompatActivity {
2      private static final int REQUEST_PATH = 1;
3      String curFileName;
4      EditText edittext;
5      String curFilepath;
6      private ProgressDialog progressDialog;
7
8      @Override
9      protected void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.activity_fileexplorer);
12         edittext = (EditText)findViewById(R.id.editText);
13         Toolbar myToolbar = (Toolbar) findViewById(R.id.my_toolbar);
14         setSupportActionBar(myToolbar);
15         ActionBar ab = getSupportActionBar();
16         ab.setDisplayHomeAsUpEnabled(true);
17     }
18
19     @Override
20     public boolean onCreateOptionsMenu(Menu menu) {
21         // Inflate the menu; this adds items to the action bar if it is
               present.
22         getMenuInflater().inflate(R.menu.scan_actions, menu);
23         return true;
24     }
25
26     @Override
27     public boolean onOptionsItemSelected(MenuItem item) {
28         switch (item.getItemId()) {
29             case R.id.menu_scan:
30                 Log.v("Menu", "Scan selected");
31                 if(CanScan()){
32                     Log.v("Dialog","will scan now "+curFileName);
33                     new PredictionTask().execute();
34                 }else{
35                     AlertDialog.Builder builder = new AlertDialog.Builder(this
```

```
                    );
36                  builder.setMessage("Can't scan selected file")
37                          .setPositiveButton(R.string.ok, new
                                DialogInterface.OnClickListener() {
38                              public void onClick(DialogInterface dialog,
                                    int id) {
39                                  //close dialog
40                              }
41                          });
42                  AlertDialog sd = builder.create();
43                  sd.show();
44              }
45          default:
46              return super.onOptionsItemSelected(item);
47      }
48  }
49
50  private boolean CanScan(){
51      try{
52          PackageManager pm = getPackageManager();
53          PackageInfo info = pm.getPackageArchiveInfo(curFilepath,0);
54          Log.v("PackageInfo",info.toString());
55          return true;
56      }catch(Exception e){
57          Log.v("Error CantScan",e.toString());
58          return false;
59      }
60  }
61
62  public void getfile(View view){
63      Intent intent1 = new Intent(this, FileChooser.class);
64      startActivityForResult(intent1,REQUEST_PATH);
65  }
66  protected void onActivityResult(int requestCode, int resultCode, Intent
        data){
67      if (requestCode == REQUEST_PATH){
68          if (resultCode == RESULT_OK) {
69              curFileName = data.getStringExtra("GetFileName");
70              edittext.setText(curFileName);
71              curFilepath = data.getStringExtra("GetFilePath");
72              Log.v("APK Filepath",curFilepath);
73          }
74      }
75  }
```

```
76
77      //To use the AsyncTask, it must be subclassed
78      private class PredictionTask extends AsyncTask<Void, Integer, Void>
79      {
80          String PredResult;
81          //Before running code in separate thread
82          @Override
83          protected void onPreExecute()
84          {
85              progressDialog = ProgressDialog.show(FileBrowserActivity.this,"
                    Scanning...",
86                      "Scanning the apps, please wait.", false, false);
87          }
88
89          //The code to be executed in a background thread.
90          @Override
91          protected Void doInBackground(Void... params)
92          {
93              synchronized (this)
94              {
95                  File file = new File(curFilepath);
96                  PredResult = Prediction.NumApkPred(file,getApplicationContext
                        ());
97                  Log.v("Pred-result",curFileName + " is " + PredResult);
98
99              }
100             return null;
101         }
102
103         //Update the progress
104         @Override
105         protected void onProgressUpdate(Integer... values)
106         {
107             progressDialog.setProgress(values[0]);
108         }
109
110         //after executing the code in the thread
111         @Override
112         protected void onPostExecute(Void result)
113         {
114             progressDialog.dismiss();
115             AlertDialog.Builder builder = new AlertDialog.Builder(
                    FileBrowserActivity.this);
116             builder.setMessage(curFileName + " is " + PredResult)
```

```
117                     .setPositiveButton(R.string.ok, new DialogInterface.
                            OnClickListener() {
118                         public void onClick(DialogInterface dialog, int id) {
119                             //close dialog
120                         }
121                     });
122             AlertDialog sd = builder.create();
123             sd.show();
124         }
125     }
126 }
```

Figure B.3 FileBrowserActivity.java : Classe Java pour l'activité de scan des fichiers locaux

```
 1 public class FileChooser extends ListActivity {
 2 
 3         private File currentDir;
 4         private FileArrayAdapter adapter;
 5         @Override
 6         public void onCreate(Bundle savedInstanceState) {
 7             super.onCreate(savedInstanceState);
 8             currentDir = new File(Environment.getExternalStorageDirectory().
                   getPath());
 9             fill(currentDir);
10         }
11         private void fill(File f)
12         {
13             File[]dirs = f.listFiles();
14             this.setTitle("Current Dir: "+f.getName());
15             List<Item>dir = new ArrayList<Item>();
16             List<Item>fls = new ArrayList<Item>();
17             try{
18                 for(File ff: dirs)
19                 {
20                     Date lastModDate = new Date(ff.lastModified());
21                     DateFormat formater = DateFormat.getDateTimeInstance();
22                     String date_modify = formater.format(lastModDate);
23                     if(ff.isDirectory()){
24                         File[] fbuf = ff.listFiles();
25                         int buf = 0;
26                         if(fbuf != null){
27                             buf = fbuf.length;
28                         }
```

```
29                      else buf = 0;
30                      String num_item = String.valueOf(buf);
31                      if(buf == 0) num_item = num_item + " item";
32                      else num_item = num_item + " items";
33                      dir.add(new Item(ff.getName(),num_item,date_modify,ff.
                            getAbsolutePath(),"directory_icon"));
34                  }
35                  else
36                  {
37                      fls.add(new Item(ff.getName(),ff.length() + " Byte",
                            date_modify, ff.getAbsolutePath(),"file_icon"));
38                  }
39              }
40          }catch(Exception e)
41          {
42
43          }
44          Collections.sort(dir);
45          Collections.sort(fls);
46          dir.addAll(fls);
47          if(!f.getName().equalsIgnoreCase("sdcard"))
48              dir.add(0,new Item("..","Parent Directory","",f.getParent(),"
                    directory_up"));
49          adapter = new FileArrayAdapter(FileChooser.this,R.layout.file_view
                ,dir);
50          this.setListAdapter(adapter);
51      }
52      @Override
53      protected void onListItemClick(ListView l, View v, int position, long
            id) {
54          super.onListItemClick(l, v, position, id);
55          Item o = adapter.getItem(position);
56          if(o.getImage().equalsIgnoreCase("directory_icon")||o.getImage().
                equalsIgnoreCase("directory_up")){
57              currentDir = new File(o.getPath());
58              fill(currentDir);
59          }
60          else
61          {
62              onFileClick(o);
63          }
64      }
65      private void onFileClick(Item o)
66      {
```

```
67              Intent intent = new Intent();
68              intent.putExtra("GetPath",currentDir.toString());
69              intent.putExtra("GetFileName",o.getName());
70              intent.putExtra("GetFilePath",o.getPath());
71              setResult(RESULT_OK, intent);
72              finish();
73          }
74  }
```

Figure B.4 FileChooser.java : Classe Java pour l'explorateur de fichier

```
1  public class Item implements Comparable<Item>{
2      private String name;
3      private String data;
4      private String date;
5      private String path;
6      private String image;
7
8      public Item(String n,String d, String dt, String p, String img)
9      {
10         name = n;
11         data = d;
12         date = dt;
13         path = p;
14         image = img;
15     }
16     public String getName()
17     {
18         return name;
19     }
20     public String getData()
21     {
22         return data;
23     }
24     public String getDate()
25     {
26         return date;
27     }
28     public String getPath()
29     {
30         return path;
31     }
32     public String getImage() {
```

```
33          return image;
34      }
35      public int compareTo(Item o) {
36          if(this.name != null)
37              return this.name.toLowerCase().compareTo(o.getName().toLowerCase()
                    );
38          else
39              throw new IllegalArgumentException();
40      }
41  }
```

Figure B.5 Item.java : Classe Java pour les fichiers de l'explorateur

```
1  public class FileArrayAdapter extends ArrayAdapter<Item>{
2      private Context c;
3      private int id;
4      private List<Item>items;
5
6      public FileArrayAdapter(Context context, int textViewResourceId,
7                              List<Item> objects) {
8          super(context, textViewResourceId, objects);
9          c = context;
10         id = textViewResourceId;
11         items = objects;
12     }
13     public Item getItem(int i)
14     {
15         return items.get(i);
16     }
17     @Override
18     public View getView(int position, View convertView, ViewGroup parent) {
19         View v = convertView;
20         if (v == null) {
21             LayoutInflater vi = (LayoutInflater)c.getSystemService(Context.
                    LAYOUT_INFLATER_SERVICE);
22             v = vi.inflate(id, null);
23         }
24         final Item o = items.get(position);
25         if (o != null) {
26             TextView t1 = (TextView) v.findViewById(R.id.TextView01);
27             TextView t2 = (TextView) v.findViewById(R.id.TextView02);
28             TextView t3 = (TextView) v.findViewById(R.id.TextViewDate);
29             ImageView imageCity = (ImageView) v.findViewById(R.id.fd_Icon1);
```

```
30              String uri = "drawable/" + o.getImage();
31              int imageResource = c.getResources().getIdentifier(uri, null, c.
                    getPackageName());
32              Drawable image = c.getResources().getDrawable(imageResource);
33              imageCity.setImageDrawable(image);
34
35              if(t1!=null)
36                  t1.setText(o.getName());
37              if(t2!=null)
38                  t2.setText(o.getData());
39              if(t3!=null)
40                  t3.setText(o.getDate());
41          }
42          return v;
43      }
44  }
```

Figure B.6 FileArrayAdapter.java : Classe Java pour l'affichage des fichiers

```
1  public class InstalledAppActivity extends AppCompatActivity {
2      private PackageManager packageManager = null;
3      static public  List<ApplicationInfo> applist = null;
4      static public ApplicationAdapter listadaptor = null;
5      private LinkedList<String> mListItems;
6      ListView listView;
7      static public int appToScanPos;
8      static public ArrayList<Integer> appsToScanPos;;
9      public static ProgressDialog progressDialog;
10     private static Context mContext;
11     public static Context getAppContext(){
12         return mContext;
13     }
14
15     @Override
16     public boolean onCreateOptionsMenu(Menu menu) {
17         // Inflate the menu; this adds items to the action bar if it is
               present.
18         getMenuInflater().inflate(R.menu.scan_allactions, menu);
19         return true;
20     }
21
22     @Override
23     public boolean onOptionsItemSelected(MenuItem item) {
```

```
24          switch (item.getItemId()) {
25              case R.id.menu_scan:
26                  // User chose the "Settings" item, show the app settings UI...
27                  Log.v("Menu", "Scan selected");
28                  DialogFragment sd;
29                  if(CanScan()) sd = new ScanDialog();
30                  else sd = new CantScanDialog();
31                  sd.show(getSupportFragmentManager(),"scan dialog");
32                  return true;
33
34              case R.id.menu_scan_all:
35                  Log.v("Menu", "Scan all selected");
36                  DialogFragment sda;
37                  sda = new ScanAllDialog();
38                  sda.show(getSupportFragmentManager(),"scan all dialog");
39                  return true;
40
41              default:
42                  // If we got here, the user's action was not recognized.
43                  // Invoke the superclass to handle it.
44                  return super.onOptionsItemSelected(item);
45          }
46      }
47
48      //changed to fill a list of apps to scan instead of verifying if only one
            APP is checked
49      public boolean CanScan(){
50          appsToScanPos = new ArrayList<Integer>();
51          int checkedAppPosition = 0;
52          while(checkedAppPosition < listadaptor.checkList.size()){
53              if(listadaptor.checkList.get(checkedAppPosition)){
54                  appsToScanPos.add(checkedAppPosition);
55              }
56              checkedAppPosition++;
57          }
58          return (appsToScanPos.size()>0);
59      }
60
61      public static class CantScanDialog extends DialogFragment {
62          @Override
63          public Dialog onCreateDialog(Bundle savedInstanceState) {
64              // Use the Builder class for convenient dialog construction
65              AlertDialog.Builder builder = new AlertDialog.Builder(getActivity
                    ());
```

```
66              builder.setMessage(R.string.cant_scan)
67                      .setPositiveButton(R.string.ok, new DialogInterface.
                            OnClickListener() {
68                          public void onClick(DialogInterface dialog, int id) {
69                              //close dialog
70                          }
71                      });
72              return builder.create();
73          }
74      }
75
76
77
78      public static class ScanDialog extends DialogFragment {
79          @Override
80          public Dialog onCreateDialog(Bundle savedInstanceState) {
81              AlertDialog.Builder builder = new AlertDialog.Builder(getActivity
                    ());
82              String question = "";;
83              for(int i : appsToScanPos){
84                  question += applist.get(i).packageName+",";
85              }
86              builder.setMessage("Do you really want to scan "+question+" for
                    malware ?")
87                      .setPositiveButton(R.string.yes, new DialogInterface.
                            OnClickListener() {
88                          public void onClick(DialogInterface dialog, int id) {
89                              new InstalledAppActivity.PredictionTask().execute
                                    ();
90
91                          }
92                      })
93                      .setNegativeButton(R.string.no, new DialogInterface.
                            OnClickListener() {
94                          public void onClick(DialogInterface dialog, int id) {
95                              Log.v("Dialog","won't scan now");
96                          }
97                      });
98              return builder.create();
99          }
100     }
101
102     static public class ScanAllDialog extends DialogFragment {
103         @Override
```

```
104         public Dialog onCreateDialog(Bundle savedInstanceState) {
105             // Use the Builder class for convenient dialog construction
106             AlertDialog.Builder builder = new AlertDialog.Builder(getActivity
                    ());
107
108             String apps = "";
109             for(ApplicationInfo app : applist){
110                 apps += app.packageName + ",";
111             }
112             Log.v("appsScan", apps);
113
114             builder.setMessage("Do you really want to scan all installed
                    applications for malware ?")
115                     .setPositiveButton(R.string.yes, new DialogInterface.
                            OnClickListener() {
116                         public void onClick(DialogInterface dialog, int id) {
117                             appsToScanPos = new ArrayList<>();
118                             for(int i = 0; i < applist.size(); i++){
119                                 appsToScanPos.add(i);
120                             }
121                             new PredictionTask().execute();
122                         }
123                     })
124                     .setNegativeButton(R.string.no, new DialogInterface.
                            OnClickListener() {
125                         public void onClick(DialogInterface dialog, int id) {
126                             Log.v("Dialog","won't scan now");
127                         }
128                     });
129             // Create the AlertDialog object and return it
130             return builder.create();
131         }
132     }
133
134     @Override
135     public void onCreate(Bundle savedInstanceState) {
136         super.onCreate(savedInstanceState);
137         mContext = this;
138         setContentView(R.layout.activity_installed_app);
139         Toolbar myToolbar = (Toolbar) findViewById(R.id.my_toolbar);
140         setSupportActionBar(myToolbar);
141         ActionBar ab = getSupportActionBar();
142         ab.setDisplayHomeAsUpEnabled(true);
143         packageManager = getPackageManager();
```

```
144         new LoadApplications().execute();
145         listView = (ListView) findViewById(android.R.id.list);
146         mListItems = new LinkedList<String>();
147         listView.setAdapter(listadaptor);
148     }
149
150     private List<ApplicationInfo> checkForLaunchIntent(List<ApplicationInfo>
            list) {
151         ArrayList<ApplicationInfo> applist = new ArrayList<ApplicationInfo>();
152         for (ApplicationInfo info : list) {
153             try {
154                 if (null != packageManager.getLaunchIntentForPackage(info.
                        packageName)) {
155                     applist.add(info);
156                 }
157             } catch (Exception e) {
158                 e.printStackTrace();
159             }
160         }
161
162         return applist;
163     }
164
165     private class LoadApplications extends AsyncTask<Void, Void, Void> {
166         private ProgressDialog progress = null;
167
168         @Override
169         protected Void doInBackground(Void... params) {
170             applist = checkForLaunchIntent(packageManager.
                        getInstalledApplications(PackageManager.GET_META_DATA));
171             listadaptor = new ApplicationAdapter(InstalledAppActivity.this,
172                     R.layout.row, applist);
173
174             return null;
175         }
176
177         @Override
178         protected void onCancelled() {
179             super.onCancelled();
180         }
181
182         @Override
183         protected void onPostExecute(Void result) {
184             listView.setAdapter(listadaptor);
```

```
185             progress.dismiss();
186             super.onPostExecute(result);
187         }
188 
189         @Override
190         protected void onPreExecute() {
191             progress = ProgressDialog.show(InstalledAppActivity.this, null,
192                     "Loading application info...");
193             super.onPreExecute();
194         }
195 
196         @Override
197         protected void onProgressUpdate(Void... values) {
198             super.onProgressUpdate(values);
199         }
200     }
201     //To use the AsyncTask, it must be subclassed
202     public static class PredictionTask extends AsyncTask<Void, Integer, Void>
203     {
204         String PredResult;
205         //Before running code in separate thread
206         @Override
207         protected void onPreExecute()
208         {
209             progressDialog = ProgressDialog.show(getAppContext(),"Scanning..."
                    ,
210                     "Scanning the apps, please wait. (0/"+appsToScanPos.size()
                        +")", false, false);
211         }
212 
213         //The code to be executed in a background thread.
214         @Override
215         protected Void doInBackground(Void... params)
216         {
217             /* This is just a code that delays the thread execution 4 times,
218              * during 850 milliseconds and updates the current progress. This
219              * is where the code that is going to be executed on a background
220              * thread must be placed.
221              */
222             //Get the current thread's token
223             synchronized (this)
224             {
225                 //scanning
226                 PredResult = "";
```

```
227                 for(int i : appsToScanPos){
228                     PredResult += applist.get(i).packageName + " is " +
                            Prediction.NumApkPred(applist.get(i), getAppContext())+
                            "\n";
229                     publishProgress(i);
230                 }
231                 Log.v("Pred-result",PredResult);
232             }
233             return null;
234         }
235 
236         //Update the progress
237         @Override
238         protected void onProgressUpdate(Integer... values)
239         {
240             progressDialog.setMessage("Scanning the apps, please wait. ("+
                    values[0]+"/"+appsToScanPos.size()+")");
241         }
242 
243         //after executing the code in the thread
244         @Override
245         protected void onPostExecute(Void result)
246         {
247             progressDialog.dismiss();
248             AlertDialog.Builder builder = new AlertDialog.Builder(
                    getAppContext());
249             builder.setMessage(PredResult)
250                     .setPositiveButton(R.string.ok, new DialogInterface.
                            OnClickListener() {
251                         public void onClick(DialogInterface dialog, int id) {
252                             //close dialog
253                         }
254                     });
255             AlertDialog sd = builder.create();
256             sd.show();
257         }
258     }
259 }
```

Figure B.7 InstalledAppActivity.java : Classe Java pour l'activité de scan des applications installées

```
1 public class ApplicationAdapter extends ArrayAdapter<ApplicationInfo> {
```

```
2       public List<ApplicationInfo> appsList = null;
3       private Context context;
4       private PackageManager packageManager;
5       public ArrayList<Boolean> checkList = new ArrayList<Boolean>();
6
7       public ApplicationAdapter(Context context, int textViewResourceId,
8                                 List<ApplicationInfo> appsList) {
9           super(context, textViewResourceId, appsList);
10          this.context = context;
11          this.appsList = appsList;
12          packageManager = context.getPackageManager();
13
14          for (int i = 0; i < appsList.size(); i++) {
15              checkList.add(false);
16          }
17      }
18
19      @Override
20      public int getCount() {
21          return ((null != appsList) ? appsList.size() : 0);
22      }
23
24      @Override
25      public ApplicationInfo getItem(int position) {
26          return ((null != appsList) ? appsList.get(position) : null);
27      }
28
29      @Override
30      public long getItemId(int position) {
31          return position;
32      }
33
34      @Override
35      public View getView(int position, View convertView, ViewGroup parent) {
36          View view = convertView;
37          if (null == view) {
38              LayoutInflater layoutInflater = (LayoutInflater) context
39                      .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
40              view = layoutInflater.inflate(R.layout.row, null);
41          }
42
43          ApplicationInfo data = appsList.get(position);
44          if (null != data) {
45              TextView appName = (TextView) view.findViewById(R.id.app_name);
```

```
46              TextView packageName = (TextView) view.findViewById(R.id.
                    app_package);
47              ImageView iconview = (ImageView) view.findViewById(R.id.app_icon);
48
49              CheckBox checkBox = (CheckBox) view.findViewById(R.id.cb_app);
50              checkBox.setTag(Integer.valueOf(position)); // set the tag so we
                    can identify the correct row in the listener
51              checkBox.setChecked(checkList.get(position)); // set the status as
                     we stored it
52              checkBox.setOnCheckedChangeListener(mListener); // set the
                    listener
53
54              appName.setText(data.loadLabel(packageManager));
55              packageName.setText(data.packageName);
56              iconview.setImageDrawable(data.loadIcon(packageManager));
57          }
58          return view;
59      }
60
61      CompoundButton.OnCheckedChangeListener mListener = new CompoundButton.
            OnCheckedChangeListener() {
62          public void onCheckedChanged(CompoundButton buttonView, boolean
                isChecked) {
63              checkList.set((Integer)buttonView.getTag(),isChecked); // get the
                    tag so we know the row and store the status
64          }
65      };
66  }
```

Figure B.8 ApplicationAdapter.java : Classe Java pour l'affichage des applications installées

```
1  public class AppInstalledReceiver extends BroadcastReceiver {
2      //seems to be called TWICE when "updating" an app : maybe remove
           PACKAGE_REPLACED from Manifest.xml if PACKAGE_ADDED is always called
3      @Override
4      public void onReceive(Context context, Intent intent) {
5          String packageName = intent.getData().getEncodedSchemeSpecificPart();
6          String prediction;
7          try{
8              PackageManager packageManager = context.getPackageManager();
9              ApplicationInfo appInfo = packageManager.getApplicationInfo(
                   packageName, 0);
10             prediction = "Prediction: " + appInfo.packageName + " is " +
```

```
                Prediction.NumApkPred(appInfo, context);
11          }catch(Exception e){
12              Log.e("Exception",e.toString());
13              prediction = "malware prediction failed for "+packageName;
14          }
15
16          Intent notificationIntent = new Intent(context, MainActivity.class);
17          notificationIntent.putExtra("prediction", prediction); // <-- HERE I
                PUT THE EXTRA VALUE
18          PendingIntent contentIntent = PendingIntent.getActivity(context, 0,
                notificationIntent, 0);
19
20          createNotificationChannel(context);
21          NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(
                context)
22                  .setSmallIcon(R.drawable.ic_dialog_alert_holo_light)
23                  .setContentTitle("Malware Detection")
24                  //.setContentText(packageName+"has been installed")
25                  .setContentText(prediction)
26                  .setPriority(NotificationCompat.PRIORITY_HIGH)
27                  .setContentIntent(contentIntent);
28          NotificationManagerCompat notificationManager =
                NotificationManagerCompat.from(context);
29          // notificationId is a unique int for each notification that you must
                define
30          notificationManager.notify(1, mBuilder.build());
31      }
32
33      private void createNotificationChannel(Context context) {
34          // Create the NotificationChannel, but only on API 26+ because
35          // the NotificationChannel class is new and not in the support library
36          if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
37              CharSequence name = "notif";
38              String description = "notif";
39              int importance = NotificationManager.IMPORTANCE_DEFAULT;
40              NotificationChannel channel = new NotificationChannel("notif",
                    name, importance);
41              channel.setDescription(description);
42              // Register the channel with the system; you can't change the
                    importance
43              // or other notification behaviors after this
44              NotificationManager notificationManager = context.getSystemService
                    (NotificationManager.class);
45              notificationManager.createNotificationChannel(channel);
```

```
46              }
47          }
48  }
```

Figure B.9 AppInstalledReceiver.java : Classe Java pour effectuer des prédictions sur les nouvelles installations