RÉPUBLIQUE DU CAMEROUN
****
PAIX - TRAVAIL - PATRIE
****
UNIVERSITÉ DE YAOUNDÉ 1
****
FACULTÉ DES SCIENCES
****
DÉPARTEMENT D'INFORMATIQUE
****

REPUBLIC OF CAMEROON
****
PEACE - WORK - FATHERLAND
****
UNIVERSITY OF YAOUNDÉ 1
****
FACULTY OF SCIENCE
****
DEPARTMENT OF COMPUTER SCIENCE
****

En vu d'obtention du diplôme de Master Recherche en Informatique
Option :

Informatique Fondamentale

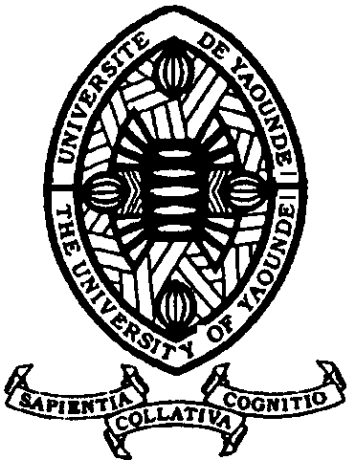
DETECTION DES APPLICATIONS MOBILES MALVEILLANTES A L'AIDE D'UN MODELE D'APPRENTISSAGE PROFOND

Présenté par :

YOUMBI WOUWE Moise Loic
Matricule :
18z2218

Supervisé par D_r **Norbert TSOPZE**

Année Académique 2020/2021



Université de Yaoundé 1
Département d'Informatique

DETECTION DES APPLICATIONS MOBILES MALVEILLANTES A L'AIDE D'UN MODELE D'APPRENTISSAGE PROFOND

Présenté et soutenu par :
YOUMBI WOUWE MOISE LOIC
18z2218

En vue d'obtention du diplôme de Master Recherche en Informatique

Sous la direction de :
Dr. Norbert TSOPZE
Chargé de Cours

Année Académique :
2019 - 2020

Dédicace

Je dédie ce mémoire à ma très chère famille et à ma progéniture.

Remerciements

Si ce travail est ce qu'il est à présent, c'est grâce au soutien, à l'appui et à la participation de plusieurs personnes et entités que je tiens à remercier. Je rends grâce au bon Dieu, qui est ma force, mon refuge et mon salut.
J'exprime ma profonde gratitude à mon encadreur Dr. Norbert TSOPZE qui, en plus de son apport académique, m'a beaucoup enseigné des leçons de vie et de milieu professionnel.
Je dis grandement merci à Mr Florentin JIECHIEU pour son soutien incontournable sur la compréhension des algorithmes du machine Learning et du Deep Learning.
Je remercie l'entreprise CYBER GUARDIANS SARL et son DG Mr. Fabrice K Ndjidie pour ce stage, le financement qu'ils m'ont offert durant la réalisation de ce mémoire et l'expérience professionnelle acquise durant ce dernier.
Merci aux membres du jury pour leur entière disponibilité et pour l'honneur qu'ils m'accordent en acceptant l'évaluation de ce travail.
Que ce travail me permette de rendre hommage à tous les enseignants pour leur disponibilité tout au long de ma formation.
Merci à l'Université de Yaoundé1 notamment au département Informatique pour la qualité d'enseignant durant mon parcourt.
Merci à l'équipe UMMISCO pour ce confortable cadre de travail mis à notre disposition.
Merci à l'Université de Douala notamment à l'ex département de Mathématique-Informatique de nous avoir fait bénéficier d'une double formation en Mathématiques et Informatique, qui par ailleurs nous a aidée dans la compréhension des bases mathématiques du Machine Learning.
Merci à mes parents qui m'ont toujours soutenu, encouragé, éduqué et veillent sur moi pour mon bien être. Merci à Monsieur DJOUMBI MOUNGOUE Charles pour son accueil chaleureux à Douala.
Merci à Mr. Fabrice YAMGA pour de nombreux conseils qu'il me donne.
Merci à Mme Pascaline TCHOUKOUACHI pour les nombreuses aides qu'elle m'a toujours offertes durant mon cursus académique. Merci à mes frères et sœurs pour leurs encouragements et leurs soutiens.
Merci à mon fils Jores Kameni et à mes nièces chéries du fait qu'en les élevant je me dois d'être responsable. Merci à tous mes camarades pour leurs encouragements et nos échanges.
Merci également à mes oncles et tantes, cousins et cousines et à tous ceux-là que je n'ai pas pu énumérer pour leur soutien infini qu'ils m'ont accordé de prêt ou de loin.

Table des matières

Table des figures

Liste des tableaux

Résumé

Chapitre 1

Introduction Générale

1.1 Introduction

Le marché des Smartphones et tablettes s'étend de plus en plus chaque jour. L'évolution de ces appareils et l'augmentation de la demande des utilisateurs poussent les développeurs à produire des logiciels pour ces appareils. Cette production inonde le marché d'un nombre incalculable d'applications mobiles, qui apportent de nombreux avantages aux utilisateurs. Cependant, cette popularité de la plateforme mobile est, en fait, une arme à double tranchant qui a, aussi, fait d'elle une cible importante d'attaques quotidiennes. En effet, environ 2000 applications malveillantes envahissent le store Android chaque jour (Sophos, 2014). Ce flux considérable d'incessantes attaques contre les appareils mobiles, affecte non seulement la vie privée d'un très grand nombre d'utilisateurs, mais aussi l'économie des organisations intégrant les applications mobiles dans leurs infrastructures de technologies de l'information. Par conséquent, ceci rend la sécurité de cette plateforme une des préoccupations les plus importantes et prioritaires. En réalité, nombreuses applications au vue de l'utilisateur ne sont pas forcément malicieuses dans certains cas, mais peuvent être extrêmement dangereuses dans d'autres cas. Ces applications dangéreuses sont appelées malwares. On peut citer à titre d'exemple, une application qui a la faculté de prendre des captures d'écran. Cette application ne contient pas de portion de code malveillant, ce qui empêche sa détection par les antis malwares, mais si cette application fonctionne en arrière-plan, et que l'utilisateur ouvre une autre application qui permet la gestion de son compte bancaire, la première application pourrait bien être malveillante en prenant des captures d'écran du mot de passe de l'utilisateur. Plusieurs scénarios encore plus complexes que cet exemple peuvent se présenter, et peuvent éventuellement affecter la vie privée de l'utilisateur.

Cette croissance exponentielle d'applications mobiles s'explique par le fait que tout le monde que ce soit un développeur tiers ou une entreprise, a la possibilité de développer une application et la déployer gratuitement sur des plates formes comme Google Play. Une application mobile est un logiciel applicatif développé pour un terminal électronique mobile et qui oofre un service précis pour l'utilisateur. Elles sont pour la plupart distribuées depuis des plateformes de téléchargement (parfois elles-mêmes contrôlées par les fabricants de smartphones) telles l'App store (plateforme d'Apple), Google Play (plateforme de Google/ Android), ou encore le Microsoft store (plateforme de Microsoft pour Windows 10 mobile). Chacun des systemes d'exploitations mobile dispose d'un langage de programmation propre

à lui, permettant le développement de ses applications. Les applications pour les terminaux Apple sont développées dans un langage principalement dédié à ces applications mobiles, le Swift. Celles de Windows Mobile, sont développées en C#, le langage aussi utilisé pour les programmes exécutables des Microsoft Windows (.exe). Le système d'exploitation Android utilise quant à lui, un langage universel, le java. Dans le cadre de ce mémoire, nous nous intéréssons à la détection des applications mobiles malveillantes fonctionnants sur le système Android. Les applications fonctionnants sur les terminaux Apple ou Windows mobile ne sont pas prises en compte.

1.2 Problématique

De nombreux travaux de recherche se sont attaquées au problème de détection des malwares fonctionnant sur le système Android. Ces travaux peuvent être regroupés en deux catégories : certains compilent l'APK avant d'analyser, d'autres par contre désassemblent ce dernier et ensuite se basent soit sur le code source ou sur les appels de méthodes. Les limites des approches du premier groupe sont le fait d'installer et d'exécuter absolument l'APK et ensuite connaitre tous les chemins d'exécution. Ce qui demande beaucoup de resssources. Ces approches du second groupe sont limitées par le fait qu'elles ne prennent pas en compte la trace d'exécution d'un APK ou aussi la redéfinition des appels de méthodes d'API. La question à laquelle ce mémoire contribue à la réponse est : comment améliorer la détection des applications mobiles malveillantes en prenant en compte toutes caractéristiques de l'APK (code source, trace d'exécution, intention, ...) ?

1.3 Méthodologie

Pour répondre à ce problème, nous proposons une approche basée sur les codes sources et les chemins d'exécution. En effet, un APK désassemblé produit plusieurs informations parmi lesquelles son code source et les permissions. Les permissions présentent les demandes explicites d'accès à des ressources de l'appreil mobile alors que le code source décrit les instructions à exécuter. Parmi ces instructions, figurent les appels de méthodes qui expriment les différents chemins d'exécutions du programme. La méthodologie proposée est donc décrite ainsi : une fois l'APK désassemblé, nous analyserons le code source de deux façons : traiter le code source (les fichiers java et AndroidManifest.xml) comme le texte avec les outils de NLP, et extraire le graphe d'appel de méthodes pour avoir la trace d'exécution de l'APK. Le code source est par la suite transformé en matrices de numérique par un algorithme de représentation de texte et le graphe d'appel de méthode transformé en une matrice par une méthode de représentation de graphe. Les matrices obtenues sont par la suite concatenées et soumises en entrée à un CNN qui classera l'APK concerné comme bénigne ou comme malveillante.

1.4 Plan du mémoire

Le chapitre 2 sera axé sur les généralités et état de l'art, le chapitre 3 portera sur la méthodologie, le chapitre 4 portera sur les expérimentations et enfin le chapitre 5 sera sur la conclusion et les perspectives.

Chapitre 2

Généralités et état de l'art

Dans ce chapitre, nous présenterons le système d'exploitation Android, la structure des APKs fonctionnant sur ce dernier, ses vulnérabilités et enfin l'ensemble des travaux effectués autour de ce sujet.

2.1 Système Android

Dans le guide du développeur, **Android** est défini comme étant une pile logicielle, c'est-à-dire un ensemble de logiciels destinés à fournir une solution clé en main pour les appareils mobiles.

2.1.1 Structure du système Android

La pile d'Android comporte un système d'exploitation (comprenant le noyau Linux), les applications clés telles que le navigateur web, le téléphone et le carnet d'adresse ainsi que des logiciels intermédiaires entre système d'exploitation et les applications. L'ensemble est organisé en couches distinctes :

- Le noyau Linux avec les pilotes,
- Des bibliothèques logicielles telles que :Webkit/Blink, OpenGl ES, SQLite ou FreeType
- Un environnement d'exécution et des bibliothèques permettant d'exécuter des programmes prévus pour la plateforme java,
- Un lot d'applications standard qui comprend un environnement de bureau.

Androïd supporte les intentions explicites et implicites. Une application peut définir les composants cibles directement dans l'Intent (intention explicite) ou demander au système Androïd de rechercher les composants en se basant sur les données de l'Intent (intention implicite).
Le noyau Linux, utilisé pour les fondations d'Android, fournit les services classiques des systèmes d'exploitation : utilisation des périphériques, accès aux réseaux de télécommunication, manipulation de la mémoire et des processus et contrôle d'accès.

2.1.2 Android et la plateforme Java

Jusqu'à la version 4.4, Android comporte une machine virtuelle nommée Dalvik, qui permet d'exécuter des programmes prévus pour la plate-forme java. C'est une machine virtuelle

conçue dès le départ pour les appareils mobiles et leurs ressources réduites (peu de puissance de calcul et peu de mémoire). La majorité, voire la totalité des applications sont exécutées par la machine virtuelle Dalvik.
Le byte code de Dalvik est différent de celui de la machine virtuelle java d'Oracle (JVM) et le processus de construction d'une application est différent : le code source de l'application en langage java, est tout d'abord compilé avec un compilateur standard qui produit un byte code pour JVM (Byte code standard de la plateforme java) puis ce dernier est traduit en byte code pour Dalvik par un programme inclus dans Android, du byte code qui pourra alors être exécuté par le systeme d'exploitation Android [3].
L'ensemble de la bibliothèque standard d'Androïd ressemble à J2SE (Java Standard Edition) de la plateforme java. La principale différence est que la bibliothèque graphique AWT et Swing sont remplacées par des bibliothèques d'Android. Le développement d'applications pour Android s'éffectue avec un ordinateur personnel sous Mac OS, Windows ou Linux en utilisant le JDK de la plateforme java et des outils pour Android. Des outils qui permettent de manipuler le téléphone ou la tablette, de la simuler par une machine virtuelle, de créer des fichiers APK (les fichiers de paquet d'Android), de déboguer les applications et d'y ajouter une signature numérique. Ces outils sont mis à disposition sur la forme de plugin pour l'environnement de développement éclipse.
La bibliothèque d'Android permet la création d'interfaces graphiques selon un procédé similaire à java FX : l'interface graphique peut être construite par déclaration et peut être utilisée avec plusieurs chartes graphiques. La programmation consiste à déclarer la composition de l'interface dans des fichiers XML ; la description peut comporter des ressources (des textes et des pictogrammes). Ces déclarations sont ensuite transformées en objets tels que des fenêtres et des boutons, qui peuvent être manipulés par de la programmation java. Les écrans ou les fenêtres (activités dans le jargon d'Android), sont remplis de plusieurs vues.
A partir de la version 5.0, l'environnement d'exécution ART (Android Runtime) remplaça la machine virtuelle Dalvik pour pallier ses limites et celles du système. Avec ART, contrairement à Dalvik, les fichiers de package d'application Android (portant l'extension .apk) ne sont plus lancés directement, mais décompressés et lancés avec de nouvelles bibliothèques et API.

2.2 Applications mobiles

Une application mobile est un logiciel applicatif, un programme téléchargeable sur smartphone ou tablette qui comporte un fichier qui est installé puis exécuté par le système d'exploitation de notre mobile.
L'extention d'une application mobile est **APK**. Ce fichier ($nom_fichier.APK$)$estcodéen$: $javaouKotlinpourAndroid(smartphoneettablette)ObjectiveCouSwiftpourIOS(appareilsApple)$.
$Lefichierapkcontientlesdossierssuivants$: *Ledossiersrc, res, gen, assets, le fichier AndroidManifest.xml et le*

Le dossier src : Ce dossier contient le code source java de notre application . Il suit les conventions standard des packages java. Par exemple, une classe *com.example.Foo* serait situé dans le dossier : *src/com/example/Foo.java*

Le dossier res : Ce dossier contient toutes les ressources de l'application et c'est là qu'on déclare la disposition en utilisant XML. Le dossier res contient tous les fichiers de mise en page, des images, des thèmes et les chaines.

Le dossier gen : Il est généré automatiquement lors de la compilation des mises en page XML dans res/. Il contient généralement un seul fichier, R.java. Ce fichier contient les constantes dont nous avons besoin de référencer les ressources de la res/ folder. Il n'est pas conseiller de modifier quoi que ce soit dans ce dossier.

Le dossier assets : Ce dossier contient des fichiers divers nécessaires à notre application. Si notre application a besoin d'un actif binaire pour fonctionner, le placer ici. Pour utiliser ces fichiers, nous avons besoin de les ouvrir à l'aide des interfaces de programmation d'application (API).

Le fichier AndroidManifest : Le manifest contient des informations essentielles au sujet de l'application que le système Android a besoin. Il comprend la définition des activités que l'application utilise, les services qu'elle fournit, les autorisations d'acces aux ressources de l'appareil et les informations de base comme le nom de l'application. C'est dans ce fichier que la plupart des développeurs définissent les intentions, les permissions, les activités et bien d'autres ressources nécessaires pour l'application. Les éléments essentiels du fichier AndroidManifest sont les suivants :

A) **Les permissions :** A chaque installation d'une nouvelle application androïde, nous donnons souvent de nombreuses autorisations aux applications, sans forcément y faire attention. Les permissions sont les différentes autorisations que demande une application Androïde avant d'être installée. Nous avons souvent tendance à accepter les demandes de permission sans regarder les autorisations que nous laissons à l'application en question, qui abuse parfois et demande des accès qui sont loin d'être indispensable à leur bon fonctionnement. Le système de permission est conçu pour cloisonner les différents droits d'accès aux données pour chaque utilisateur. Pour cela, un UserId est créé sur le téléphone à chaque installation d'une nouvelle application, comme sur linux. Tous les processus et les accès nécessaires à l'application utiliseront cet userID afin que celle-ci ait un accès exclusif à ses propres fichiers et qu'aucune autre application ne puisse y acceder... À moins que la permission lui soit donnée. Les différents groupes de permissions Il existe au total plus de 150 permissions sur Androïde que nous pouvons retrouver sur le site web https ://developer.android.com/reference/android/Manifest.permission . Sinon il est à noter que ce chiffre peut changer en fonction des nouvelles mises à jour d'Android. Quelques groupes de permissions que nous pouvons citer sont les suivants : *CALENDAR* (READ_CALENDAR,WRITE_CALENDAR),
CONTACTS (READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS)
PHONE (READ_PHONE_STATE, READ_PHONE_NUMBERS, CALL_PHONE, ANSWER_PHONE_CALLS, ADD_VOICEMAIL)

B) **Les intentions :** Elles sont des messages asynchrones qui permettent aux composants d'une application de demander des fonctionnalités à d'autres composants androïd. Les intentions permettent d'interagir avec nos propres composants ou composants d'autres applications. Par exemple, une activité peut démarrer une autre activité pour prendre une photo. Les intentions sont les objets de type android.Content.Intent. Le code peut l'envoyer au système androïde pour définir les composants que nous ciblons. Par exemple, avec la méthode startActivity(), nous pouvons définir que cet intention doit être utilisé pour démarrer une activité.

Les fichiers .dex :
Celui-ci est le bytecode Dalvik de l'application [11, 9].

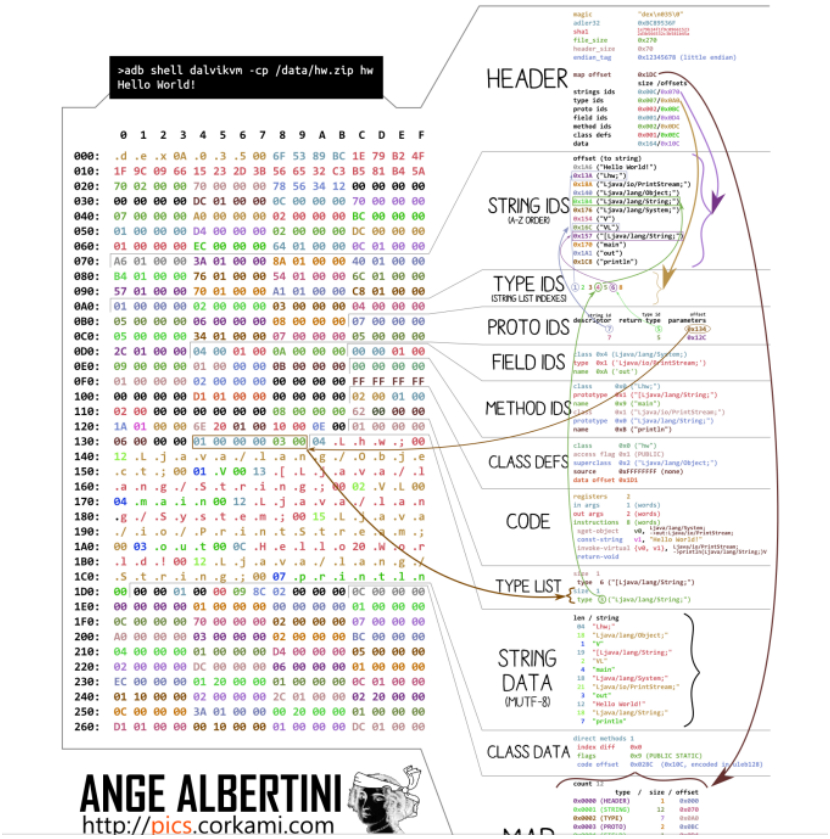


FIGURE 2.1 – contenu du fichier .DEX apres désarchivage de l'APK

comme le montre la Figure 2.1,

- L'en-tête contient des informations sur l'organisation du fichier dex
- Identifiant des chaines de caracteres : C'est une liste d'identifiants de chaînes de caractères. Chaque élément pointe vers un emplacement dans la section des données où la chaîne originale est stockée.
- Identifiants des Types : Cette liste contient les identificateurs de tous les types (classes, tableaux ou types primitifs) auxquels ce fichier fait référence, qu'ils soient définis dans le fichier ou non. La chaîne d'identificateurs proprement dite est stockée dans la section des données. Les éléments de cette liste pointent vers les éléments de la liste des identificateurs de chaîne et qui à leur tour pointent vers la chaîne d'identificateur de type stockée dans la section des données.
- Identifiant des prototypes :Il s'agit d'une liste d'identificateurs de prototypes de méthodes. Chaque élément de cette liste contient trois éléments : - shorty idx pointe vers la chaîne id item du descripteur shorty de ce prototype - type de retour id Précisez le type de retour en pointant sur le poste correspondant du type

id - Paramètre Offset Offset du début du fichier à la liste des types de paramètres pour ce prototype. Il doit pointer vers l'emplacement dans la section des données. Les données qui s'y trouvent doivent être en format "liste de types". Cette valeur serait égale à zéro si aucun paramètre

— field ids : Il s'agit d'identifiants pour tous les champs visés par ce fichier, qu'ils soient définis dans le fichier ou non

— Identifiant des méthodes : Il s'agit d'identifiants pour toutes les méthodes auxquelles ce fichier fait référence, qu'elles soient définies dans le fichier ou non.

— ...

Un APK peut avoir plusieurs fichiers Classes.dex ; ceci depend du nombre d'appel de méthodes d'API utilisé dans ce dernier.

2.3 Types d'Intent

Le systeme d'exploitation Androïd supporte les Intent explicites et implicites. Une application peut définir les composants cibles directement dans l'intent (intent explicite) ou demander au système Androïde de rechercher les composants en se basant sur les données de l'Intent (Intent implicite).

L'intent Filter : Ils sont aussi utilisés pour signaler au système androïde qu'un événement particulier est survenu. Les composants androïdes enregistrent leur Intent Filter soit statiquement dans le fichier AndroidManifest.xml, soit dynamiquement dans le code. Un filtre d'intention est défini par sa catégorie, son action et ses filtres de données. Il peut également contenir des métadonnées supplémentaires. Si une intention est envoyée au système Androïde, ce dernier (systeme Android) initie la détermination du récepteur en utilisant les données contenues dans l'objet intention. En plus de cela, elle détermine les composants qui sont enregistrés pour les données de l'intention. Si plusieurs composants sont enregistrés pour le même Intent Filter, l'utilisateur peut alors décider quel composant doit être lancé.
Les intentions explicites : Les intentions explicites définissent explicitement le composant qui doit être appelé par le système Androïde en utilisant le nom de classe java comme identifiant. Si cette classe représente une activité, le système Androïde la démarrera. Les intentions explicites sont généralement utilisées au sein d'une application et sont contrôlées par le développeur de l'application.
Les intentions implicites : Les intentions implicites précisent l'action qui devrait être effectuée avec éventuellement les données pour cette action. Par exemple, demander au système androïde d'afficher une page web. Tous les navigateurs installés doivent être enregistrés avec l'intention correspondant en utilisant un Intent Filter.
Si une intention implicite est envoyée au système androïde, il recherche tous les composants qui sont enregistrés avec l'action spécifique et le type de données approprié. Si un seul composant est trouvé, Androïde démarre directement ce composant. Si plusieurs composants sont identifiés par le système androïde, l'utilisateur devra indiquer quel composant utiliser à l'aide d'une boîte de dialogue de sélection.

2.4 Vulnérabilités liées au système Android

Les vulnérabilités liées à ce systeme exploitation peuvent etre regroupées enplusieurs catégories :

2.4.1 Vulnérabilités dues au système de permissions

Le système de permissions laisse la possibilité à des failles de sécurité, on peut par exemple citer des applications qui abusent de la confiance des utilisateurs, et qui demandent des permissions excessives. Ou encore des applications sans aucune permission, mais qui arrivent à exécuter certaines fonctionnalités, en exploitant d'autres failles dans le système.

2.4.2 Vulnérabilités dues aux développeurs d'applications

Afin de séduire les développeurs d'applications, Google a mis en place un SDK gratuit pour Eclipse, ainsi qu'un SDK visuel pour les développeurs qui n'ont pas forcément de grandes compétences en matière de programmation. On peut donc se poser des questions sur la sécurité des applications, surtout que la concurrence dans le domaine des applications pousse ces développeurs à se concentrer sur les fonctionnalités apparentes que sur la sécurité des applications.

2.4.3 Vulnérabilités dues au manque d'expérience des utilisateurs

Plusieurs failles profitent du fait que les utilisateurs sont mal formés, ou encore pas assez vigilants. Ces utilisateurs ne voient pas leurs Smartphones comme étant des minis ordinateurs, et ne donnent pas assez d'importance à leurs sécurités. Ce manque de méfiance chez les utilisateurs permet à certaines menaces de se concrétiser.

2.4.4 Vulnérabilités dues au manque de fiabilité du store

Google n'exige pas de contrôle sévère sur les applications hébergées au sein du store : les applications ne sont pas vérifiées et l'identité des développeurs n'est pas vérifiée non plus. Plusieurs applications malveillantes envahissent donc le store en exploitant ce manque de contrôle. Ainsi on retrouve toutes sortes d'applications malveillantes, certaines fausses applications disponibles sur le store se font passer pour d'autres applications, comme des applications bancaires par exemple. D'autres applications malveillantes abusent des permissions autorisées par les utilisateurs afin d'exécuter des tâches malveillantes, etc.

2.5 Attaques contre le système Android

2.5.1 Attaques par abus de permissions

Comme expliqué précédemment, le modèle de sécurité Android est basé sur les permissions. L'utilisateur a donc le choix d'autoriser ou d'interdire les permissions demandées par les applications au moment de leurs installations. Ce modèle dépend fortement des choix de l'utilisateur, ce qui le rend une arme à double tranchant, car la majorité des utilisateurs Android n'ont pas les connaissances suffisantes en matière du système d'exploitation, et encore moins des notions de sécurité.
Une catégorie d'applications malveillantes tire donc avantage du manque de vigilances des utilisateurs, qui généralement ne prêtent pas attention à la liste des permissions requises par l'application lors de son installation. Ces applications peuvent donc abuser de la confiance de l'utilisateur, en demandant des permissions excessives. Ces permissions sont donc sollicitées par des portions de code malveillantes cachées dans l'application, afin d'exécuter des tâches qui peuvent affecter la confidentialité des données de l'utilisateur, ou encore effectuer des appels téléphoniques, envoyer des SMS, obtenir la liste des contacts de l'utilisateur, etc. Un bon exemple d'une application malveillante appartenant à cette catégorie est le fameux jeu de serpent TapSnake. Cette application demande à son installation la permission d'accéder aux données GPS et à internet. Une fois que les permissions sont accordées, l'application envoie les coordonnées GPS des victimes à un serveur, et qui seront utilisées à des fins commerciales.

2.5.2 Attaques exploitant les vulnérabilités des applications

Le nombre de développeurs d'applications Android augmente, mais ces derniers ne possèdent pas forcément des bonnes compétences dans le domaine de la sécurité des logiciels mobilies, car les IDE "Integrated Development Environment" de développement Android sont assez simples, et à la portée de la grande majorité de développeurs. De plus, un grand nombre de développeurs prête plus d'attention au côté fonctionnel des applications qu'à leurs sécurités. Ce qui fait que les applications installées sur le périphérique tournant sous Android peuvent comporter des vulnérabilités dues à des erreurs de conception, des erreurs de programmation, etc.

2.5.3 Attaques par collaborations entre applications

Certaines applications d'apparence légitimes peuvent avoir un risque important aux utilisateurs. Même si ces applications ne demandent pas de permissions aux utilisateurs, ils arrivent quand même à leurs fins malicieuses en se servant d'une autre application vulnérable installée sur le périphérique.
Ces applications se basent essentiellement sur les Intents afin de réaliser leurs tâches malicieuses. Supposons qu'une application A, comportant une vulnérabilité, soit

installée sur le périphérique et que cette application permet l'envoi des SMS. Si le module qui permet d'envoyer des SMS n'est pas protégé par une permission qui restreint l'accès de ce dernier à l'application A, une application B malveillante peut donc grâce à une portion de code caché, faire une requête à ce module via un Intent, pour que l'application A envoie le SMS désiré par l'application B. Plusieurs scénarios de ce type peuvent exister. Ce qui permet à des applications malveillantes d'exécuter certaines tâches sans forcément avoir les permissions nécessaires.

2.6 Attaques par espionnage des Intents

2.6.1 Écoute des intents

Quand une application A envoie un Intent vers une application B, l'application A précise l'action de l'intent (ce qu'elle attend de l'application B). L'application B grâce à un filtre déclaré dans son fichier androidManifest.xml, identifie donc si elle peut traiter l'action de l'application A ; si c'est le cas, l'intent sera récupéré par l'application B.
Plusieurs types d'attaques peuvent se faire en interceptant les données sensibles transportées de l'application A vers l'application B. Une application malveillante peut donc profiter de la situation si le développeur n'a pas protégé les intents par des permissions. Cette application peut donc, par exemple, déclarer des filtres correspondants à toutes les actions possibles, ce qui la mène à intercepter tous les intents non protégés. De plus, l'application malveillante peut modifier les données circulées dans l'intent envoyé par l'application A, et les envoyer vers l'application B, afin de réaliser des tâches malveillantes (Lacombe, 2009).

2.6.2 Détournement des activités

En plus de transporter des données, les intents servent à lancer des requêtes entre applications ; une application A peut donc lancer une activité appartenant à une application B, via les intents. Une application malveillante peut donc modifier le comportement de l'intent, afin de lancer une activité désirée par cette dernière, au lieu de l'activité légitime.

2.7 Etat de l'art sur la détection des applications malveillantes

Plusieurs travaux de recherche se sont interessés à la détection des malwares sous Android. Ces travaux sont regroupées de la maniere suivante :

2.7.1 Approche de détection basée sur les permissions, les intentions et les appels de méthodes API Android

Certains travaux à l'instar de [16, 4] utilisent l'approche basée sur les permissions en examinant la liste des permissions qui peuvent être utilisées pour identifier les applications malveillantes. L'utilisation des permissions permet d'effectuer un filtre rapide pour détecter plus de 81% des échantillons malveillants [14].
Dans le cadre d'analyse, CICAndMal2017 propose deux niveaux de classification tel que définies sur la FIGURE 2.2

— La classification binaire statique (SBC)

— La classification dynamique des logiciels malveillants (DMC)

Ce principe suppose que si la première couche basée sur la classification statique détecte un malware suspect, il y'a plus de possibilités d'intentions malveillantes dans ce dernier. De plus si un APK est détecté suspect avec la couche basée sur la partie statique, l'analyseur doit le considérer comme un malware pour la couche suivante tout en essayant de le classifier selon sa famille en se basant sur son contenu (code source). Ainsi, ils sont en mesure de réduire le risque de faire confiance à des échantillons inconnus.
La deuxième couche tente de classer les échantillons de logiciel malveillants en quatre catégories et 39 familles de logiciels malveillants tout en utilisant les appels d'API et les fonctions de trafique réseaux.
En effet, CICAndMal2017 se base premierement sur les permissions et intentions définies dans le fichier AndroidManifest.xml. La premiere couche utilise l'algorithme de RandomForest pour pouvoir faire la classification binaire et la seconde couche se base sur un reseau de neuronnes pour pouvoir classer les APKs malveillantes en famille. Les donnees du code source sont transformées en numerique en utilisant le principe de NLP pour le prétraitement et l'algorithme word2vec pour la conversion numerique. Les limites

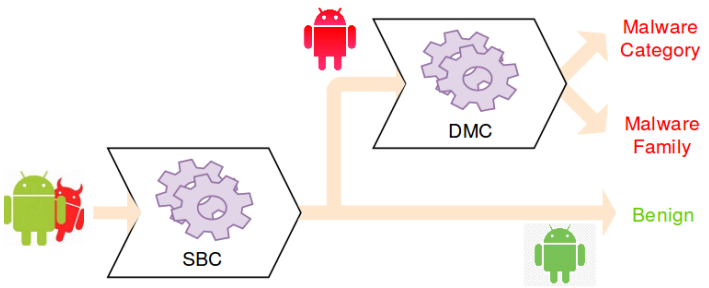


FIGURE 2.2 – Methode d'analyse de CICAndMal2017

de cette approche sont les suivantes :

— Cette approche ne permet pas d'analyser la trace d'exécution des appels de méthodes d'un APK [6, 15, 1, 17].

— Il peut aussi arriver qu'un développeur malveillant procède à la redéfinition des appels de méthodes API Android et cette intention malveillante n'est pas aussi

prise en compte par cette approche de détection.

2.7.2 Approche de détection basée sur les graphes d'appel d'API

Certaines applications ne semblent pas vraiment malveillantes, mais peuvent avoir une méthode qui utilise une autre méthode d'une autre application installée sur notre appareil. Afin de savoir exactement si une application effectue des actions malveillantes, certains travaux utilisent les graphes d'appels de méthodes d'API pour générer le chemin d'exécution d'un APK [12, 2].
Les graphes d'appel d'API sont dérivés des données collectées par le suivi des traces d'instructions et des invocations d'appels systèmes des applications androïdes données. Les comportements d'une application androïde sont considérés comme les interactions entre quatre types de composants : (activités, services, récepteurs de diffusion et observateurs de contenu). Un avantage est créé lorsqu'un composant (méthode appelant d'une activité) commence le cycle de vie de l'autre composant (la méthode appelée de l'autre activité). La génération du graphe se fait avec des outils comme *Androguard* ou *Flowdroid framework* [13]. Puis ces graphes sont transformés en vecteur numérique en utilisant des algorithmes du graph embedding tel que *node2vec* afin d'être passé à un réseau de neurone pour pouvoir détecter si une application est malveillante ou pas.
Cette approche s'avère insuffisante, car ne traite pas le code source de l'application ; vu que certains développeurs au lieu d'utiliser les méthodes propres a Android peuvent redéfinir ces méthodes afin d'effectuer leurs intentions malveillantes. Dans ce cas, il est possible qu'elle ne détecte pas.

2.7.3 Approche de détection basée sur le code source et les informations contenues dans le fichier AndroidManifest.xml

Parmi les approches de cette catégorie, nous allons présenter deux : MalDozer et ROCKY.

— **MalDozer** est l'une des techniques de détection et d'attribution de familles de logiciels malveillants pour Androïd. Ce système se base sur le code source java en s'appuyant sur le principe de la séquence brute des appels de méthode d'API de l'application extraite et apprend automatiquement les modèles malveillants ou bénins.
Tout d'abord l'APK est désassemblé en utilisant une des commandes de désassemblage telles : *dex2jar* ou *apktool*. Par la suite, l'APK est considéré comme un ensemble de sequences d'appel de methode d'API. ces derniers (appels de methode) sont extraits en suivant une expression réguliere. Afin de pouvoir transformer chaque méthode de facon unique, un dictionnaire de données est construit en associant chaque appel de méthode à un identifiant unique.
[7] transforme les identifiants des appels de méthode en vecteur numerique suivant le modèle des NLP en utilisant *word2vec* et le résultat est passé dans une couche de convolution de réseaux de neurones. Le réseau neuronal utilisé ici suit

deux étapes : (1) la première étape permet de dire si le logiciel est malveillant ou bénin et la (2) deuxième étape permet de classer les applications malveillantes en famille. Cette méthode se justifie suivant le fait que seules les fonstionnalités d'une application peuvent être malveilantes et non toute l'application entiere.

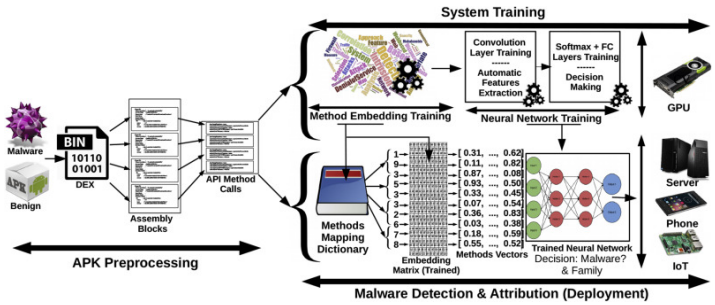


FIGURE 2.3 – Modèle de detection de MalDozer

Toutes fois nous constatons que cette méthode s'avère insuffisante dans le cas où il peut exister des intentions implicites dans l'APK. Un apk peut toujours ne pas contenir d'action malveillante dans le code, mais peut arriver à utiliser d'autres APKs déja installés sur notre appareil pour éffectuer ses actions malveillantes. c'est dans ce sens que nous pouvons dire que MalDozer presente quelques limites . Il serait plus judicieux de prendre en compte la trace d'execution des appels de methodes de l'APK pour générer tous les chemins empruntés par chaque point d'entrée ou point de sorti afin de capturer toutes les intentions implicites cachées.

— [9] **ROCKY** est une méthode de classification d'aplication malveillante se basant sur le code source contenu dans les fichiers smali. L'idee de ROCKY est d'adapter la technique de NLP afin de l'utiliser pour le pretraitement du code source issu du désassemblage des APKs dans le but de classifier. [9] prend en entrée un APK, par la suite désassemble ce dernier en extrayant uniquement le code source issu du fichier classes.DEX. Un prétraitement du texte est effectué par élimination des ponctuations et des commentaires. Les termes tels **a.get()** sont retransformés en **a, get**. [9] déclare que les ponctuations et les comentaires ne sont pas les éléments pouvant rendre une application malveillante. De plus, les termes clés du langage java sont éliminés du texte car ces derniers représentent les mots les plus fréquents dans le code. Des exemples sont : **import, java, util,**. D'autres elements tels les mots les moins frequents dans le code sont eliminés. Cette élimination s'éffectue suivant un seuil fixé. Le reste du fichier texte est utilisé pour pouvoir classer l'APK selon qu'il soit maline ou bénigne. Dans la partie de detection, Rocky se base uniquement sur le contenu du code source.
Il est à noter que ce modele tout comme [16], en plus de faire la detection parvient aussi a classer les applications malveilantes en famille en se basant sur les familles de permissions contneues dans le fichier AndroidManifest.xml.
Bien que ce modele soit aussi performant, il en ressort qu'il regorge les mêmes failles que [7] ; notamment la non prise en compte des points d'entrées ou de sorties de l'application. On peut aussi constater que les applications qui exploitent les

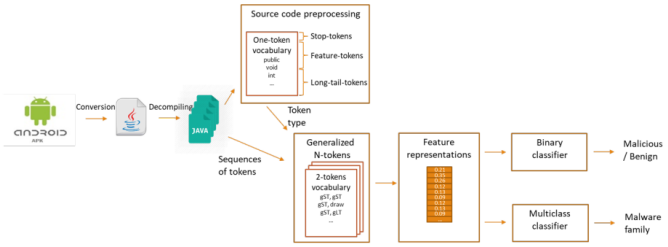


FIGURE 2.4 – Modèle de detection de ROCKY

permissions et les intentions pour un intérêt malveillant peuvent ne pas etre bien detectées du fait que la prise en compte du contenu du fichier androidManifest.xml n'est pas utilisée dans le premier volet qui est celui de la detection.

2.8 Tableau comparatif des modèles de l'état de l'art

article	Permissions et intentions	Code source java	Graphe d'appel de méthode
Maldozer	NON	OUI	NON
CICAndMal2017	OUI	OUI	NON
ROCKY	OUI	OUI	NON
Pektas2020DeepLF	NON	NON	OUI
Notre modèle	OUI	OUI	OUI

TABLE 2.1 – Tableau comparatif des modeles de l'etat de l'art

le tableau ci-dessus présente les differentes approches existantes traitant la detection des applications malveillantes fonctionnant sur le systeme Android. Nous pouvons constater que nombreuxes approches se basent sur le code source [7, 16, 9] et les informations contenues dans le fichier AndroidManifest.xml [16, 9]. D'autres par contre [12] se basent plustot sur le graphe d'appel de methode de l'APK tout en capturant tous les points d'entrees et sorties pouvant etre considerés comme vulnerabilité a ladite application.

Toutefois, ces diffenrentes methodes de l'etat de l'art presentent des failles telles : la non prise en compte de la redéfinition des appels de méthodes d'API Android, la non prise en compte des injections des intentions malveillantes dans le code source de l'application [12]. De plus, certaines apllications au lieu de contenir des portions malveillantes au sein de l'application peuvent plustot utiliser d'autres applications deja installées sur notre mobile afin d'effectuer leurs intentions malveillantes.

Chapitre 3

Méthodologie

Ce chapitre Présente la solution proposée dans le dans le cadre de ce mémoire. Le modèle que nous proposons prend en paramètre un fichier APK et le classe comme malveillant ou benigne. C'est dans ce dernier que se trouvent le code java, les permissions et les intentions issues du fichier AndroidManifest.xml et enfin les graphes obtenus par génération de tous les chemins d'exécution des appels de méthode d'API de l'APK. Lorsqu'un APK est désassemblé, tous les fichiers java sont concatenés en un seul fichier txt afin de le transformer en vecteur numérique ; les permissions et les intentions issues du fichier AndroidManifest.xml sont aussi extraites pour pouvoir capturer toutes les autorisations ou les actions que chaque APK peut contenir. Afin de savoir si un appel de méthode sort de l'APK pour appeler une action malveillante, ou aussi de savoir s'il existe des appels de méthodes pouvant s'exécuter en arrière-plan pendant l'exécution d'un APK, le graphe d'exécution de chaque APK est généré pour avoir tous les chemins d'exécution de chaque action définie dans ce dernier. Dans ce chapitre, nous allons présenter le modele proposé, la méthode de transformation des fichiers java en numerique, ensuite celle des permissions et les intentions issues des fichiers AndroidManifest.xml, et aussi celle des graphes d'appel de méthode d'API. Et enfin nous présenterons le modèle d'apprentissage profond choisi pour la détection.

3.1 Modèle proposé

La FIGURE 3.1 présente un cadre général de la solution proposée
:
Les principales actions sont :

— **Désassemblage des APKs :** ce processus consiste à obtenir le fichier classes.dex et le fichier AndroidManifest.xml de ladite application. Comme dit dans le chapitre 2, le fichier classes.dex permettra d'obtenir le code source de l'APK et aussi le graphe des appels de méthodes d'API que peut suivre cette derniere (application) pendant son execution. le fichier **AndroidManifest.xml** par contre permet d'obtenir les permissions et les intentions explicites.

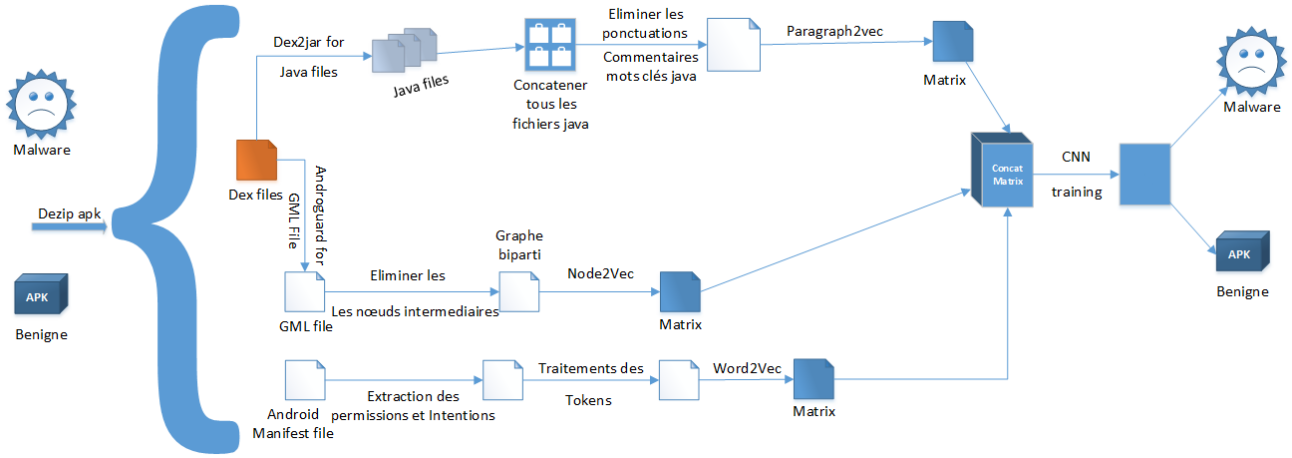


FIGURE 3.1 – Modele de détection proposé

— **Génération du graphe :** Dans le but de savoir s'il n'existe pas des actions malveillantes en arrière-plan, et aussi de savoir s'il existe des appels de méthodes qui sortent de l'application afin d'effectuer des intentions malveillantes, le graphe d'appel de méthode d'API est généré afin de capturer la trace d'exécution de chaque APK. ce dernier est possible grace au fichier *classes.dex*.

— **Elémination des noeuds intermédiaires :** afin de capturer les points d'entrée et sorties de chaque chemin d'exécution de l'APK, les noeuds intermediaires sont éléminés du graphe. Cette élémination permet d'obtenir un graphe biparti moins dense que celui initial. C'est ce dernier qui connaitra une représentation vectorielle.

— **Représentation vectorielle :** Cet étape prend en compte le graphe biparti et le transforme en numérique. Chaque noeud du graphe aura désormais une représentation unique permettant de le caractériser. Cette représentation sera exploitée dans le modele de CNN entant que caractéristique de classification de l'APK. Tout dabord, le resultat de la transformation vectorielle sera stocké dans une matrice. En d'autres termes, le graphe des chemins d'exécution des appels de méthodes d'API de chaque APK est transformé en numerique et stocké dans une matrice M3.

— **L'extraction des permissions et intentions :** cette phase permet d'extraire les information pertinentes du fichier AndroidManifest.xml afin de capturer toutes les démandes d'autorisation abusives de l'APK et aussi de capturer les intentions malveillantes de ce dernier. Le resultat est stocké dans un fichier texte.

— **Traitements des tockens :** Vu que ces informations sont souvent soit en majuscules , soit en minuscule, une convention se doit d'etre adopté : la transformation de toute information pertinante en minuscule. De plus, les éléments inutiles sont éléminés. Et le résultat de cet étape est représenté en numerique grace à une technique du word embedding.

— **Représentation vectorielle :** cette étape permet de représenter chaque permission de facon unique de telle maniere que cette information permette de savoir si l'APK est malveillante ou pas. le résultat de cette représeantation est stocké dans une matrice **M1**.

— **Désassemblage du fichier classes.Dex :** A cet étape, le code source est extrait du fichier classes.DEX par désassemblage de ce dernier.Le résultat obtenu est un ensemble de dossier dont chacun contient les fichiers java. Chaque dossier est un package contenant les fichiers java représentant les différentes classes de l'APK.

— **Concaténation des fichiers java et prétraitement :** afin de pouvoir effectuer un prétraitement des codes sources, tous les fichiers java de l'application sont concatenés en un seul fichier texte. Et par la suite les termes inutiles sont éliminés. Le résultat est transformé en numerique grace à un algorithme de word embedding.

— **Représentation vectorielle :** Le fichier obtenu apres le prétraitement du texte est transformé en numérique. La matrice **M2** est celle obtenue apres cette transformation.

—

3.2 Représentation vectorielle du fichier java

Lorsque nous désassemblons une application Androïde, nous nous basons sur le fichier *classes.dex* pour retrouver le code java. ce désassemblage transforme chaque package de l'application en un dossier et chaque classe dudit package est un fichier contenant les fichiers java.
Afin de convertir le code source en données numériques, nous concaténons tous ces fichiers en un seul fichier *txt* que nous convertissons.

- **Prétraitement du code source**

  Le fichier texte obtenu contient près de 60 000 (soixante mille mots) ; si nous voulons utiliser la méthode word2vec pour traiter chaque mot du fichier, le resultat serait une matrice de près de 60 000 (soixante milles) vecteurs. Ce qui sera très volumineux en termes de puissance de calcul.
  Pour pallier à ce défaut, nous avons classé les termes d'apres en fonction de la fréquence d'apparition. Au cours de l'expérimentation, les termes les plus fréquents sont es mots clés java. Pour chaque APK, ces mots clés java, les commentaires linéaires, multilinéaires sont supprimés. Les appels de méthodes tels **a.b**, **a.get**, sont convertis en deux mots **(a, b ; a, get)** séparés en supprimant les ponctuations [9]. Le fichier obtenu est un fichier complètement traité dont nous avons supprimé les mots clés java, les commentaires et les ponctuations.
  Le fichier texte résultant est un fichier dont le nombre de mot y apparaissant est presque réduit à moitié. Sauf pour certains APKs de plus de 400 000 (quatre cents mille) mots.
  En appliquant une conversion du texte en vecteur comme doc2vec, sa sortie ne

sera qu'un seul vecteur pour chaque APK ; cette conversion peut entrainer une perte de qualité. Toutes les parties d'un APK ne sont pas malveillantes. Ce sont des blocs de code qui le rendent malveillant d'apres [7]. C'est pour cette raison que nous utilisons un modèle de paragraphe2vec que nous ré-entrainons sur notre jeu de données vu que nous travaillons sur le code source. Le texte est divisé en près de 300 parties et chaque partie est converti en vecteur numérique de taille 100.

- **Présentation de paragraph2vec**

Notre algorithme paragraph2vec fonctionne exactement comme un modèle word2vec sauf qu'au lieu de convertir les mots, lui il converti les paragraphes ou les blocs de texte. Comme dit précédemment, nous divisons notre fichier en 300 blocs (le résultat obtenu sera 300 vecteurs) et nous étiquetons les mots de chaque paragraphe contenu dans un tableau ; donc nous avons un tableau de 300 tableaux ou chaque élément de ce tableau est un tableau formé des mots constituant le paragraphe en question. Et le tout est converti en utilisant gensim word2vec pour une phase d'entrainement. Par la suite nous sauvegardons le modèle qui a été entrainé sur tout notre jeu de données. Ce modèle est utilisé pour convertir nos données afin de constituer la première partie de notre matrice.

3.3 Méthode d'extraction et de conversion numérique des permissions et des intentions issues du fichier AndroidManifest.xml

Dans le souci de prendre en compte toutes les autorisations et les intentions implicites qu'un apk androïde peut avoir, après désassemblage de l'apk, nous extrayons toutes ces informations dans le fichier AndroidManifest.xml [16].

3.3.1 Prétraitement des permissions et des intentions

Comme décrit dans le chapitre 2 les permissions sont les différentes autorisations que nous demande une application mobile avant d'être installée. Les intentions (intentions implicite) par contre sont des éléments qui nous permettent d'appeler les methodes externes (méthode d'une autre application) à l'application. Elles sont extraites et transformées en données numériques grâce à l'algorithme word2vec. Vu qu'elles se présentent sous cette forme :**permission, android.permission.READ_SMS**, **android.permission.WRITE_SMS**, **android.permission.RECEIVE_SMS** ou **android.intent.action.AIRPLANE_MODE**, **android.intent.action.SEND**, nous faisons d'une permission ou intention un seul mot car si nous voulons supprimer les ponctuations afin de faire d'une permission plusieurs mots, nous risquerons de perdre le sens de ladite permission [9] ; raison pour laquelle nous envoyons directement cette dernière dans notre algorithme.
Le choix de *word2vec* vient du fait que, chaque permission ou intention est une action

particuliere de l'APK, de ce fait, il est plus judicieux de l'analyser élément par élément que de les analyser en bloc *(paragraph2vec ou doc2vec)*. Raison pour laquelle une permission ou intention a une representation numerique unique, permettant de la distinguer de facon unique afin de mieux l'analyser selon qu'elle soit malveillante ou benigne.

3.4 Description du graphe d'exécution d'appel de méthode d'API

L'examen des séquences d'appel de l'API peut aider à relever éfficacement les intentions d'une application. Par exemple, lorsqu'une application veut obtenir l'identifiant de l'appareil, elle doit exploiter l'appel API ou la série d'appels API de la plateforme Androïde.
Le Control Flow Graph (CFG) est une représentation graphique de tous les chemins d'exécution possibles qui sont appelés pendant l'exécution d'un APK. Dans la plateforme Androïde, chaque méthode de la classe java peut être représentée sous forme de CFG. Et dans l'ensemble, l'application entière peut être présentée par la combinaison des CFG de ses méthodes.
Formellement, un CFG est un graphe dirigé G=(N,E,T) ou N est un ensemble fini de nœuds représentant les APIs, et E représente un ensemble finis d'arêtes qui créent des liens entre des instructions successives et T est l'ensemble formé des neouds entrants et des noeuds sortants de l'APK. Un neoud (methode d'API) est dit entrant lorsqu'un service peut etre démarré à partir de ce dernier. De la meme maniere, un noeud (méthode d'API) est dit sortant lorsque l'appel de cette méthode cree un pon entre ladite application et une autre application installée sur notre mobile. De ce fait un arc(n1, n2) ou appel de méthode ; indique qu'un appel API désigné par n1 est suivi de l'appel n2 sur l'un des chemins d'exécution ou n1 et n2 sont des méthodes d'API du systeme d'exploitation Android.

3.4.1 Méthode de construction du graphe

Pour construire le graphe de contrôle d'une application entière, il faut tout d'abord le ou les points d'entrées.
Les applications Androïdes sont différentes des anciennes applications java, qui ont un point d'entrée appelé méthode principale. Cependant, les applications Androïds peuvent avoir plusieurs points d'entrée. Le *point d'entrée* agit comme une passerelle vers différents types d'évènements tels que le lancement d'un service, le toucher de l'écran, le changement de volume du téléphone, etc... invoqués par l'application simultanément par leurs gestionnaires d'événement.
En effet, le point d'entrée d'une application est la principale fonction invoquée par le systeme Android en fonction de l'apparition d'évènements prédéfinis initiée par l'utilisateur ou le système d'exploitation Androïd. Ces trois types de points d'entrée pour les applications Androïdes sont l'activité, le récepteur de diffusion et les services.
L'orsqu'un ensemble de points d'entrée principaux fournis par lesysteme d'exploitation

Android est extrait, le graphique d'appel de l'API est construit sur la base des points de départ de l'application. S'il y'a un appel vers l'autre méthode, un lien de la méthode de l'appelant vers l'autre méthode est créé afin d'inclure tous les appels API possibles de l'application Androïd. Tous les appels API accessibles à partir des points de départ sont dessinés selon l'ordre des appels.
Le graphe des appels de méthode généré est un graphe orienté attribué. ces attribues sont :

- L'identifiant du noeud,
- La description du noeud (si un noeud **a** fait partir du package **A**, et de la classe **AA**, alors la description sera $A/AA/a$,
- La relation entre ce noeud et le noeud appelé : si le noeud (méthode a appelle la méthode b, alors la relation sera : $\boldsymbol{a}$–>$\boldsymbol{b}$,
- L'adresse mémoire qu'occupe ce noeud,
- les informations relatives au dit neoud, si jamais il est un point d'entree (**ENTRYPPOINT**) pour l'APK, un point de sortie (**EXTERNAL**) ou aucun des deux.

Le fichier obtenu apres génération du graphe est tel qu'une premiere partie contient les noeuds et leurs differentes informations et la deuxieme partie est un ensemble de d'arrets définissant la relation entre deux methodes de l'APK ; si jamais il en existe. La FIGURE 3.3 ci-dessus illustre bien cette description.

```
node [
  id 3113
  label "Landroid/support/v4/view/ViewCompat$JbMr1ViewCompatImpl;->getLabelFor(Landroid/view/View;)I [access_flags=public] @ 0x39564"
  external 0
  entrypoint 0
]
node [
  id 3114
  label "Landroid/support/v4/view/ViewCompat$JbMr1ViewCompatImpl;->setLabelFor(Landroid/view/View; I)V [access_flags=public] @ 0x39580"
  external 0
  entrypoint 0
]
edge [
  source 5
  target 6
]
edge [
  source 5
  target 7
]
```

FIGURE 3.2 – Schema d'escriptif du graphe obtenu apre génération

3.4.2 Transformation numérique des graphes

- Prétraitement du graphe
  Vu qu'une approche de detection de cette méthodologie est basée sur le code source afin d'étudier le contenu de chaque APK, l'approche basée sur les graphes permet de capturer les points de sorties ou les points d'entrée d'une application donnée. En effet, les noeuds intermediaire (*compris entre le noeud entrant et le noeud sortant*) ne présentent aucune vulnérabilité dans ce cas car ils permettent juste de rester dans l'APK. Il est question ici de detecter toutes les vulnérabilités que présente un

APK. Pour ce faire, tous les chemins partants d'un noeud quelconque du graphe sont générés et par la suite les noeuds situés aux extremeité de chaque chemein sont retenus et classés selon deux ensembles : **ensemble des points entrant** et **ensemble des points de sorties**. Un neoud (une méthode) fait partir des noeuds entrants s'il n'est appelé par aucun autre noeud. De la meme maniere, un noeud (une méthode) fait partir des points de sortie s'il n'appelle aucun autre neoud.
Pour chaque noeud de l'ensemble des points d'entrées, on verifie s'il existe un chemin entre ce dernier et tout autre noeud de l'ensemble des noeuds sortants. Si tel est le cas, la relation est maintenue. Le résultat de ce prétraitement est un **graphe biparti** dont un coté est fait des méthodes entrants et un autre coté est fait des méthodes sortants .

— Représentation numérique
Le graphe biparti obtenu du prétraitement est représenté en numérique en utilisant une technique du graph embedding. Chaque noeud a une représentation unique contribuant à la détection de vulnérabilité.

3.5 Présentation des réseaux de neurones convolutifs

L'apprentissage automatique (apprentissage machine) ou apprentissage statistique est un champ d'étude de l'intelligence artificielle qui se fonde sur des approches mathématiques et statistiques pour donner aux ordinateurs la capacité d'apprendre à partir de données, c'est-à-dire d'améliorer leurs performances à résoudre des taches sans être explicitement programmés pour chacune.
Le deep Learning ou apprentissage profond est un type d'intelligence artificielle dérivé du Machine Learning. L'une de ses variantes est le réseau deneurone convolutif.

Ces réseaux sont utilisés pour tout usage autour de l'image ou de la vidéo.

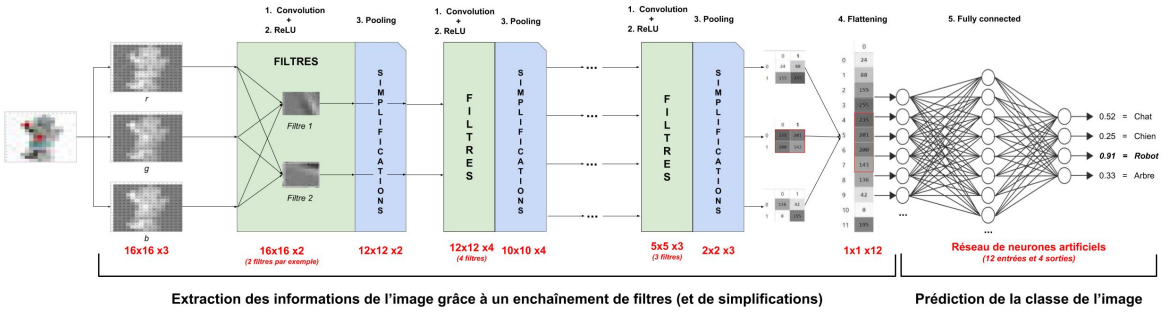


FIGURE 3.3 – Architecture standard d'un réseau convolutif

Le nom *réseau convolutif* renvoie à un terme mathématique : ***produit de convolution.***.
En termes simples, l'idée est qu'on applique un *filtre* à l'image d'entrée, les paramètres du filtre seront appris au fur et à mesure de l'apprentissage. Un filtre appris permet par exemple de détecter les angles dans une image si les angles servent à classifier au

mieux ces dernières.

Exemple de convolution 6*6 :

On peut noter deux avantages importants inhérents aux réseaux convolutifs :

— Le réseau peut apprendre par étape à reconnaître les éléments caractéristiques d'une image. Pour reconnaître un visage par exemple : il apprendra à reconnaître d'abord des paupières, des pupilles, pour arriver à identifier des yeux ;

— une fois un élément appris à un endroit de l'image le réseau sera capable de le reconnaître n'importe où d'autre dans l'image [8].

Comme le présente la FIGURE 3.4 ci-dessus, il existe quatre types de couches pour un réseau de neurones convolutif :

— **La couche de flattening**
Dernière étape de la partie "extraction des informations", le flattening consiste simplement à mettre bout à bout toutes les images (matrices) obtenues pour en faire un (long) vecteur. Les pixels (en réalité ce ne sont plus des images ou des pixels, mais des matrices de nombres, donc les pixels sont ces nombres) sont récupérés ligne par ligne et ajoutés au vecteur final.
En fait, le réseau de neurones (étape 5) prend simplement en entrée un vecteur (à chaque neurone d'entrée on envoie une seule valeur).

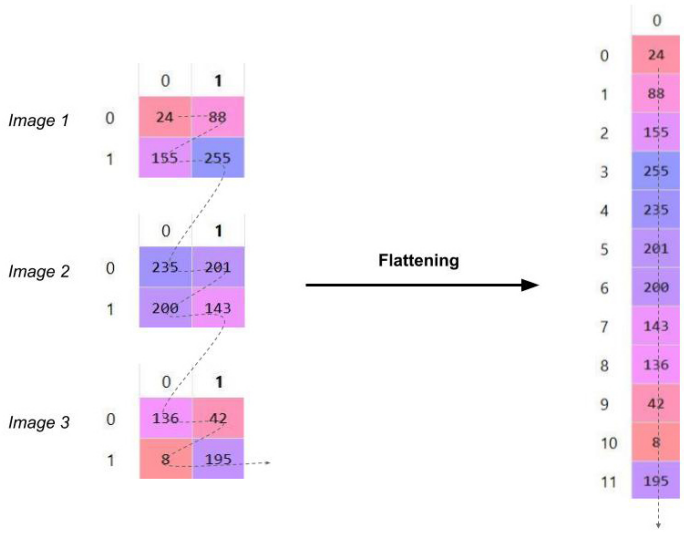


FIGURE 3.4 – Opération de flattening

Par conséquent, dans l'absolu, rien n'empêche d'utiliser un flattening qui lit les matrices par colonne, ou même qui mélange tous les "pixels" (sans en changer les valeurs), tant que le procédé de flattening reste toujours le même.

— **La couche de convolution :**
Ce premier bloc fait la particularité de ce type de réseaux de neurones, puisqu'il

fonctionne comme un extracteur de features. Pour cela, il effectue du template matching en appliquant des opérations de filtrage par convolution. La première couche filtre l'image avec plusieurs noyaux de convolution, et renvoie des "feature maps", qui sont ensuite normalisées (avec une fonction d'activation) et/ou redimensionnées.
Elle est la composante clé des réseaux de neurones convolutifs. Son but est de reperer un ensemble de filtres clés dans les images en entrée. De ce fait, un filtrage par convolution est éffectué ; le principe est de faire "glisser" une fenêtre représentant le filtre sur l'image, et de calculer le produit de convolution entre le filtre et chaque portion de l'image balayée.
La couche de convolution reçoit donc en entrée plusieurs images, et calcule la convolution de chacune d'entre elles avec chaque filtre. Les filtres correspondent exactement aux features que l'on souhaite retrouver dans les images. Le résultat obtenu pour chaque paire (image, filtre) est une carte d'activation, qui nous indique où se situent les features dans l'image : plus la valeur est élevée, plus l'endroit correspondant dans l'image ressemble à la feature. Ces filtres sont capables de déterminer tout seul les éléments discriminants d'une image, en s'adaptant au problème posé.Les noyaux des filtres désignent les poids de la couche de convolution et sont initialisés puis mis à jour par rétropropagation du gradient.

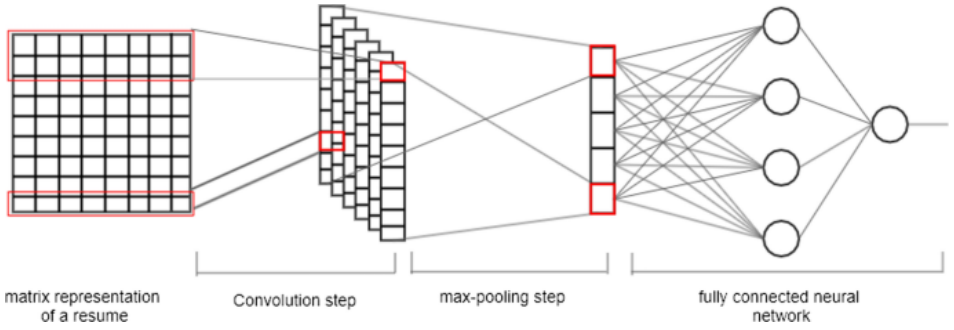


FIGURE 3.5 – Structure de la partie convolutive d'un réseau de neurone convolutif
source : [5]

— **La couche de pooling**
Elle est souvent placée entre deux couches de convolution : elle reçoit en entrée plusieurs feature maps, et applique à chacune d'entre elles l'opération de pooling. L'opération de *pooling* consiste à réduire la taille des images, tout en préservant leurs caractéristiques. Pour se faire, l'image est découpée en cellules régulière, puis la valeur maximale de chaque cellule est gardée. Afin de ne pas perdre trop d'informations, il est préférable d'utiliser despolling de petites tailles (2*2 ou 3*3). Le résultat obtenu est le meme nombre de features mapps en entrée mais de tailles reduites.
La couche de pooling permet de réduire le nombre de paramètres et de calculs dans le réseau tout en améliorant l'éfficacité du réseau et en évitant le surapprentissage. Cette couche rend moins sensible le réseau par rapport à la position des features.

Il existe plusieurs types de pooling tels :

- max pooling : permet de prendre en considération la valeur maximale de la sélection. De par sa nature, il permet de simplifier les calculs sur l'image.
- mean pooling : calcule la moyenne des pixels de la sélection. Le résultat est la valeur intermédiaire pour représenter ce lot de pixels.
- sum pooling : c'est la moyenne sans avoir divisé par le nombre de valeurs (on ne calcule que leur somme).

Pour appliquer le pooling, il faut commencer par sélectionner un carré de pixels de taille 2×2 (pour un pooling de 2×2) puis on calcule la valeur qui va venir remplacer ce carré selon l'un des poling cités ci-dessus. Ensuite, le carré est décalé vers la droite de 1 cases si le stride (= pas) vaut 1 par exemple (généralement, il vaut 1 ou 2).

— **La couche de correction ReLU**
Cette couche permet d'opérer les fonctions mathématiques sur les signaux de sorties ReLU (Rectified Linear Units) désigne la fonction réelle non-linéaire définie par ReLU(x)=max(0,x).

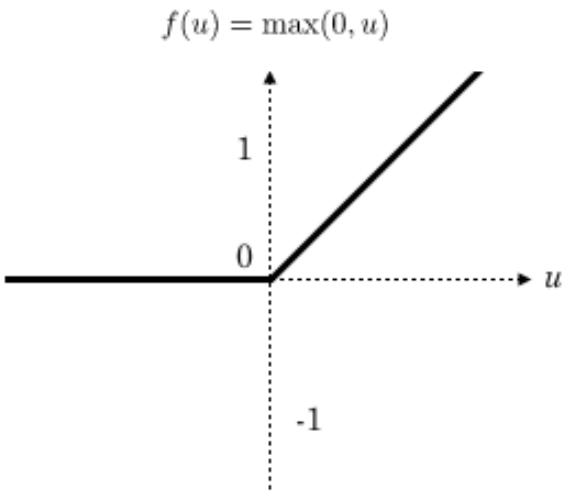


FIGURE 3.6 – Allure de la fonction ReLU

La couche de correction *ReLU* remplace donc toutes les valeurs négatives reçues en entrées par des zéros. Elle joue le rôle de fonction d'activation. Cette fonction, appelée aussi « fonction d'activation non saturante », augmente les propriétés non linéaires de la fonction de décision et de l'ensemble du réseau sans affecter les champs récepteurs de la couche de convolution. Il existe d'autres fonctions d'activations telles :

— La correction par tangente hyperbolique f(x) = $tanh(x)$,

— La correction par la tangente hyperbolique saturante : f(x) = $|tanh(x)|$

— La correction par la fonction sigmoïde f(x)=$(1+e^{-x})^{-1}$

De maniere générale, la correction Relu est préférable, car il en résulte la formation de réseau neuronal plusieurs fois plus rapide sans faire une différence significative à la généralisation de précision.

— **La couche fully-connected**

La couche completement connectée constitue toujours la dernière couche d'un réseau de neurones convolutif.

Cette couche reçoit un vecteur en entrée et produit un nouveau vecteur en sortie. Pour cela, elle applique une combinaison linéaire puis éventuellement une fonction d'activation aux valeurs reçues en entrée.

La dernière couche fully-connected permet de classifier l'image en entrée du réseau : elle renvoie un vecteur de taille N, où N est le nombre de classes du problème de classification d'images. Chaque élément du vecteur indique la probabilité pour l'image en entrée d'appartenir à une classe. Chaque valeur du tableau en entrée "vote" en faveur d'une classe. Les votes n'ont pas tous la même importance : la couche leur accorde des poids qui dépendent de l'élément du tableau et de la classe. Ce traitement revient à multiplier le vecteur en entrée par la matrice contenant les poids. Le fait que chaque valeur en entrée soit connectée avec toutes les valeurs en sortie explique le terme fully-connected. Le réseau de neurones convolutif apprend les valeurs des poids de la même manière qu'il apprend les filtres de la couche de convolution : lors de phase d'entraînement, par rétropropagation du gradient.

La couche fully-connected détermine le lien entre la position des features dans l'image et une classe. En effet, le tableau en entrée étant le résultat de la couche précédente, il correspond à une carte d'activation pour une feature donnée : les valeurs élevées indiquent la localisation (plus ou moins précise selon le pooling) de cette feature dans l'image. Si la localisation d'une feature à un certain endroit de l'image est caractéristique d'une certaine classe, alors un poids important est accordé à la valeur correspondante dans le tableau.

3.6 Choix de l'algorithme

Comment fonctionne le réseau convolutif proposé ?

La matrice obtenue par concaténation des matrices issues des graphes, code source, permissions et intentions représente un APK. Cette derniere est passée dans la partie de convolution. Vu qu'il s'agit ici des textes et non des images, le type de convolution utilisé est une conv1D afin d'extraire les caracteristique sur chaque ligne de la matrice. Les filtres définies permettent d'extraire ces dernieres la fonction d'activation utilisé est la fnction RELU. Comme définie plus haut, elle met toutes les valeurs negatives a 0. Par la suite, la couche de pooling est celle de GlobalMaxPooling cette derniere extrait pour chaque filtre donnée la valeur la plus maximale afin de rendre la detection plus optimale. La couche de Flatenning a été ommise du fait que le resultat obtenu de la couche de GlobalMaxPooling est un vecteur une ligne/ une colonne. le résultat est directement

passé dans la couche completement connectée. cette derniere fait la classification binaire de l'APK selon qu'il soit malveillant ou bénin tel que présente la FIGURE 3.8.

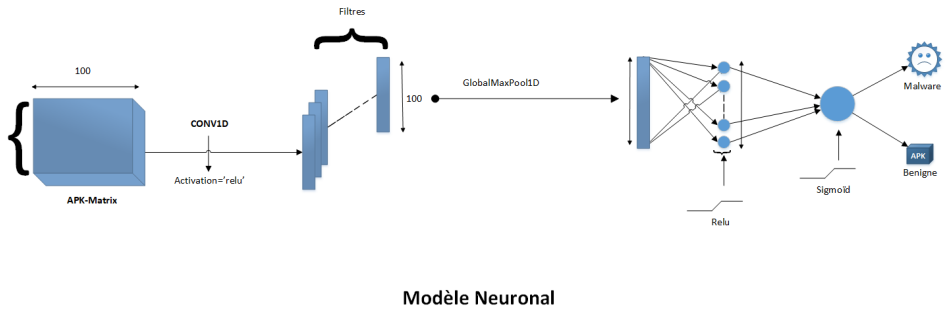


FIGURE 3.7 – Modèle neuronal proposé

Chapitre 4

Expérimentation

L'expérimentation de ce modele permettra de mieux comprendre la qualité du jeu de donnée utilisé ainsi que l'évaluation des differents algorithmes utilisés sur le plan de la complexité, le temps d'exécution et aussi le proptocole expérimental.

4.1 Présentation du jeu de données et méthodes de désassemblage des APKs

4.1.1 Jeu de données

Le jeu de données utilisé est disponible à l'url "http ://205.174.165.80/CICDataset/CICMalAnal2017/Dataset/", puisqu'il s'agit d'un apprentissage supervisé, le jeu de données contient deux classes telles une classe des APKs bénignes et une autre classe formée des APKs malwares. Tous les fichiers contenus dans ce jeu de données sont sous format .APK.
Le jeux de données téléchargé est reparti en famille de *malware Androide* telles : ***Adware_APK (721Mo), Scareware_APK (843Mo), SMSMalware (181Mo), RansomWare (100Mo)*** et des *APKs Benignes* telles : ***Benign_APK 2015 (4.0Go), Benign_APK 2016 (8.1Go), Benign_APK 2017 (6.7Go)***.
Vu qu'il s'agit d'une classification, c'est à dire prendre un APK et dire à la fin s'il est malveillant ou benin, toutes les familles de malware sont regroupées en une seule famille **(la famille des malwares)**. De même, toutes les différentes familles d'APKs bénins, sont regroupées en une seule famille **(la famille des bénins)**. En conclusion, les 02 (deux) classes de données obtenues sont : **la classe des *Malwares* et la classes des *Benins*** contenant respectivement 1669 et 1800 APKs.

4.1.2 Méthodes de désassemblage des APKs

— **Apktool**
Pour pouvoir utiliser cette commande, il faut installer une version de java >= 7 et télécharger le fichier .jar suivant ce lien par exemple (https ://ibotpeaches.github.io/Apktool/install/) de apktool sur internet afin de l'installer de

telle manière l'interpréteur de commande linux puisse le prendre en compte.

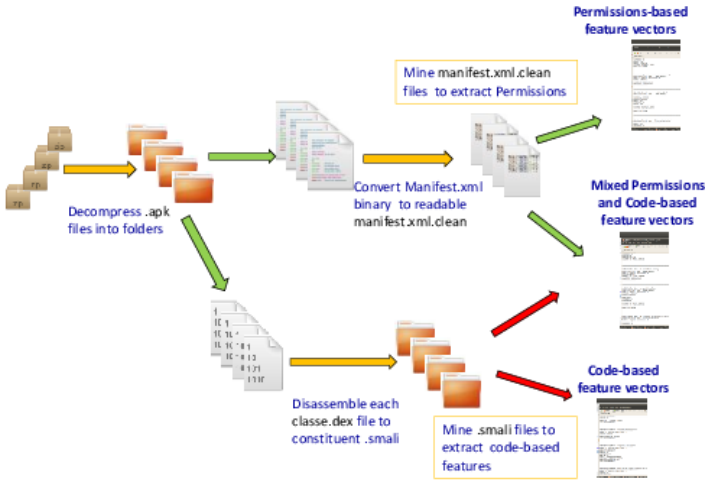


FIGURE 4.1 – Apktool : méthode de désassemblage

la commande *apktool d nomApk.apk* permet de décompiler un APK. Le résultat obtenu contient un dossier smali. Apktool désassemble un APK en transformant toutes les méthodes en fichier nomMethode.smali et la lecture de ces derniers se fait grâce à l'éditeur (jadx-gui).

— **Dex2jar**
Tout comme apktool, elle nécessite une version de java >= 7 et aussi il faut installer un gradle-build qui puisse faciliter le processus. son installation peut se faire a travers le lien suivant : (https ://sourceforge.net/p/dex2jar/wiki/UserGuide/).

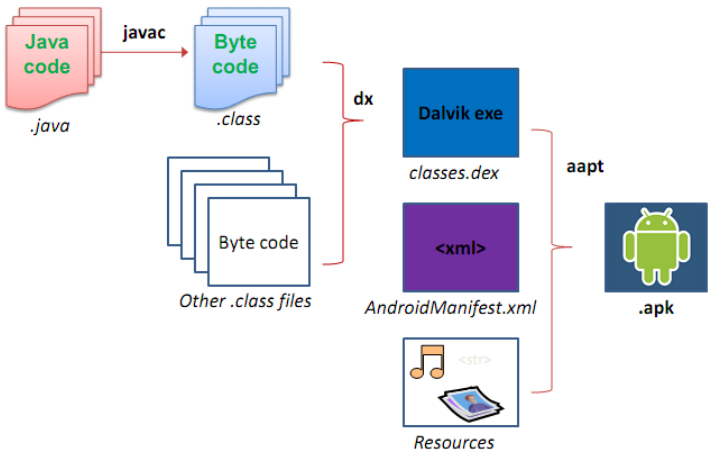


FIGURE 4.2 – Dex2jar : méthode de désassemblage

Tout daboord, il faut renommer le fichier .APk en nomApk.zip avant de proceder

au désarchivage [18]. Le résultat donne un ensemble de dossiers et de fichiers dont les plus importants sont les fichiers AndroidManifest.xml et le fichier classes.dex. Le but est de pouvoir lire le fichier classes.dex déja désassemblé en utilisant la commande *./d2j-dex2jar.sh classes.dex*. La lecture du fichier classes.dex se fait grace à l'éditeur (jadx-gui).

— **Jadx-gui :** Jadx-gui est un éditeur de texte qui permet de lire les fichiers java, class, smali, ... Son installation peut se faire en suivant le lien suivant https ://github.com/skylot/jadx .

— **Analyse des fichiers obtenus par désassemblage**
Comme dit plus haut, apktool transforme chaque méthode d'une application en un fichier smali, et tous ceux-ci sont contenus dans un même dossier, par contre dex2jar permet d'obtenir un fichier .jar que l'on peut visualiser le contenu avec un éditeur ; il est aussi possible de générer les fichiers java d'un APK en utilisant *jadx*. Par comparaison avec apktool, dex2jar regroupe toutes les méthodes d'une classe dans leur propre classe (un même dossier). De ce fait celui-ci par utilisation ou par analyse facilite la tâche en diminuant les va-et-vient.

4.2 Protocole expérimental

4.2.1 Environnement de travail

Le prétraitement des données, s'est effectué sur un système *Ubuntu20.04, processeur 64bits, corei3, 8Go de RAM*. Pour l'entraienement du modele, nous avons utilisé *google colab avec 12Go de RAM en mode GPU*.

4.2.2 Analyse temporelle des algorithmes utilisés

— Script Shell de désassemblage
La vitesse d'execution de ce script depend du volume de l'application. Plus un APK est volumineux, plus le nombre de fichier *classes.dex* l'est aussi et plus le temps du désassemblage est aussi élévé.
La FIGURE 4.3 présente le temps d'exécution en fonction du nombre de fichier clsses.dex.

Ce temps dépend à la fois du nombre de noeuds contenus dans le graphe de désassemblage, du volume du fichier AndroidManifest.xml et du nombres de fichier classes.dex contenus dans l'APK.

— Algorithmes de prétraitement

(a) Elimination des noeuds intermediaires

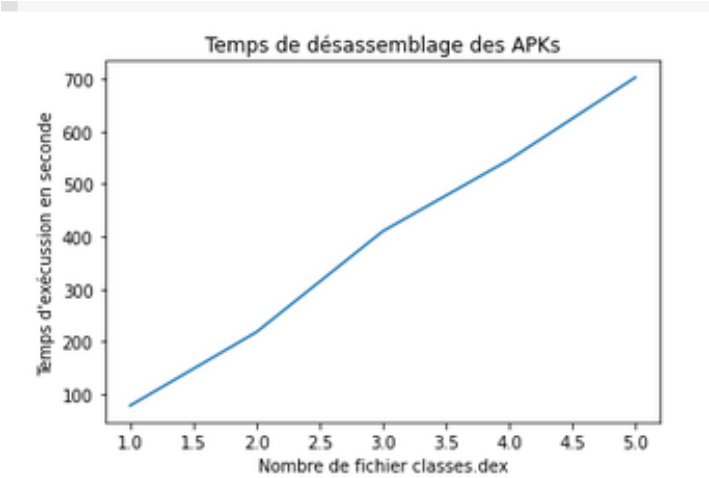


FIGURE 4.3 – Temps d'execution de l'algorithme de désassemblage des APKs en fonction du nombre de fichier classes.dex

Comme dit dans le chapitre 03, l'algorithme pour éliminer les noeuds intermediaires s'exécute en temps $O(n^2)$. De ce fait, plus le nombre de noeuds est élévé, plus le temps d'élimination l'est aussi. Ma FIGURE 4.4 Ci-contre présente le temps d'exécution de l'algorithme d'élimination des noeuds intermédiaires en fonction ds secondes.

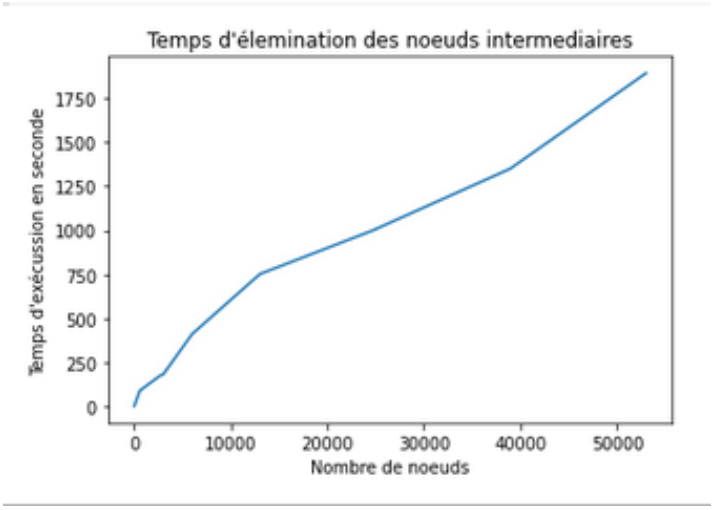


FIGURE 4.4 – élimination des noeuds intermediaires en fonction du temps

Pour éliminer les noeuds intermédiaires à partir d'un graphe donné, tous les chemins du graphe sont générés en utilisant la méthode python

get_all_shortest_paths(noeud) pour cahque noeud. Par la suite, la méthode **neighbors(i, mode="out")** est utilisée pour former les noeuds entrants. De la même manière, les noeuds sortnants sont aussi générés en utilisant la méthode **neighbors(i, mode="in")**. Pour chaque noeud de l'ensemble des points d'entrées n1, s'il existe un chemin entre ce dernier et n'importe quel autre noeud de l'ensemble des points de sortis n2, les deux noeuds sont sauvegardés, ces derniers formeront le graphe biparti qui sera transfomé. La vérification d'existance de chemin entre n1 et n2 se fait grace à la méthode **ispath(n1, n2)** de igraph.

(b) Elimination des ponctuations, commentaires et mots clés java
Afin de savoir l'ensemble des mots à éliminer dans le code source, l'algorithme de **Kmeans-clustering** est utilisé avec 03 clusters ; d'apres les résultats obtenus les mots à éliminer sont *les mots clés java*. L'élimination des ponctuations se fait aussi en utilisant la category python **string** en utilisant la liste **punctuation** comme ROCKY.
L'élimination des commentaires multilignes ou ceux tenants sur une seule ligne se fait en utilisant *les expressions régulieres java* ***r'/.*////.*'*** ce grâce à la bibliothèque **re** de python. En conclusion, ce bout de code s'exécute en quelques secondes sur le fichier résultant de la concaténation des fichiers java d'un APK, permettant un gain de temps.

(c) Extraction des permissions et intensions
Lorsqu'un APK est désassemblé, son fichier *AndroidManifest.xml* est généré en utilisant **une commande de Androguard**. Vu qu'il faut juste extraire les permissions et les intentions issus de ce fichier, le fichier est lu ligne par ligne ; dès qu'une ligne contient le mot **uses-permission**, cette derniere est récupérée afin que ladite permission soit extraite grace à la fonction **split()** sur cette ligne. Ce résultat est stocké dans un fichier. Aucun traitement n'est effectué sur ces derniers sauf la transformation de tout les lettres formant la permission en *minuscule*. La même opération est éffectuée sur les **intentions**. Sauf qu'au niveau de la récupération de la ligne contenant une telle action, nous utilisons les mots **action** ou **category**. cet Algorithme s'exécute au pire des cas en une (01) seconde sur un fichier manifest.

— Algorithme d'entrainement du modele
L'algorithme prend en entrée un APK Androide et dit à la sortie s'il est malveillant ou bénin. Vu que certains APKs du jeux de données ont un nombre de noeuds élévé, le modele **Node2vec** met plus de temps sur des APKs de plus de *cinq mille* (5000) noeuds, de ce fait chaque graphe est converti et sauvegardé dans un fichier txt. C'est ce dernier qui est utilisé dans la phase d'entrainement. le temps d'exécution de ce denier est compris entre 1heure à 2heure sur des APKs de plus de 50000 (cinquante mille) noeuds et pres de 30 minutes à 1 heure sur des APKs contenant entre 25 000 (vingt cinq mille) à 50000 (cinquante mille) noeuds. Ceux ayant un nombre de noeuds inferieur compris entre 10000 (dix mille) à 25000 (vingt cinq mille) noeuds prennent un temps compris entre 5minutes à 30minutes et enfin ceux ayant un nombre de noeuds inférieur à 5000 prennent un temps inférieur à 5 minutes voire 0.1 secondes pour des APKs ayant 34 noeuds. Les modeles **(word2vec**

ou **paragraph2vec)**, sont directement utilisés dans l'algorithme d'entrainement. Tout d'abord, ces modeles **(word2vec** et **(paragraph2vec** son entrainés sur tout le jeux de données car le travail s'effectue sur du code source. Leurs entrainements peuvent prendre pret de 5 minutes ou 25 minutes selon que ce soit respectivement celui de **word2vec** ou **paragraph2vec**.
Le modele neuronal construit utilise le **famework Tensorflow**.
Les couches de neurone sont celles issues de la bibliotheque **keras**.
Les métriques sont calculées à partir de la bibliotheque **sklearn**.
Afin de faire une meilleure classification binaire, la fonction d'activation de la couche de sortie (un seul neurone) est la **la fonction sigmoid** ; celle de la premiere couche est **la fonction relu** car elle facilite les calculs.
La fonction de correction des erreurs est le **binary_crossentropy** car elle est la mieux citée dans les articles de l'état de l'art.

Puisque la taille de chaque vecteur est 1*100, 80 filtres de tailles 1*100 sont utilisés dans la couche de **Conv1D**, le vecteur retourné par la couche **GlobalMaxPool1D()** est un vecteur de taille 1*80 car pour chaque filtre, on retiens la caractéristique maximale. Vu que la couche de **GlobalMaxPool1D()** retourne un vecteur de taille 1 * 80, l'opération de Flattenning n'est plus valable. le résultat est directement passé dans la couche *Completement connectée*.
Cette derniere (*Completement connectée*) est constituée de 02 (deux) couches telles :

— 80 neurones sur la couche d'entrée ; avec pour fonction d'activation **activation='relu'**

— 01 (un) neurone en sortie car il s'agit d'une classification binaire et sa fonction d'activation est **activation='sigmoid'**.

4.3 Résultats, Comparaisons et Discussions

4.3.1 Résultat

Comme mentionné dans la *section 3.6 du chapitre 3*, le réseau de neurones convolutifs prend en parametre deux classes de données de 2930 (deux milles neuf cents trente APKs) répartis en deux classes telles 873 (huit cent soixante treize) APKs malwares et 790 (sept cent quatre vingt dix) APKs bénignes. 80% (soit 1303 APKs) de ce dernier est utilisé pour faire l'entrainement et 20% pour le test. Cet algorithme prend environ 1h30 minutes pour s'executer et le resultat obtenu est le suivant :

— Nombre d'epochs : **21**

— Accuracy (Données d'entrainement) : **99.8%**

— Accuracy (Données de test) : **96.64%**

— Précision : **93,26%**

— Rappel : **84,63%**

4.3.2 Comparaisons

Le tableau ci-dessous présente la comparaison entre le modele proposé et celui de CICAndMal2017 vu que ce modele utilise le même jeu de données que ce dernier.

Modele	Accuracy	Précision	Rappel
Modele Proposé	96.64%	93.26%	84.63%
CICAndMal2017	—	95.3%	—

TABLE 4.1 – Tableau de comparaison des modeles

L'écart entre CICAndMal2017 et le modele prpoposé se situe au niveau de la quantité de données utilisée.

4.3.3 Discussions

Le modèle conçu a la propriété d'être adaptable sur d'autres applications codés en d'autres languages comme le C, le C#, python et autres. Pour mieux les adapter, il faut suivre la procédure de réentrainement des différents modeles utilisés dans ce dernier sur le jeux de données en question.
S'il arrive que l'on ne puisse pas générer le graphe du chemin d'exécussion par ce que le langage ne le permet pas, on peut ne pas prendre en compte l'approche basée les graphes.
Comme autre propriété de ce modele, il peut etre réentrainé sur les jeux de données plus récents afin d'etre mis à jour ou de prendre en compte les modifications apportées sur les frameworks de développements ou des appels de méthodes d'API utilisés dans le cadre de développment des APKs tournant sur le systeme Android. Ce modele a la capacité d'etre évolutive ou portative car il peut respectivement etre mis à jour ou etre utilisé dans d'autres applications.
Toutefois, il en ressort que ce modele ne soit pas parfait à case du manque de machines puissantes pour mieux l'entrainer sur des APKs de volume élévé en nombre de noeuds ou sur un grand jeux de données vu la quantité d'espace disponible sur Google Drive car la capacité des donnees utilisée est de 23Go. Afin d'être précis pour la détection au niveau de l'analyse du code source, le travail futur consistera à ne plus découper le code source en bloc de paragraphes mais d'utiliser les expressions régulieres java afin d'extraire plustot les differentes méthodes definies dans le code source qui par la suite seront transformés en numérique grâce à **paragraph2vec**.
Tous les APKs du jeux de donnéés n'ont pas été pris en compte du fait que l'algorithme qui permet d'éliminer les noeuds intermédiaires et celui du node2vec prennent assez de temps. Donc les APKs de plus de 55000 (cinquante cinq milles) noeuds ne sont pas pris en compte pour l'entrainement de ce modele (ils représentent pret de 20% de notre jeux de données). Une autre perspective est celle de la modification de l'algorithme **Node2vec** afin de reduire son temps d'exécussion sur les graphes à grand volume de noeuds.

Conclusion et Perspectives

Il était question dans ce mémoire de développer un modele de deep learning pour la détection des Androides malwares. Il existe plusieurs approches pour résoudre ce probleme : l'approche basée sur l'extraction des appels de methode d'API Androide et l'utilisation des permissions, des intentions contenues dans le fichier AndroidManifest.xml, l'approche basée sur l'analyse du code source en representant chaque mot du code source comme un élément pouvant rendre un APK malveillant. D'autres approches sont basées sur la génération des graphes d'execution de l'APK en éssayant de représenter tous les chemins que forment les appels de méthodes d'un APK lors de son éxecution. Vu que ces dernieres présentent des limites, nous avons combiné ces différentes approches en nous basant sur l'analyse du code source, l'extraction des permissions et intentions et aussi la génération du graphe d'execution de l'APK.
D'apres les expérimentations que nous avons effectué, il nous a fallu éliminer les mots clés java, les ponctuations et les commentaires dans le code source ; de plus, vu que les graphes obetenus sont tres denses en terme de noeud, nous avons essayé d'éliminer les noeuds intermediaire dans chaque chemin afin de capturer uniquement les points d'entrées et sorties dans l'APK. La transformation de ces éléments en vecteur numerique nous a permi d'entrainer un réseau de neurones convolutifs. Nous avons pu constater que ce modele présente des résultats tres satisfaisantes **99.84**% sur les données d'entrainements et **96.64**% sur les données de test en utilisant 1663 APKs dans le jeux de données comparé à celle de l'état de l'art (CICAndMal2017) avec **95.5**% sur les données de test.
En perspectives, afin de capturer exactement les portions ou blocs de codes malveillants pour la detection au niveau de l'analyse du code source, nous comptons ne plus decouper le code source en bloc de paragraphes mais d'utiliser les expressions régulieres java afin d'extraire les blocs de méthodes définies dans ce dernier et les transformer en numérique.
Nous n'avons pas pu prendre en compte tous les APKs de notre jeux de donnéés du fait que l'algorithme qui permet d'éliminer les noeuds intermédiaires et celui du node2vec prenait assez de temps sur notre petite machine (8Go de RAM, corei3). Donc les APKs de plus de 55000 (cinquante cinq milles) noeuds ne sont pas pris en compte pour l'entrainement de ce modele (ils représentent pret de 20% de notre jeux de données). Nous comptons aussi modifier l'algorithme **Node2vec** afin de reduire son temps d'exécussion sur les graphes à grand volume de noeuds.

Bibliographie

[1] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin : Effective and explainable detection of android malware in your pocket. 02 2014.

[2] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout : Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 217–228, New York, NY, USA, 2012. Association for Computing Machinery.

[3] Chien-Wei Chang, Chun-Yu Lin, Chung-Ta King, Yi-Fan Chung, and Shau-Yin Tseng. Implementation of jvm tool interface on dalvik virtual machine. pages 143 – 146, 05 2010.

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

[5] Kameni Florentin Flambeau Jiechieu and Norbert Tsopze. Skills prediction based on multi-label resume classification using cnn with model predictions explanation. *Neural Computing and Applications*, pages 1–19, 2020.

[6] J. Jung, H. Kim, D. Shin, M. Lee, H. Lee, S. Cho, and K. Suh. Android malware detection based on useful api calls and machine learning. In *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pages 175–178, 2018.

[7] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. Maldozer : Automatic framework for android malware detection using deep learning. *Digital Investigation*, 24 :S48 – S59, 2018.

[8] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.

[9] Roni Mateless, Daniel Rejabek, Oded Margalit, and Robert Moskovitch. Decompiled apk based malicious code classification. *Future Gener. Comput. Syst.*, 110 :135–147, 2020.

[10] Abdelmonim Naway and Yuancheng LI. Using deep neural network for android malware detection, 2019.

[11] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Abdurrahman Pektas and T. Acarman. Deep learning for effective android malware detection using api call graph embeddings. *Soft Computing*, 24 :1027–1043, 2020.

[13] Waqar Rashid. *Automatic Android Malware Analysis*. PhD thesis, 07 2018.

[14] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Bringas, and Gonzalo Alvarez. *PUMA : Permission Usage to Detect Malware in Android*, volume 189, pages 289–298. 01 2013.

[15] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici. Mobile malware detection through analysis of deviations in application network behavior. *Computers Security*, 43 :1 – 18, 2014.

[16] L. Taheri, A. F. A. Kadir, and A. H. Lashkari. Extensible android malware detection and family classification using network-flows and api-calls. In *2019 International Carnahan Conference on Security Technology (ICCST)*, pages 1–8, 2019.

[17] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. Droidmat : Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*, pages 62–69, 2012.

[18] Xiaolu Zhang, Frank Breitinger, and Ibrahim Baggili. Rapid android parser for investigating dex files (rapid). *Digital Investigation*, 17 :28 – 39, 06 2016.