TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

API	Application Programming Interface
CPU	Central Processing Unit (processeur)
GID	Group Identifier
ID	Identifiant
iOS	iPhone Operating System
IPC	Inter-Process Communication
OS	Operating System (système d'exploitation)
RAM	Random Access Memory
SMS	Short Message Service
UID	Seconde
RAM	User Identifier

LISTE DES SYMBOLES ET UNITÉS DE MESURE

O	octet
Mb	Megabyte, Mo (Mégaoctet)
Gb	Gigabyte, Go (Gigaoctet)
Kb	Kilobyte, Ko (Kilooctet)
KHz	KiloHertz
MHZ	Mégahertz, unité de mesure de fréquence du Système international (SI)
Ms	Milliseconde
S	Seconde

INTRODUCTION

Mise en contexte

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 migrer de plus en plus vers le développement relié à ces plateformes. Cette migration inonde le marché d'un nombre incalculable d'applications mobiles, qui apportent de nombreux avantages aux utilisateurs.

Cette forte croissance des applications mobiles est permise non seulement grâce à l'augmentation des ventes des Smartphones ces cinq dernières années (voir figure 0.1), mais aussi grâce à la facilité d'accès aux applications par le biais des plateformes de téléchargement d'applications.

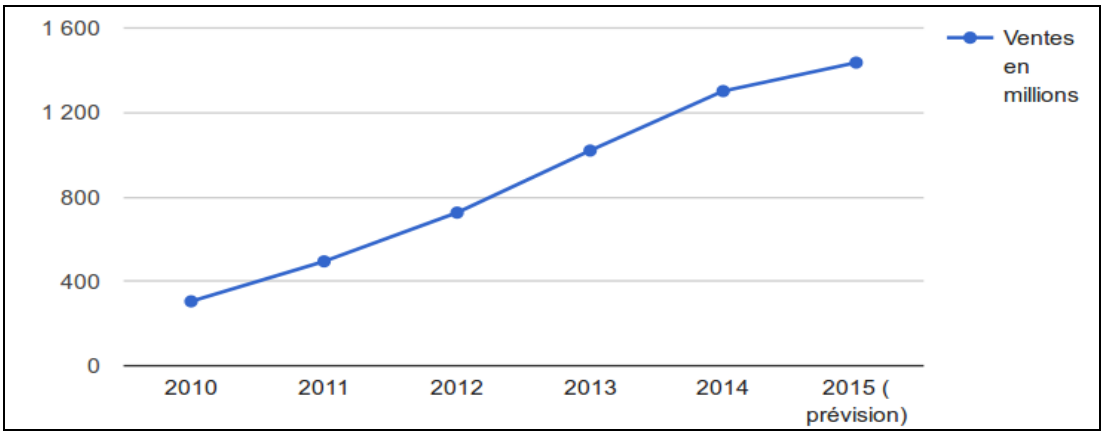

Figure 0.1 Livraisons mondiales des Smartphones.
Tirée de (ZDNet, 2016)

Les plateformes de téléchargement d'applications permettent d'accéder aux applications mobiles ainsi que de gérer leurs visibilités aux utilisateurs. Il en existe plusieurs, mais celle qui se démarque le plus est la plateforme d'Android.

Android est un système d'exploitation open source pour appareils mobiles, basé sur le noyau Linux, racheté et géré par Google. Il est leader dans le segment des Smartphones depuis la fin de l'année 2010, et le numéro un des ventes dans le segment des tablettes depuis 2012 (ZDNet, 2016).

La figure 0.2 montre que Android a su s'imposer aux ventes par rapport aux autres systèmes d'exploitation mobiles, où il possède plus de 70% du marché. Le facteur ayant permis cette grosse part du marché à Android, est le nombre important d'appareils mobiles sur lesquels il fonctionne contrairement à son rival iPhone OS, qui lui, exige un périphérique spécifique.

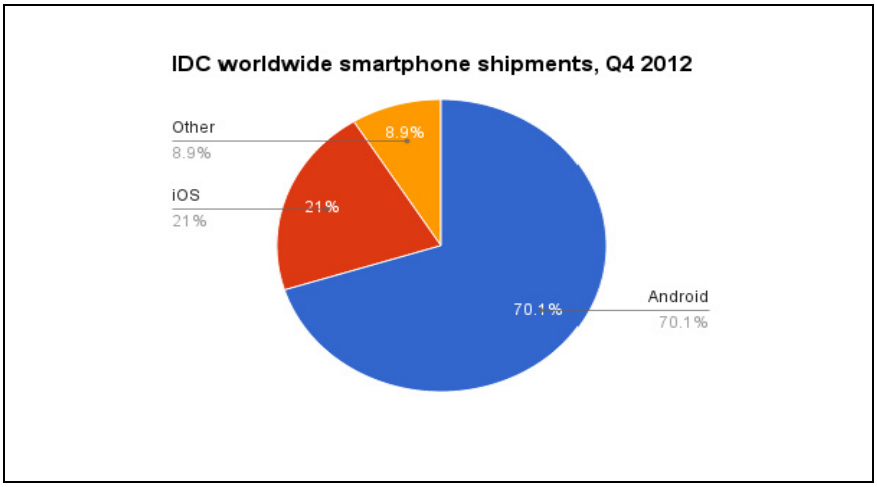

Figure 0.2 Le marché mondial des Smartphones.
Tirée de (Stark, 2013)

Cependant, cette popularité de la plateforme Android 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 Android, affecte non seulement la vie privée d'un très grand nombre d'utilisateurs, mais aussi l'économie des organisations intégrant Android dans leurs infrastructures TI. Par conséquent, ceci rend la sécurité de cette plateforme une des préoccupations les plus importantes et prioritaires.

Dans ce présent travail, une solution visant à modifier le comportement des applications Android est proposée afin de fournir un mécanisme de contrôle permettant l'application d'un ensemble de politiques de sécurité. Ces dernières sont introduites par l'utilisateur et permettent une exécution sécuritaire de l'ensemble des applications.

Problématique

Bien que les applications Android offrent de nombreux avantages ainsi qu'un grand confort cognitif à notre quotidien, ces dernières présentent diverses vulnérabilités dont il est essentiel de se protéger. En effet, Android est la plateforme qui subit le plus d'attaques; 95% des attaques contre les plateformes mobiles ciblent la plateforme Android (Rahul, 2013). Des applications malicieuses peuvent donc pénétrer dans le périphérique de l'utilisateur, et réaliser plusieurs tâches malveillantes, qui vont de la collecte des données personnelles des utilisateurs, jusqu'à l'endommagement du périphérique en question.

Plusieurs solutions visant à protéger l'utilisateur contre les différentes attaques ont été élaborées; de celles qui identifient les portions de code malveillant dans les applications, jusqu'à celles qui détectent le comportement étrange des applications. Une première catégorie de solution similaire à celle déployée sur les ordinateurs de bureaux a donc été commercialisée par les sociétés de sécurité « Kaspersky », « Symantec » et « Mcafee ». Ces solutions sont basées sur la détection statique des signatures, qui sont des portions de code répertoriées dans une base de données. Plusieurs chercheurs se sont aussi intéressés à ce mode de détection de malwares, comme (Feng et al., 2014), (Yerima et al., 2013), etc. Mais cette technique est loin d'être parfaite, puisqu'elle comporte plusieurs inconvénients, comme le fait qu'elle ne puisse détecter que les malwares répertoriés. De plus, les développeurs des applications malveillantes utilisent plusieurs techniques permettant de cacher les signatures, on cite à titre d'exemple, le repackaging des applications, l'obfuscation du code, etc. D'où l'impossibilité de lutter contre les vulnérabilités du jour zéro.

Une deuxième catégorie de solutions basée sur la détection d'anomalies est apparue, cette technique compare le comportement de l'application en question, à un modèle qui correspond à une activité normale du système (Dini et al., 2012) (Sanz et al., 2014). Cette solution lutte

contre les vulnérabilités du jour zéro. Cependant, la standardisation du comportement normal est une tâche complexe. De plus, les algorithmes utilisés pour augmenter le taux de détection sont souvent complexes.

Ces deux familles de solutions peuvent apporter certains avantages, mais elles sont loin d'être parfaites. Car le plus grand problème des solutions de sécurité apportées aux appareils mobiles découle des contraintes de ressources caractérisant les périphériques, à savoir, le processeur, la batterie, la mémoire, etc.

Malheureusement, il n'existe pas de solution parfaite, puisque d'un côté le risque zéro n'existe pas, et d'un autre côté, le terme "applications malveillantes" est trop vaste, puisque certaines applications ne sont dangereuses que dans certains contextes définis.

La présente étude s'intéresse à une catégorie d'applications qui ne sont pas forcément malicieuses dans certains cas, mais qui peuvent être extrêmement dangereuses dans d'autres cas. 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 anti 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 (El-Serngawy et Talhi, 2015). Plusieurs scénarios encore plus complexes que ce petit exemple peuvent se présenter, et peuvent affecter la vie privée de l'utilisateur.

Cette étude vise à élaborer une solution permettant non seulement le contrôle des appels aux méthodes API par un utilisateur d'un Smartphone mais aussi de fournir un moyen de contrôle des périphériques au sein des entreprises. Si, par exemple, une société fournit des Smartphones à ses employés, un administrateur peut déployer la solution proposée dans la présente étude afin d'assurer un contrôle sur l'ensemble des appareils et d'appliquer certaines politiques de sécurité propres à l'entreprise, par exemple: l'interdiction de l'utilisation de la caméra au sein de l'entreprise, où lors des réunions, etc.

La solution proposée, prend en compte cette collaboration entre les applications afin de permettre à l'utilisateur d'appliquer un ensemble de politiques de sécurité visant à interdire toute tâche malveillante réalisée par une, deux ou plusieurs applications et ce, tout en trouvant le meilleur compromis entre la sécurité des applications et la consommation des ressources des périphériques.

Ce travail vise donc à réaliser une solution centralisée, permettant le contrôle des applications installées sur le périphérique, tout en interceptant les appels aux méthodes API, puisque Android est basé sur l'appel à ces dernières, afin de réaliser les différentes tâches (envoyer SMS, captures d'écran, ce ordonnés GPS …). Le but de cette étude est donc le contrôle de ces appels afin d'appliquer les décisions de l'utilisateur représentées par un ensemble de politiques de sécurité.

Objectifs et réalisations

L'objectif principal de la présente étude vise à élaborer une solution permettant la sécurisation des applications Android et plus précisément le contrôle des appels aux méthodes API. Ceux-ci étant responsables de la plupart des actions réalisées par les applications Android. Une solution centralisée permettant le contrôle des applications est proposée. Celle-ci assure l'exécution des décisions de sécurité prises par l'utilisateur tout en essayant de trouver le meilleur compromis entre la sécurité et la consommation des ressources limitées sous la plateforme Android. Plusieurs sous objectifs sont traités par notre présente étude, à savoir :

- Mettre en place un contrôleur d'applications centralisé, permettant la gestion des appels aux méthodes API effectués par l'ensemble des applications installées dans le périphérique.
- Mettre en place un Framework de réécriture d'applications permettant à ces dernières de communiquer avec le contrôleur d’applications.

- Étudier les différentes possibilités de modification du code byte Dalvik [1] et implémenter une solution permettant l'injection de nouvelles portions de code (classes, méthodes, etc.) dans une application APK[2] (Android Application Package); i.e., une application mobile distribuée et installée sur le système Android.
- Étudier les moyens permettant la décompilation et la recompilation des fichiers APK.
- Mettre en place un système d'échange de données entre le contrôleur d'applications et les applications réécrites.
- Mettre en place un système de spécification de politiques de sécurité basée sur un ensemble de conditions.

Pour réaliser cet ensemble d'objectifs, un intérêt est, tout d'abord, porté sur la structure des fichiers API ainsi qu'aux possibilités de modification du code byte des applications, puisque l'étude est basée sur les applications compilées (APK) et non pas sur le code source. Cette étape mène à comprendre et à maîtriser le langage de programmation intermédiaire Smali (smali/baksmali, 2004), basé sur le code byte.

Mais avant d'entamer ces modifications, nous nous intéressons d'abord, à la décompilation et la recompilation des fichiers APK, afin d'obtenir le code byte Smali, acceptant les modifications. Pour ce faire, la présente étude s'intéresse à l'outil APKTOOL (Tumbleson, 2015), permettant ce « BackSmaling ».

Le présent travail est basé sur l'interception des appels aux méthodes API, et pour implémenter ce mécanisme, l'étude s'intéresse auxdites méthodes, à leur fonctionnement ainsi qu'à la façon selon laquelle elles sont utilisées dans les applications Android. Par la suite, le top 20 des méthodes API les plus dangereuses est considéré.

Après avoir maîtrisé les modifications des fichiers APK, un Framework de modifications automatique des applications Android est mis en place. Ce système permet la décompilation

---

[1] Dalvik est une machine virtuelle destinée aux téléphones mobiles et tablettes tactiles, qui est incorporée dans le système d'exploitation Android.
[2] Un APK est une collection de fichiers compressés pour le système d'exploitation Android, qui constitue un paquet.

des fichiers APK, comme il permet d'apporter les modifications nécessaires aux bons endroits ainsi qu'à compiler le nouveau fichier APK qui sera apte à communiquer avec le contrôleur d'applications.

Dans cette étude, un intérêt est également porté aux moyens de communication entre les applications Android, afin d'établir un système de communication entre les applications réécrites et le contrôleur d'applications développé. Pour cela, les Intents [3] (Chilowicz, 2012) sont utilisés.

L'étude met en œuvre le contrôleur d'applications. Celui-ci représente une solution centralisée, permettant le contrôle des applications installées sur le périphérique, tout en consultant un ensemble de politiques de sécurité introduites par l'utilisateur.

Un système permettant à l'utilisateur d'introduire facilement les politiques de sécurité souhaitées est mis en place dans la présente étude. Ce système permet une spécification de politiques basées sur plusieurs conditions, que l'utilisateur peut définir et combiner.

- Les principales contributions du présent travail sont : Une solution centralisée permettant le contrôle des appels aux méthodes API sollicitées par l'ensemble des applications installées sur le périphérique.
- Une solution permettant de lutter contre la collaboration des applications pour la création d'un contexte malicieux.
- Un Framework de modification automatique des APK permettant leur renfoncement, en ajoutant des portions de code byte. Implémenté en deux variantes, une première fonctionnant sur un serveur distant, et une deuxième destinée aux périphériques mobiles.
- Une manière simple et efficace d'introduction des politiques de sécurité.

[3] Mécanisme de communication entre processus dans le système android.

Méthodologie suivie

La méthodologie de recherche adoptée dans cette étude est représentée dans la figure 0.3. Cette dernière montre que les vulnérabilités de la plateforme Android ainsi que les mécanismes de sécurité élaborés sont étudiés en premier lieu. Cette étude a permis de définir une problématique de recherche qui est l'impact des appels des méthodes API non sécurisé sur la plateforme Android.

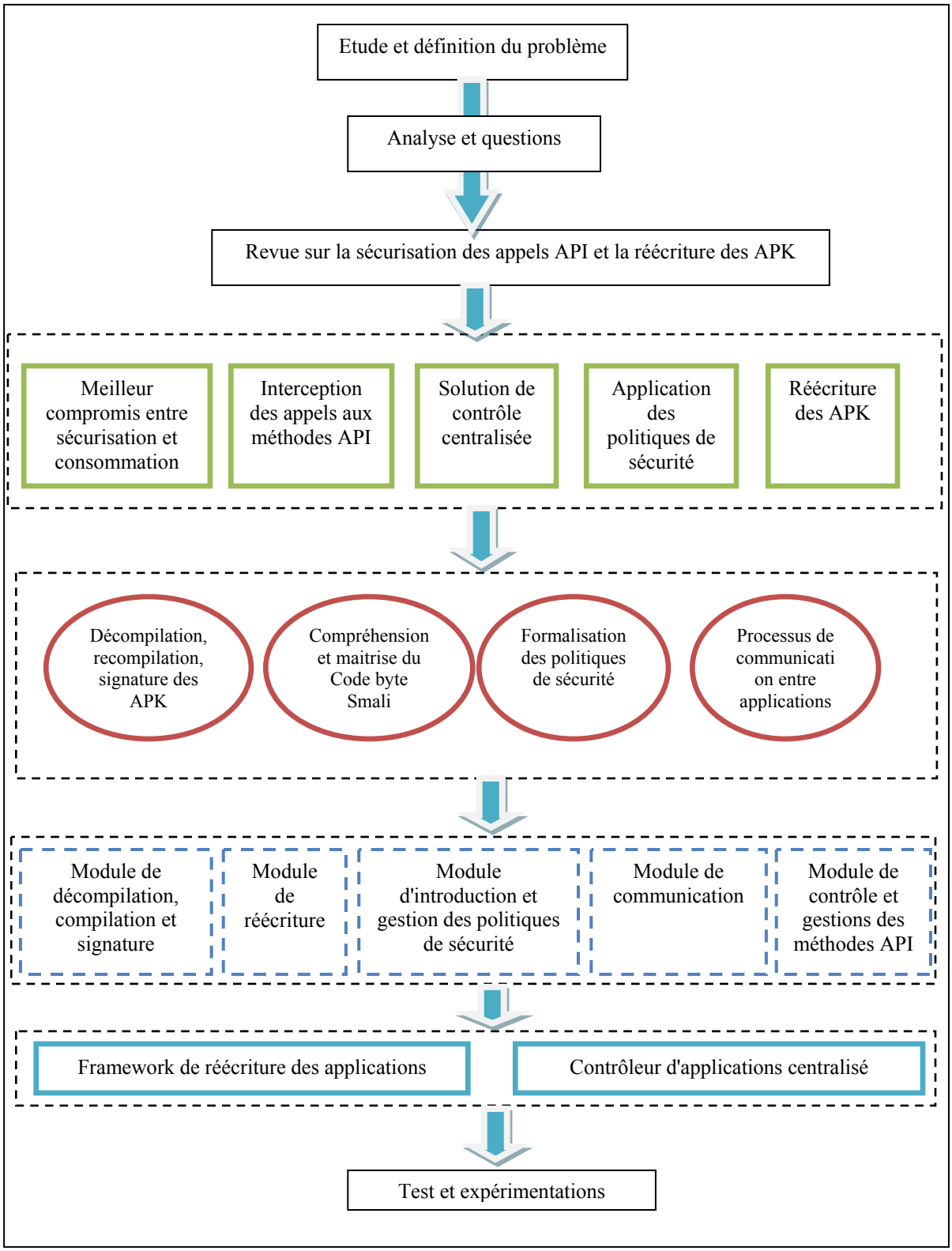

Figure 0.3 Méthodologie de notre étude

Après avoir défini la problématique, une revue de littérature est présentée dans laquelle les techniques et les mécanismes de sécurisation des appels aux méthodes API sont identifiés. Cette revue de littérature a permis de porter un intérêt à la réécriture des applications et la modification du byte code.

Par la suite, les caractéristiques de la solution développée sont identifiées. Celles-ci sont représentées dans la figure 0.3 par des cercles rouges. Il est à noter que ces caractéristiques représentent le cœur du présent travail, puisque la solution proposée se base sur la décompilation des applications Android, la modification du code byte Smali, ainsi que la recompilation du code byte pour la génération de la nouvelle application. Cette dernière va par la suite communiquer avec le contrôleur d'applications grâce aux intents. Le contrôleur à son tour consulte les politiques de sécurité introduites par l'utilisateur afin de prendre la décision adéquate par rapport à l'exécution de la méthode API sollicitée.

En se basant sur les caractéristiques déjà identifiées, plusieurs modules, représentés dans la figure 0.3 par des rectangles bleus, ont été modélisés. Ceux-ci permettent la réalisation de l'ensemble des tâches proposées par la solution développée dans cette étude.

Ensuite, les modules préalablement modélisés sont par la suite implémentés en deux blocs. Le premier étant le Framework de réécriture automatique des APK, tandis que le deuxième étant le contrôleur d'applications, permettant le contrôle des méthodes API sollicitées par l'ensemble des applications installées sur le périphérique et ce, en se basant sur les politiques de sécurité introduites par l'utilisateur au sein du contrôleur.

Il est à noter que des tests réalisés sur plusieurs axes de la solution développée ont été effectués par la suite. Ces tests visent à évaluer le fonctionnement de notre solution, ainsi que sa consommation de ressources.

Organisation du rapport

L'organisation du rapport est comme suit : les concepts de base de la plateforme Android sont tout d'abord présentés. Ce chapitre présente un survol sur l'architecture, les mécanismes de base, et les failles de sécurité de cette plateforme.

Par la suite, dans le chapitre 2, différentes solutions permettant la sécurisation de la plateforme Android sont étudiées et analysées. Notre analyse comporte des solutions appartenant à deux axes de sécurisation de la plateforme Android, à savoir, la modification du Framework, et la modification des applications.

Le chapitre 3 illustre l'architecture de la solution proposée. Un Framework de réécriture d'application, ainsi qu'un contrôleur d'applications sont donc présentés.

Le chapitre 4 contient les ensembles de données et les paramètres d'évaluation utilisés dans nos expériences. Ce chapitre comporte les différents tests réalisés sur l'ensemble de la solution proposée (Framework, et contrôleur), et illustre les résultats obtenus.

Une conclusion ainsi qu’une discussion des travaux futurs sont présentés au dernier chapitre.

CHAPITRE 1

SÉCURITÉ DU SYSTÈME ANDROID

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 migrer de plus en plus vers le développement relié à ces plateformes. Cette migration inonde le marché d'un nombre incalculable d'applications mobiles, qui apportent de nombreux avantages aux utilisateurs, mais qui peuvent représenter des risques importants.

Nous introduisons dans ce chapitre la plateforme Android, où nous nous intéressons aux bases de ce système, ainsi qu'à ces composants. Nous présentons par la suite le modèle de sécurité déployé par Android, et un aperçu de ses mécanismes de sécurité. Nous nous intéressons ensuite aux vulnérabilités affectant le système Android, et aux façons dont elles sont employées. Et nous terminons par la présentation d'un certain nombre de contre-mesures déployées afin de limiter les attaques.

1.2 Le système d'exploitation Android

Android est un système d'exploitation Open source pour terminaux mobiles, il a été conçu et développé par la start-up Android, et racheté par la suite par la société Google qui l'a annoncé officiellement le 15 novembre 2007 (Gannes, 2007). Android est fondé sur un noyau modifié du système d'exploitation Linux, il supporte diverses plateformes embarquées telles que ARM, x86, MIPS, etc. Son architecture est conçue d'une manière permettant l'intégration des applications Google comme Gmail, Google agenda, YouTube ou encore Google maps, et a été proposé à tous les constructeurs des téléphones mobiles, tout en autorisant les modifications par ces fabricants (Cooper, Shahriar et Haddad, 2014).

Comme le représente la figure 1.1, l'architecture d'Android est constituée de quatre couches principales, qui sont représentées du plus bas niveau, qui est la couche la plus éloignée de

l'utilisateur (représenté en rouge dans la figure 1.1), jusqu'au niveau le plus proche de l'utilisateur (Crussell, Gibler et Chen, 2012).

Puisque Android est basée sur le système d'exploitation Linux, le noyau de ce dernier représente le niveau le plus bas de Android. Cette couche assure la liaison entre le matériel et le logiciel, et elle fournit aussi les services les plus fondamentaux tels que la sécurité, la gestion des processus, la gestion de la mémoire, les fonctionnalités des communications réseau, etc. Android ne s'occupe donc pas de l'interaction avec le matériel, c'est la raison qui fait que Android soit compatible avec la plupart des périphériques mobiles, puisque c'est la tâche du constructeur de spécifier ses composants matériels dans le noyau (BOUGARMOUD, 2013).

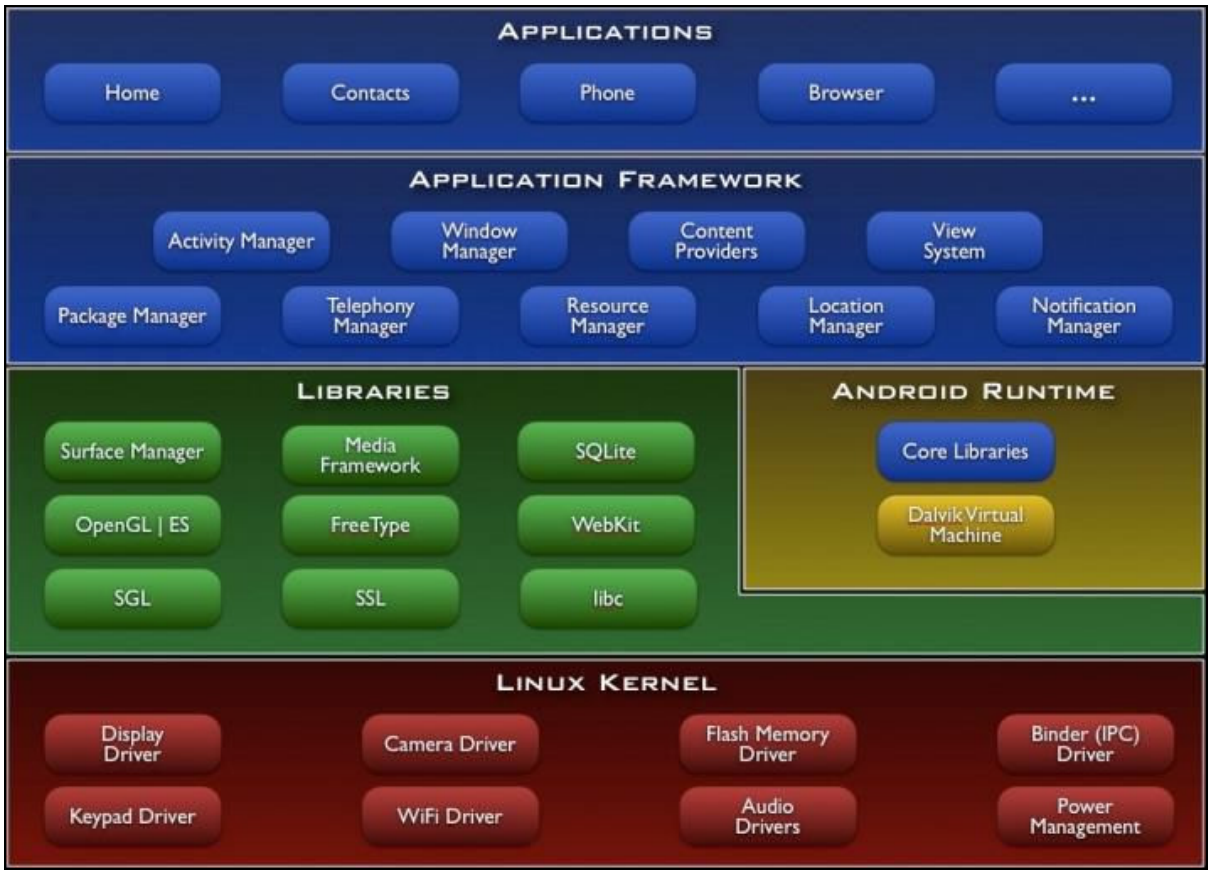

Figure 1.1 Architecture du système d'exploitation Android.
Tirée de (Crussell, Gibler et Chen, 2012)

Un moteur d'exécution est un programme qui assure le fonctionnement d'autres programmes, ce mécanisme est intégré dans l'architecture d'Android plus précisément la couche au-dessus du noyau Linux. Cette couche contient des bibliothèques natives écrites en C/C++, des bibliothèques Java, ainsi que des bibliothèques spécifiques à Android et la machine Virtuelle Dalvik. Ces bibliothèques servent essentiellement à fournir des fonctionnalités aux couches applicatives d'Android (Shabtai, 2009).

La couche Framework offre aux développeurs des services de haut niveau, qui sont appelés à partir des applications sous format java. Cette couche offre donc un ensemble d'API (Application Programming Interface) qui permet la création des applications riches et innovantes. Les développeurs peuvent donc à travers ces API, exécuter des services d'arrière-plan, ajouter des notifications, accéder aux services de localisation, etc.

La couche la plus haute de l'architecture Android, est la couche applications. C'est la seule couche visible par l'utilisateur du périphérique. Elle se concrétise dans un ensemble d'applications installées sur le périphérique, qui sont développées à l'aide du langage Java, et compilées en des fichiers APK (Crussell, Gibler et Chen, 2012)

1.3 La machine virtuelle Dalvik

Dalvik est une machine virtuelle incorporée dans le système d'exploitation Android. Elle permet l'exécution de plusieurs applications sur un appareil de faible capacité en mémoire, et en puissance de calcul. Contrairement à la machine virtuelle classique de la technologie java, qui est basée sur la pile, la machine Dalvik est une machine à registres qui nécessite moins d'instructions pour effectuer les mêmes opérations qu'une machine à pile. En raison de cette différence, les fichiers de byte code Java ne peuvent pas être exécutés tels qu'il sont par Dalvik, ils seront tout d'abord transformés et consolidés dans un fichier .dex (Chien-Wei et al., 2010).

La Figure 1.2 montre la différence entre le système de compilation Android par rapport aux applications Java classiques. Le code source des applications Android écrites en Java, est compilé en un byte code java classique (.class), puis recompilé par la suite en un byte code

compréhensible par la machine virtuelle Dalvik. Les fichiers .class sont donc convertis en un fichier .Dex (ScriptTol, 2015).

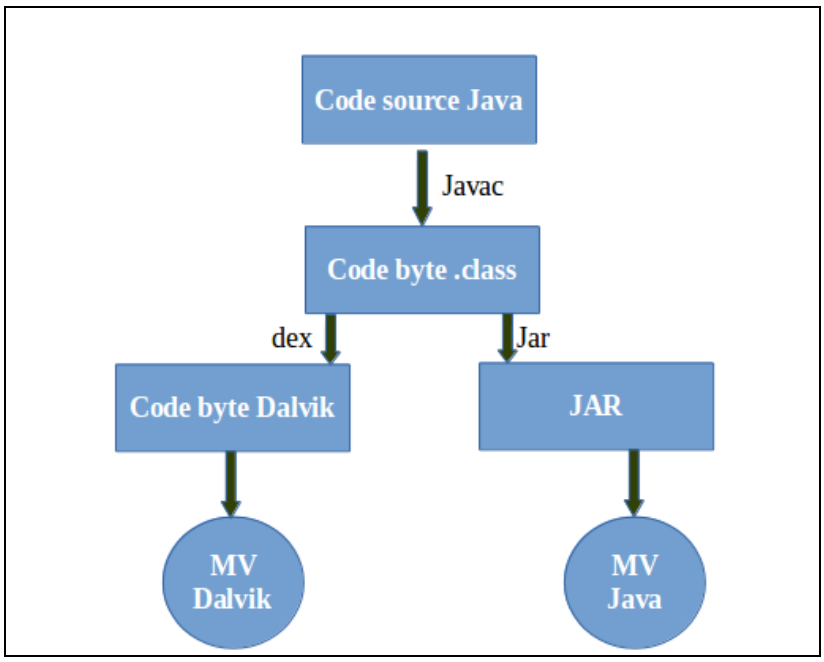

Figure 1.2 Système de compilation Dalvik

1.4 Les fichiers APK

Un fichier APK est un format de fichier utilisé pour distribuer et installer des applications, au sein du système d'exploitation Android. Autrement dit un APK est l'équivalent d'un fichier JAR conçu spécialement afin de tourner sous Android.

La figure 1.3 montre un fichier APK et ces composants. Le fichier APK contient donc : 1) un fichier .DEX qui est le byte code Dalvik de l'application; 2) un fichier resources.arsc qui contient les ressources compilées (layouts ,xml etc.); 3) un fichier de ressources non compilées (les images, les strings, etc.); et 4) un fichier AndroidManifest.xml qui contient pour sa part le nom de package, la version actuelle de l'application, ainsi que diverses autres informations, qui sont nécessaires à l'exécution de l'application sous Android (permissions, activités, intents etc.) (Garin, 2009).

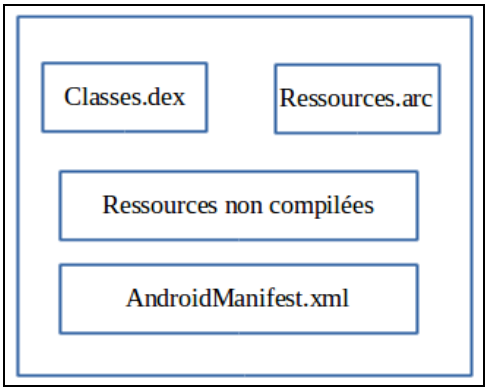

Figure 1.3 Composition d'un fichier APK

1.5 Communication inter applications (Intents)

La communication entre applications au sein du système est une notion importante, elle permet le partage des données entre les applications, l'envoi des messages entre applications, etc. Le système d'exploitation Android offre un moyen de communication puissant entre applications. Cette communication est assurée grâce au mécanisme des Intents (Guignard, 2010).

Les Intents sont basés sur le principe des Sandboxing, qui consiste à séparer presque totalement les applications entre elles. En d'autres termes, les Intents sont un ensemble de données qui peuvent être passées à un autre composant applicatif, faisant partie de la même application ou non (Guignard, 2010).

Les Intents peuvent être envoyés de deux principales façons, une première qui est explicite, et qui permet le lancement d'une classe précise. Et une deuxième dite implicite, qui permet le lancement d'une requête pour une action, sans la nécessité de préciser la classe qui s'occupe du traitement.

La figure 1.4 montre la composition des Intents, où le champ composant est utilisé dans le cas des Intents explicites, et qui identifie le destinataire de l'intent. Le champ action représente l'action attendue par le destinataire. Des informations supplémentaires sur l'action peuvent êtres apportées, et qui sont définies dans le champ catégorie. Les Intents prennent un

ensemble de données sur lesquelles le destinataire effectuera l'action, le type de ces données est inscrit dans le champ Type. Le champ extras sert à ajouter du contenu à l'intent, et le champ flags permet la modification du comportement de l'intent (Google, 2015).

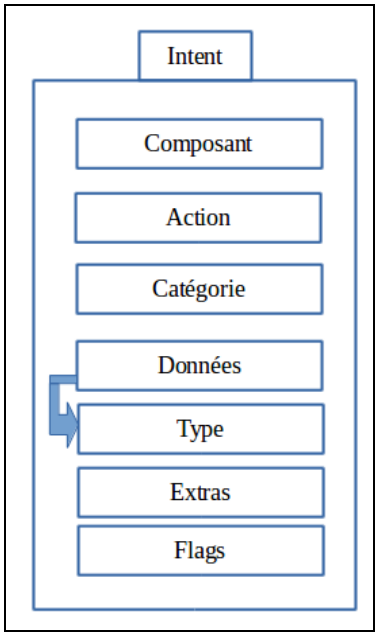

Figure 1.4 Composition d'un Intent

1.6 Modifications des APK (reverse engineering)

Il existe plusieurs outils permettant la récupération du code source java à partir du byte code. Mais comme nous l'avons déjà mentionné, une grande différence existe entre la machine virtuelle Java et la machine virtuelle dalvik, où le plus important facteur est l'utilisation des registres. Ceci pose un problème lors de la récupération du code source, puisque certaines informations sont perdues lors de la conversion, ce qui empêche le code source récupéré de s'exécuter. Le moyen efficace afin d'effectuer des modifications aux fichiers Apk tout en gardant la possibilité de la recompilation, est donc le retour vers un byte code intermédiaire appelé Smali.

1.7 Le langage Smali

Smali est un langage d'assemblage pour la machine virtuelle Dalvik, basé sur la syntaxe Jasmin's/dedexer's qui est un assembleur de byte code Java. Smali s'appuie sur les descriptions ASCII des classes de la machine virtuelle Dalvik. Le langage Smali est une représentation plus simplifiée de l'ensemble d'instructions de la machine dalvik. Cette représentation accepte les modifications. Il existe plusieurs compilateurs/décompilateurs qui permettent la conversion du byte code Dalvik (.dex) en byte code Smali et vice versa. On peut citer pour exemple « Apktool » (Dhanesh, 2015).

1.8 Modèle de sécurité d'Android

Le modèle de sécurité Android est conçu sur deux différents niveaux, à savoir la sécurité au niveau noyau Linux, et au niveau applicatif.

1.8.1 Sécurité au niveau noyau Linux

1.8.1.1 Cloisonnement

Dans le modèle de sécurité Linux, à chaque création d'un utilisateur, un nouveau UserID (UID) est créé. À sa création, une ressource appartient à l'utilisateur qui l'a créée, celui-ci peut changer les droits d'accès par la suite, par l'affectation des différents niveaux d'accès proposés, à savoir lecture (R), écriture (W) et exécution (X). Plusieurs utilisateurs peuvent faire partie d'un même groupe (GID), et des droits d'accès doivent être affectés au groupe (B.V, 2013).

Android se base sur ce concept afin de définir son modèle de sécurité au niveau noyau. À l'installation d'une nouvelle application, un nouveau UserID est donc créé, et va être attaché à l'application en question, et accompagne l'application durant sa durée de vie. Par conséquent seul cet utilisateur (l'application en question) peut accéder à son ensemble de fichiers (image, base de données, etc.). Cette architecture se base sur le principe de Sandbox, où chaque application s'exécute en s'isolant des autres applications. L'ensemble des applications signées

donc par le même certificat s'exécute sous un même groupe d'utilisateur (GID) (Garin, 2009).

La Configuration par défaut de cette architecture empêche donc l'exécution de deux applications dans un même processus. Mais des exceptions existent, si deux applications désirent donc partager le même UID, ces dernières doivent le demander grâce à la fonctionnalité sharedUserID. Ces applications partageront donc le même processus, ainsi que les mêmes permissions.

1.8.1.2 La gestion de la mémoire

Le noyau Linux permet la gestion de l'accès à la mémoire, en utilisant le concept MMU (Memory Management Unit), qui permet la séparation de la mémoire utilisée par les processus. Chaque processus est donc affecté à un espace mémoire différent. Ce mécanisme empêche les applications d'accéder à des données non autorisées (Silicium, 2014).

1.8.2 Sécurité au niveau applicatif

Le modèle de sécurité d'Android s'étend aussi sur la couche applications, par l'implémentation de quelques mécanismes offrant une couche de sécurité supplémentaire.

1.8.2.1 L'utilisation d'un langage fortement typé

Les applications Android sont développées grâce au langage Java pour lequel la sécurité des applications a toujours été une importante préoccupation. Les spécifications du langage offrent donc de fonctionnalités permettant le renforcement du code. L'utilisation du réseau est assurée grâce à l'API JSSE, qui fonctionne en utilisant des protocoles sécuritaires, à savoir SSL ou TLS. Les technologies cryptographiques sont disponibles grâce à l'API JCA/JCE. De plus, l'utilisation d'un langage fortement typé permet d'éviter des erreurs provocant des failles de débordements « Buffers overflows » (Jean-Michel, 2014).

1.8.2.2 Signature numérique

Afin que Android puisse installer une application, cette dernière doit être obligatoirement signée par une clé privée associée à un certificat numérique. Cependant, les développeurs peuvent utiliser des certificats autosignés puisqu'aucune autorité de certification n'est requise. Ces certificats sont utilisés par Android afin d'identifier l'origine de l'application, garantir l'intégrité des applications, garantir la sécurité lors des mises à jour d'applications, et autoriser certaines interactions entre les applications signées par les mêmes certificats (Ruff, 2014).

1.8.2.3 Les permissions

Android propose un système déclaratif des droits d'accès. Les applications contiennent donc un fichier nommé AndroidManifest.xml qui décrit les permissions maximales requises par l'application en question. Plusieurs fonctionnalités sont donc protégées par des autorisations, telles que l'envoi des SMS, l'accès à internet, effectuer un appel téléphonique, accéder au GPS, etc. L'utilisateur est donc libre d'accepter ou de refuser les droits d'accès associés à l'application (Moulu, 2015).

Android possède quatre catégories de permissions, à savoir, normal, dangereux, signature, et signatureOrSysteme. Les permissions de type normal sont accordées automatiquement à l'application, sans demander l'approbation de l'utilisateur. Par contre celles du type dangereux nécessitent l'accord de l'utilisateur lors de l'installation de l'application (Ruff, 2014).

Ce système de permission s'appuie sur un moniteur de référence assurant le contrôle d'accès obligatoire MAC (Mandotory Access Control) sur l'ensemble des communications inter-composants ICC (Inter-Component Communication) (Moulu, 2015).

1.8.2.4 Déverrouillage par pattern

Android propose un système de déverrouillage par pattern, qui est lancé après chaque mise en veille du téléphone. Ce système se base sur un schéma de sécurité, composé d'une matrice

de neuf points. L'utilisateur doit donc relier au moins quatre des points en suivant le modèle qu'il a préalablement établi.

1.8.2.5 La révocation

Le nombre d'applications malveillantes est en croissance flagrant. Google s'adapte avec un mécanisme de sécurité sur Android : la technique « Kill switch » est utilisée et elle permet non seulement l'élimination des applications malveillantes du store, mais aussi de supprimer à distance toutes les instances de ces applications installées sur les téléphones et identifiées par leurs certificats (Ruff, 2014).

1.8.2.6 Services et Internet

Android minimise le risque d'attaques en exploitant les vulnérabilités de services en écoute sur le réseau. La configuration par défaut du système Android n'autorise donc aucun service à écouter les connexions entrantes, ce qui réduit le risque causé par ce type d'attaques.

1.9 Les vulnérabilités du système Android

Nous présentons dans cette section les principales failles et faiblesses du système d'exploitation Android.

1.9.1 Vulnérabilités dues au système de permissions

Une question essentielle à poser concernant le système de permissions d'Android est : est-ce que chaque permission correspond à un ensemble de fonctionnalités uniques? Ce problème a déjà été posé sous Linux, avec le modèle de capabilités. La réponse à cette question est : non, car prenant par exemple CAP_SYS_MODULE qui peut compromettre l'intégrité du noyau en chargeant des modules arbitraires (Ruff, 2014).

De plus, ce 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 (Guignard, 2010).

1.9.2 Vulnérabilités dues aux fabricants de matériels

Android est adopté par plusieurs fabricants de Smartphones ce qui contribue à sa popularité. Mais la concurrence fait rage entre les fabricants de matériels. Ces derniers doivent donc minimiser les coûts de développement, ainsi que le temps de fabrication, puisque la durée de vie commerciale d'un Smartphone est de l'ordre de 6 mois. Ces facteurs mènent les fabricants à bâcler certaines tâches. Les systèmes sont à peine testés, et les fonctions de débogage sont souvent activées. Tout cela nous pousse à nous poser des questions concernant les failles de sécurité, surtout que ces fabricants apportent des modifications au système Android, afin de créer des versions personnalisées à leurs matériels, qui ne sont pas forcément sécuritaires (Jiang et Zhou, 2013).

1.9.3 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 gens 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.

1.9.4 Vulnérabilités deux au code source

Le Système Android est développé en faisant recours à du code Open Source pouvant contenir des failles d'implémentation. De plus Android adopte certaines applications qui ne sont pas forcément reconnu par leur sécurité, comme Flash Player par exemple, ce qui mène Android à adopter les mêmes failles que ces applications. Plusieurs failles ont été découvertes dans le code source Android, on peut citer par exemple « Rage Against the

Cage », une faille liée au mauvais comportement du processus adb et qui permet essentiellement d'avoir l'accès au Root.

1.9.5 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 mini-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.

1.9.6 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.

1.10 Les attaques conte le système Android

La popularité d'Android le rend une cible de plusieurs types d'attaques. Diverses applications malveillantes existent, et tentent d'exploiter différentes vulnérabilités dans la plateforme Android, en partant de celles qui profitent du manque de vigilance des utilisateurs et cela en abusant des permissions, et en arrivant à des types plus sophistiqués qui exploitent des vulnérabilités applicatives, des vulnérabilités système, etc. Nous allons identifier dans cette partie les principales attaques menées contre le système Android (Hogarth, 2013).

1.10.1 Attaques par abus de permissions

Comme exliqué 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.

1.10.2 Attaques exploitant les vulnérabilités du noyau Linux

Comme expliqué précédemment, la plateforme Android est basée sur le noyau Linux, ce qui offre à Android le modèle performant de sécurité de Linux. Cependant ce noyau n'est pas infaillible non plus. Ce qui oblige le système Android à vivre avec les mêmes failles du noyau Linux.

Une catégorie d'applications malveillantes vise à exploiter les failles contenues dans le noyau Linux. L'ultime but de ces applications est d'élever les privilèges accordés à ces dernières, afin de contourner le système de SandBox implémenté sur Android. Cette manœuvre permet donc à ces applications malveillantes d'accéder à l'intégralité du système de fichiers, d'accéder aux API sensibles, etc. (Ruff, 2014).

RageAgainstTheCage est un exemple de vulnérabilité découverte dans le code source du noyau Linux, et qui permet d'avoir un accès administrateur (Root) sur l'appareil attaqué.

1.10.3 Attaques exploitant les vulnérabilités du Framework Android

Comme déjà mentionné auparavant, Le Framework Android est situé juste au-dessus du noyau Linux. Ce Framework souffre aussi d'un nombre de vulnérabilités, qui se présentent comme des erreurs de conception dans le code source du Framework. Des vulnérabilités peuvent se situer dans les modules cœur du Framework. Pour corriger ces vulnérabilités, les développeurs doivent donc revoir la conception de ces modules, ce qui peut affecter le fonctionnement du Framework en entier.

1.10.4 Attaques exploitant les vulnérabilités des applications

Le nombre de développeurs d'applications Android augmente, mais ces derniers ne possèdent pas forcément un bon niveau de programmation, 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.

1.10.5 Attaques par collaborations entre applications

Certaines applications d'apparence légitimes peuvent causer 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.

1.10.6 Attaques par espionnage des Intents

Les Intents sont un mécanisme qui permet l'échange de données et requêtes entre les applications. Elles peuvent donc contenir des informations sensibles, qui peuvent représenter des cibles à plusieurs types d'attaques.

1.10.6.1 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 manifest, 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).

1.10.6.2 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.

1.10.6.3 Détournement des services

Les intents servent aussi à établir des connexions entre des applications et des services externes. Si une application malveillante arrive donc à intercepter l'intent qui permet cette connexion, elle peut détourner la connexion vers un service malicieux, et ainsi exécuter plusieurs tâches assurées par ce service malveillant.

La différence entre les services et les activités, est que les services ne comportent pas d'interface utilisateur, ce qui rend la détection de l'attaque difficile, puisque l'utilisateur ne se rend pas compte des données circulées entre l'application et le service malveillant.

1.10.7 Attaques par botnets

Les périphériques tournant sous Android sont eux aussi cibles des botnets, qui sont un réseau de milliers à plusieurs millions d'ordinateurs compromis par un parasite de zombification. Le but ultime de ce concept est d'augmenter la puissance d'attaque contre une cible. La personne malveillante réalisant l'attaque infecte donc le maximum de périphériques afin d'exploiter la somme des puissances de calcul de ces périphériques appelés zombies. Cette puissance d'attaque sera donc louée à des commanditaires (cybercriminels, etc.) afin de réaliser des attaques de type Spam, Phishing, déni de service, etc.

1.10.8 Attaques par injection de code

Cette technique est utilisée par les développeurs des applications malveillantes afin de tromper les utilisateurs Android, et les logiciels anti malware. Ce concept se base essentiellement sur le téléchargement des applications qui semblent légitimes. Ces applications ne contiennent donc pas des portions de code malveillant à leur état initial. Mais, elles sont programmées afin de pouvoir télécharger un code natif, une fois que l'application est installée. De cette manière les anti malware ne détectent pas ces applications et les

considèrent en tant que légitimes. Mais une fois que le code natif est téléchargé, il sera exécuté afin de réaliser des tâches malveillantes.

Les programmeurs de ces applications malveillantes ne se basent pas juste sur le développement de nouvelles applications, mais utilisent une autre méthode plus astucieuse. Ils téléchargent donc les applications les plus populaires dans le store Android (Skype, Facebook, etc.), et appliquent de l'ingénierie inverse afin de récupérer le code source. Ce code source sera donc modifié afin de doter l'application de la capacité du téléchargement du code malveillant. L'application sera donc publiée sur le store en créant un clone malveillant. Les clones sont en augmentation flagrante, et représentent un quart de l'ensemble d'applications du store Google (Viennot, Garcia et Nieh, 2014).

1.10.9 Attaques par transformation de l'APK

Deux classes de méthodes de transformation sont considérées ici, qui visent la génération de différentes variantes d'un APK, et qui sont « Repackaging » et « Code obfuscation ».

1.10.9.1 Repackaging

Cette méthode permet la génération des variantes sans modifier le byte code de l'application, elle vise essentiellement à modifier les signatures de l'application, en utilisant plusieurs techniques.

1.10.9.2 Alignement

La technique d'alignement réaligne les données d'un fichier APK, sans modifier la logique de l'application. L'outil utilisé ici est zipalign, et qui permet de réaligner les données non compressées dans un fichier APK (images, fichiers bruts). L'utilitaire zipalign est disponible dans le SDK Android, et est à l'origine conçu pour l'optimisation des APK, afin de permettre une exécution plus rapide. Le principe est de modifier la structure interne du fichier; ça permet de modifier en conséquence certains des modèles de signature. Donc si un anti malware identifie directement les logiciels malveillants en se basant sur les signatures,

l'utilisation de cette technique peut facilement échapper à la détection de ce système anti malwares.

1.10.9.3 Resignature

Pour qu'un APK puisse s'exécuter sur la plate-forme Android, une signature de ce dernier est requise. Cette signature appartient aux développeurs de l'application. L'idée ici est de resigner l'application avec une nouvelle signature, afin de tenter de contourner les antis malwares, qui détectent les applications malveillantes en se basant sur la signature. Pour resigner un APK, les services publics Keytool et jarsigner, disponibles dans le SDK Java sont utilisés.

1.10.9.4 Rebuild

Cette technique vise à décompiler le fichier APK, puis le recompiler sans apporter aucune modification. Sachant que lors de la recompilation, le nouveau fichier APK garde le même comportement que l'ancien, mais le byte code va différer à cause de la reconstruction. L'outil utilisé ici est le décompilateur/recompilateur apktool. L'APK va donc être décompilé en code Smali, puis recompilé en dalvik.

1.10.9.5 Code obfuscation

Cette technique permet de générer un nouveau APK à partir de l'ancien, en modifiant le code byte, et la taille de l'APK, mais sans toucher au comportement. Les modifications apportées n'affectent donc pas la logique de l'application. Il existe plusieurs techniques de code obfuscation, mais, quelle que soit la méthode utilisée, le processus de modification est le même. Et qui se réalise en décompilant le fichier APK en Smali, apporter les modifications nécessaires aux fichiers Smali, et les recompiler en format dalvik afin d'avoir un nouveau APK.

1.10.9.6 Insertion des méthodes défuntes

Certains antis malwares se basent sur la signature générée par la table de méthodes. Le but ici est d'ajouter une méthode défunte sans l'appeler nulle part. Cette dernière va donc être rajoutée à la table des méthodes, ce qui va modifier la signature de l'application. Afin de permettre cette modification, le fichier Smali d'une classe, va être modifié en ajoutant le code qui permet la déclaration de la méthode, juste après le constructeur de cette classe. Ce dernier peut être repéré en recherchant la ligne du code « # direct methods » (dans le byte code Smali), qui permet la déclaration du constructeur.

1.10.9.7 Renommer les méthodes

Certains antis malwares se basent sur la signature générée à partir des noms des méthodes. Le but ici, est de modifier les noms des méthodes afin de permettre la génération d'une nouvelle signature, différente de l'ancienne. Tout d'abord, une recherche est mise en place dans tous les fichiers Smali afin de différencier les méthodes appartement aux Framework Android de celles déclarées par l'utilisateur. Ensuite, pour chaque méthode déclarée par l'utilisateur, une suite de caractères générée au hasard va être rajoutée au nom des méthodes.

1.10.9.8 Modification des *« control flow gaphs (CFGs) »*

Certains antis malwares comme Androguard, peuvent s'appuyer sur les graphes CFGs afin de générer la signature de l'application. Donc si le CFG des fichiers Smali est modifié, la signature va être affectée. La technique utilisée ici est « Goto-obfuscation », où un GOTO est inséré au début de chaque méthode, afin de sauter directement à la fin de la méthode, et à la fin de cette méthode, un autre GOTO, qui permet de retourner au début de la méthode, est inséré. De cette manière, le graphe CFG de l'application est modifié sans affecter sa logique et son comportement.

1.10.9.9 Cryptage des strings

Certains antis malwares se basent sur la détection des signatures générées à partir des constantes déclarées. La manœuvre ici, est de crypter toutes les constantes avec une simple méthode de cryptographie, (exemple César), et d'appeler la méthode qui permet le décryptage juste avant l'utilisation de chaque constante cryptée.

1.11 Contre-mesures

Dans cette partie nous présentons un aperçu des contre-mesures qui peuvent être appliquées afin de réduire les risques d'attaques contre le système Android.

1.11.1 Sensibilisation des utilisateurs

La sensibilisation des utilisateurs peut permettre d'éviter un nombre important de menaces, car plusieurs vulnérabilités reposent sur les actions réalisées par les utilisateurs. Cette sensibilisation vise donc à expliquer les risques auxquels les utilisateurs peuvent faire face dans différents contextes.

Le système de permission d'Android, par exemple, demande aux utilisateurs l'approbation sur l'ensemble des permissions demandées par l'application, et le choix de l'utilisateur est critique dans certains cas. Il est donc important que l'utilisateur comprenne les différentes permissions et leurs utilités. De plus, l'utilisateur doit être informé sur les risques qui peuvent affecter son périphérique dans le cas des applications téléchargées à partir des sources inconnues.

Les utilisateurs doivent aussi être sensibilisés sur l'importance des mises à jour, non seulement du côté fonctionnel, mais aussi du côté sécurité. En effet, certains utilisateurs n'appliquent par les mises à jour proposées par les fabricants car ils ignorent qu'elles peuvent apporter des corrections de bugs de sécurité.

Plusieurs points de sensibilisation basiques peuvent ainsi être traités, comme les façons sécuritaires d'utiliser l'internet, le réseau, le cryptage de données, etc.

1.11.2 Antivirus

Les Antivirus ne sont pas aussi présents sur la plateforme Android que sur les ordinateurs. Mais l'utilisation de ces derniers peut éliminer un grand nombre de risques. Elles servent donc à détecter la présence des applications malveillantes que l'utilisateur aurait installées.

Les antivirus ne fonctionnent pas de la même façon sur Android que sur les ordinateurs, et cela à cause des différentes contraintes des plateformes mobiles, à savoir, les ressources limitées de la batterie, de la mémoire, du processeur, etc. Pour cette raison, les antivirus s'adaptent afin de protéger l'utilisateur dans sa plateforme mobile, en faisant des analyses heuristiques qui se basent des changements de comportement. Un antivirus peut donc détecter une application malveillante qui envoie des messages à des numéros inconnus par l'utilisateur, ou encore une application consommant excessivement la bande passante, etc.

1.11.3 Pare-feu

L'utilisation des pare-feu peut s'avérer utile dans le système Android. Les pare-feu peuvent réguler le trafic internet, en limitant par exemple le trafic internet à un ensemble d'applications ou au système. L'utilisateur sera donc averti lorsqu'une application tente d'envoyer des données.

Un autre type de par-feu prend en compte d'autres moyens de connexions du périphérique, l'envoi des SMS, des e-mails, etc. Ce type de pare-feu pourrait donc être contrôlé par l'utilisateur.

1.11.4 Anti-spam

L'utilisation des antispams peut aider à limiter le nombre de messages entrant dans le périphérique. Ces derniers contrôlent donc les messages indésirables provenant des différents moyens de communications des Smartphones, à savoir, les appels GSM/SIP, les SMS, les MMS, les e-mails, etc.

1.11.5 Virtualisation

La virtualisation permet d'avoir plusieurs systèmes sur le même périphérique. Ce concept peut donc être utilisé sur les Smartphones, afin d'avoir une instance du système propre et professionnel et une deuxième instance privée dans laquelle l'utilisateur n'accède pas aux données sensibles. Il peut aussi être utilisé pour une virtualisation des applications ; ce qui permet de lancer chaque application dans un environnement isolé.

1.12 Conclusion

Nous avons présenté dans ce chapitre un aperçu du système Android, où nous avons expliqué le fonctionnement de cette plateforme ainsi que ses composants. Nous nous sommes intéressés par la suite au modèle de sécurité proposé par ce système ainsi que ces mécanismes de sécurité. Plusieurs vulnérabilités affectant Android ont été abordées, afin d'éclaircir le mode d'attaque ciblant cette plateforme. Nous avons aussi présenté un ensemble de contre-mesures permettant de réduire les risques menés par l'ensemble de vulnérabilités.

CHAPITRE 2

REVUE DE LITTÉRATURE

2.1 Introduction

Le système d'exploitation Android est le système mobile à plus forte croissance dans le monde. Cette croissance s'est accompagnée de l'émergence de plusieurs applications malicieuses ciblant agressivement le système Android de diverses façons. Ces malwares sont des logiciels malveillants développés avec l'intention de nuire à la vie privée des utilisateurs. Habituellement les malwares sont installés sur les dispositifs des victimes, et permettent, en accédant à un ensemble d'autorisations, de réaliser plusieurs opérations malicieuses.

Plusieurs recherches ont classé les malwares Android en se basant sur différents aspects. Jiang et Zhou (Jiang et Zhou, 2013) ont par exemple classé les malwares selon leurs caractéristiques y compris l'installation, les méthodes d'activation, le comportement malveillant, le repackaging, etc. D'autre part, Cooper et al. (Cooper, 2014) ont classé les malwares en se basant sur leurs capacités d'exécuter certaines opérations dans le système Android, comme le vol des données privées, l'envoi des SMS, effectuer des appels téléphoniques, etc.

De plus, certains malwares modifient les applications Android afin d'exécuter certains scénarios qui ne sont malveillants que dans des contextes bien spécifiques. Ces derniers ne sont donc pas forcément détectés par les anti-malwares comme étant des logiciels malveillants. MacAfee affirme que les attaques par réécriture d'applications ont connu une augmentation de 76% en un espace de trois mois (Caruel, 2015).

Afin de combattre cette augmentation de malwares, les chercheurs se sont concentrés sur trois principaux axes de recherche. Le premier est l'analyse statique et dynamique du code des applications afin de détecter les activités malicieuses avant même que l'application ne soit installée dans le périphérique de l'utilisateur. Le second axe consiste en la modification du système Android afin d'insérer des modules de monitorage pour permettre l'interception

des activités malicieuses. Le troisième axe porte sur la virtualisation afin d'implémenter une séparation entre différents domaines pour isoler les applications installées dans le périphérique.

Dans ce chapitre, nous présentons différentes approches élaborées et appartenant à deux familles de recherche. Mais avant de présenter ces dernières, nous nous intéressons tout d'abord au repackaging des applications Android, et les travaux réalisés afin de détecter cette transformation. Nous présentons par la suite des approches permettant la modification du Framework Android afin de rajouter des mécanismes de contrôle. Puis nous nous intéressons aux approches permettant la sécurisation des applications Android en se basant sur la réécriture des applications.

2.2 Le repackaging des applications

La vie privée des utilisateurs et leurs données confidentielles sont menacés par la réécriture des applications. Cette réécriture permet aux développeurs des logiciels malveillants d'injecter des portions de code malicieuses dans des applications légitimes. Ces dernières seront par la suite recompilées et téléchargés vers le store. Afin de détecter cette transformation effectuée, et de limiter sa propagation, plusieurs chercheurs ont proposé des algorithmes de détection de repackaging. (Crussell, Gibler et Chen, 2012) ont mis en œuvre "DNADroid" qui est un outil permettant la comparaison des graphes de dépendances de deux applications afin de détecter le clonage d'applications. Habituellement une application réécrite modifie le code de l'application légitime d'une certaine manière à empêcher la détection du repackaging. Pour cela DNADroid possède un haut niveau de détection permettant de repérer les modifications apportées dans les packages, les classes, les méthodes, les variables, etc. ainsi que la détection de la suppression de ces derniers. La restructuration des méthodes est aussi prise en compte par DNADroid. Le déplacement de ces dernières entre les différentes classes est donc détectable.

(Zhou, 2012) proposent, de leur côté, DroidMOSS qui est un outil de détection de repackaging basé sur la technique de hachage appelée "fuzzy hashing". Comme l'illustre la figure 2.1, l'outil permet de séparer les instructions contenues dans l'application en plusieurs

unités. Un hachage est appliqué par la suite sur ces dernières et un pourcentage de similitude est calculé en se basant sur la distance entre chaque unité de l'application soupçonnée de repackaging avec celle de l'application originale. Lorsque la similitude dépasse un certain seuil, l'application est donc considérée comme étant un clone.

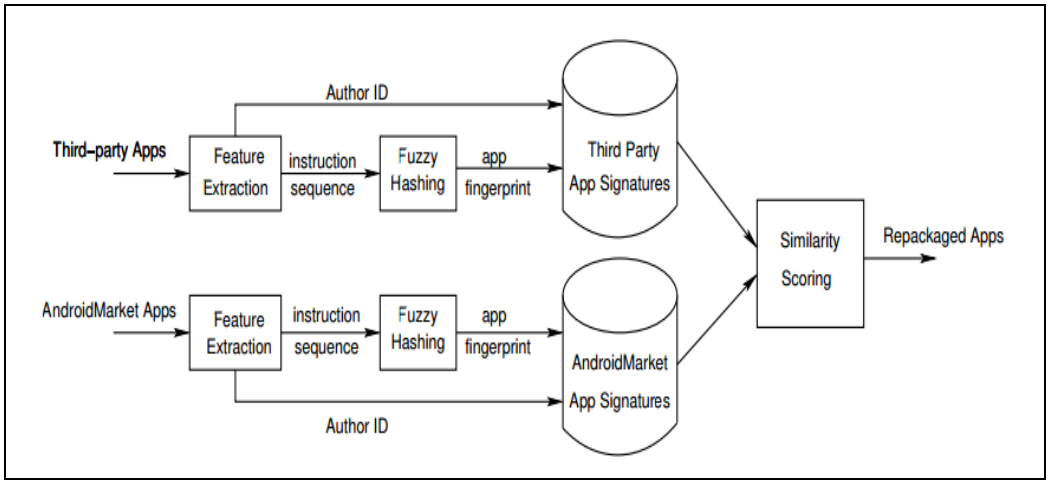

Figure 2.1 Principe de génération d'empreinte par DroidMOSS.
Tirée de (Zhou, 2012)

Une autre recherche portant sur la détection du repackaging des applications menée par (Zhou, Zhang et Jiang, 2013), présente un prototype appelé AppInk qui utilise le mécanisme du Watermarking afin de contrer les attaques par réécriture d'applications. Le Wattermarking permet d'identifier certaines portions de code afin de suivre la façon dont elles sont utilisées. AppInk introduit un concept appelé Manifest app qui permet d'appliquer le principe de watermarking en encapsulant une séquence d'événements d'entrées pour l'application sans aucune intervention de l'utilisateur.

Comme le montre la figure 2.2, AppInk comporte un module de génération du code Watermarking qui permet d'encoder la valeur du watermarking spécifié par le développeur en un graphe de structures, et transformer le graphe en un code watermarking. Par la suite, le module d'instrumentation du code source va insérer le code de watermarking dans le chemin des événements d'entrée identifiée en se basant sur le code source original de l'application. La phase suivante est assurée par le module "Watermark Recognize" qui permet l'identification

de la structure de watermarking en utilisant un modèle spécifique. Si cette dernière est identifiée, AppInk va donc récupérer le code de watermarking et vérifier son originalité.

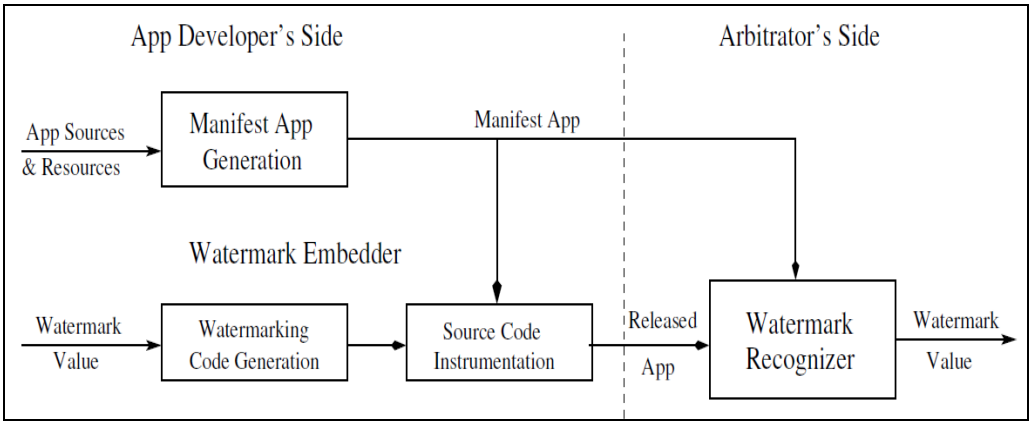

Figure 2.2 Architecture de AppInk.
Tirée de (Zhou, Zhang et Jiang, 2013)

2.3 Modification de la plateforme Android

Plusieurs chercheurs se sont intéressés à la modification de la plateforme Android afin d'apporter des modifications permettant la sécurisation de cette dernière. Plusieurs versions de cette plateforme ont donc été élaborées. (Nauman, Khan et Zhang, 2010) se sont intéressés à l'absence dans la plateforme Android, d'un modèle permettant aux utilisateurs la spécification des ressources pouvant être accessibles par les applications tierces. Apex a donc été développé comme une extension du Framework de permissions d'Android permettant aux utilisateurs de spécifier des contraintes d'exécution détaillées afin de restreindre l'utilisation des ressources par les applications sensibles. Ces contraintes sont introduites à travers des interfaces rajoutées à la plateforme Android et cela avec une moindre modification de l'architecture de sécurité existante. (Zhou et al., 2011) soutiennent la nécessité d'un module permettant le contrôle à l'accès aux informations privées. Ils ont donc proposé TISSA pour permettre aux utilisateurs le contrôle d'exécution en spécifiant quels types d'informations peut être accessible par des applications non approuvées. Comme le montre la figure 2.3, TISSA rajoute certains modules à la plateforme Android, tout en modifiant d'autres modules

existants afin de permettre au module " Privacy Setting Content Provider" de gérer les paramètres de confidentialités pour les applications non approuvées.

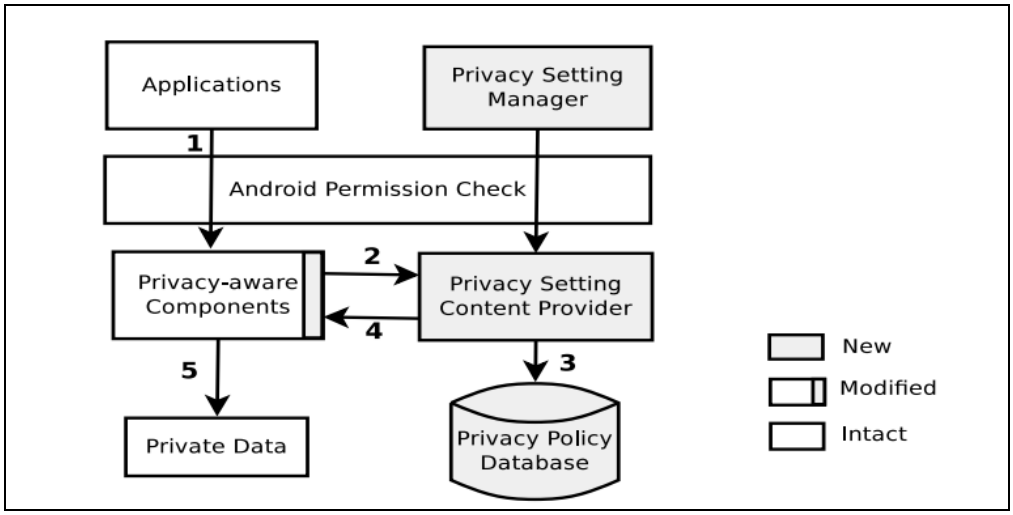

Figure 2.3 Architecture de TISSA.
Tirée de (Zhou et al., 2011)

(Hornyack et al., 2011) ont de leur côté implémenté AppFence qui intègre deux différentes approches pour la protection des données sensibles. En effet, en plus du blocage de toute tentative d'accès à des données sensibles par des applications non autorisées, AppFence propose un moyen d'identification de ces dernières. Cette solution apporte des modifications à la pile d'actions afin de mettre en vigueur des politiques de confidentialité.

Plus récemment (Feth et Pretschner, 2012) ont étendu la plateforme Android afin d'appliquer des mécanismes d'accès basés sur les données et d'utiliser des mécanismes de contrôle. Les politiques de sécurité peuvent être exprimées et appliquées avec des conditions cardinales, spatiales et temporelles. Cette solution permet de se défendre contre deux modèles d'attaques, qui sont les applications malveillantes et les utilisateurs malveillants. Dans le premier cas, certaines actions sollicitées par les applications sont contrôlées, comme l'envoi des SMS, l'accès à la liste des contacts, etc. Le deuxième cas consiste à limiter les ressources et données utilisées par les applications ; par exemple un film ne peut être joué plus de deux

fois, etc. Cette solution modifie la plateforme Android afin de pouvoir intercepter les différents événements demandés par l'utilisateur.

Comme le montre la figure 2.4, l'architecture de ce système se compose de deux composants principaux. Le premier composant est le moniteur de référence, qui quant à lui se compose de deux modules : 1) le "PEP" qui fonctionne comme étant un service système qui a pour tâche d'intercepter les événements et appliquer les politiques de sécurité ; et 2) le "PDP" qui prend la décision sur l'autorisation ou l'interdiction d'évènements. Le deuxième composant "Security Manager Application" sert principalement à fournir les interfaces utilisateurs nécessaires afin d'implémenter les politiques de sécurité.

Le contrôle des communications entre les différentes applications est réalisé en surveillant les intents. De plus le flux d'informations entre les applications et le système de fichiers est observé. La gestion de toutes les interactions est assurée par le "Security Manager" qui est déployé en tant que partie intégrante d'Android et qui ne peut pas être désinstallé par l'utilisateur.

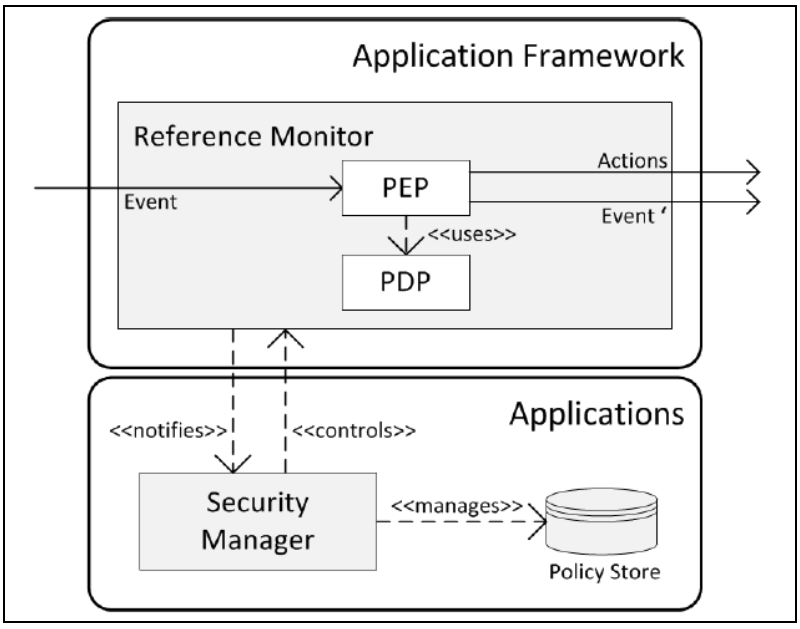

Figure 2.4 Architecture du système de contrôle d'accès.
Tirée de (Feth et Pretschner, 2012)

2.3.1 Comparaison

Chaque approche présentée offre un nombre d'avantages et d'inconvénients. Afin de classer ces approches, nous les avons comparés selon plusieurs caractéristiques qui sont :

- La méthodologie utilisée : elle représente le type de méthodologie sur laquelle se base l'approche. Certaines approches analysent dynamiquement les actions exécutées par les applications. D'autres approches s'intéressent au contrôle d'accès; elles surveillent donc les ressources auxquelles les applications tentent d'accéder. D'autres approches se basent sur l'application des politiques de sécurité afin de contrôler les applications.
- Modification requise : Certaines approches modifient le système Android contrairement à d'autres qui se basent sur la réécriture des applications.
- Nécessité de rooter le périphérique : certaines approches doivent posséder des droits administrateur, ce qui implique la nécessité de rooter le périphérique de l'utilisateur.
- Empêcher l'exécution du code natif : certaines applications ont recours à des codes natifs C, C++ etc. afin de contourner les mécanismes de sécurité et d'exécuter certaines tâches malveillantes.
- Empêcher la création du contexte malicieux : certaines applications collaborent afin de créer un contexte malicieux. Les actions exécutées par chacune des applications ne sont pas forcément reconnues comme étant malicieuses par les mécanismes de contrôles, mais la collaboration entre ces applications peut nuire à la sécurité du périphérique.

Le tableau 2.1 illustre une comparaison entre les différentes approches présentées selon les caractéristiques citées.

Tableau 2.1 Comparaison entre les approches de modification du système Android

Approche	Méthodologie utilisée	Modifications requises	Nécessité de rooter le périphérique	Empêcher l'exécution du code natif	Empêcher la création du contexte malicieux
(Feth et Pretschner, 2012)	Analyse dynamique	Système Android	Oui	Non	Non
Apex (Nauman, Khan et Zhang, 2010)	Renforcement par politique de sécurité	Système Android	Oui	Non	Non
Tissa (Zhou et al., 2011)	Contrôle d'accès aux ressources	Système Android	Oui	Non	Non
AppFence (Hornyack et al., 2011)	Analyse dynamique et contrôle d'accès aux ressources	Système Android	Oui	Non	Non

2.3.2 Limitations

La modification de la plateforme Android permet l'insertion de nouveaux mécanismes de contrôle assurant la sécurité des utilisateurs, et cela en contrôlant le flux d'informations circulé dans l'ensemble de la plateforme. Plusieurs solutions comme expliquées un peu plus haut ont montré leurs efficacités, mais cet axe de recherche souffre de plusieurs limitations considérables.

Le fonctionnement des solutions présentées requiert des droits administratifs, ce qui signifie la nécessité de rooter le périphérique. Cette opération pourrait faire face aux droits d'utilisation du périphérique, ce qui implique la perte de la garantie de l'utilisateur. De plus,

les différentes solutions permettent d'avoir plusieurs versions du Framework Android. Dans ce cas, des problèmes de compatibilité peuvent faire surface entre les applications et le système installé, ainsi qu'entre les Frameworks et les types de périphérique.

De plus, les contrôles de sécurité peuvent être limités, puisque les applications sont limitées aux politiques de sécurité prise en charge par le Framework Android modifié.

2.4 La réécriture des applications Android

Une deuxième catégorie de solutions permettant la sécurisation de la plateforme Android, est basée sur la réécriture des applications. Cette catégorie de solution ne nécessite aucune modification à la plateforme Android, et peut être déployée facilement. Ces solutions offrent généralement un Framework de modification d'applications qui permet la décompilation des applications Android, et l'injection d'un code de contrôle permettant la surveillance des actions sensibles réalisées par les applications.

(Reddy, 2011) ont développé le système ACPlib.redexer qui permet la réécriture des applications Android avec l'objectif de mettre en œuvre des autorisations plus fines dans les applications. ACPlib utilise un ensemble de services pour effectuer des appels aux API sensibles. Quand une application Android effectue un appel qui nécessite une permission spécifique, cette demande sera interceptée par le service approprié dans ACPlib, qui va exécuter la méthode et retourner le résultat. Cette manœuvre permet aux services de gérer les appels sensibles, tout en attribuant les autorisations nécessaires aux méthodes.

Cette approche est efficace pour certains types de malware, mais elle ne permet de gérer que les permissions de méthodes, sans prendre le contexte en compte. (Mulliner, 2014) utilisent la réécriture d'application afin d'effectuer une transformation permettant de révéler les vulnérabilités dans Google In-App Billing. Ils utilisent donc un faux Google store et ils effectuent une réécriture d'applications afin de lier les applications au faux store à la place du store originale. Bien que leurs objectifs de réécriture sont limités dans leurs portées et reposent sur la conversion en code source Java, leur travail illustre une des applications effectuant la réécriture des applications Android.

(DIETZ, 2011) implémentent de leur côté Quire, un système empêchant l'escalade des permissions en suivant ces dernières à travers la chaine d'appel IPC. Crepe (Conti, Nguyen et Crispo, 2011) permet d'identifier les permissions accordées durant l'installation des applications, et exige un contrôle afin de permettre l'utilisation de ces permissions dans certains contextes lors de l'exécution de l'application.

2.4.1 Réécriture des méthodes JAVA

Les méthodes JAVA peuvent être potentiellement vulnérables aux attaques; si le comportement de ces dernières est malicieux, la sécurité du périphérique peut être corrompue. (Benjamin Davis 2012) proposent une étude qui permet aux utilisateurs de modifier le comportement des méthodes.

I-arm-Droid (Benjamin Davis 2012) se base principalement sur la réécriture des APK. L'idée principale est que chaque APK va être décompilé en code Smali, et une recherche sera réalisée afin de récupérer la liste complète des méthodes existant dans l'application. L'utilisateur aura donc accès à cette liste de méthodes, et s'il soupçonne le comportement d'une parmi elles, il peut redéfinir un comportement plus approprié.

La liste de méthodes offerte à l'utilisateur comporte la signature de chaque méthode, qui est définie par le nom du package, le nom de la classe, et celui de la méthode en question. Donc, après avoir choisi la méthode désirée, l'outil élaboré dans cette étude, va permettre à l'utilisateur d'introduire du code source JAVA afin de personnaliser la méthode. Par la suite, l'outil va générer le byte code Smali propre à chaque méthode modifiée. Certaines manipulations sont nécessaires afin de permettre l'appel à ces nouvelles méthodes créées.

2.4.1.1 Principe de création des nouvelles méthodes

Pour chaque méthode que l'utilisateur choisit, et dont il redéfinit le comportement en code source JAVA, l'outil va créer une nouvelle méthode statique. Cette méthode va être créée dans une nouvelle classe et mise dans un nouveau package. Le nom de ce dernier va être généré arbitrairement afin de ne pas confondre les anciennes classes avec les nouvelles. Les

arguments des nouvelles méthodes créées dépendent des différents types de méthodes.

Si la classe qui contient la méthode qui doit être modifiée est de type final, la nouvelle classe qui sera créée est une classe qu'on appelle « stub », et si la classe est non final, la nouvelle classe sera appelée « wedge ».

Une nouvelle classe stub est créée, en gardant la même hiérarchie de l'ancienne classe, mais tout en mettant la nouvelle classe dans un package dont le nom est généré au hasard. La figure 2.5 représente un exemple de la méthode OpenStream() contenue dans la classe URL. La nouvelle classe (à droite de la figure) va donc garder la même hiérarchie.

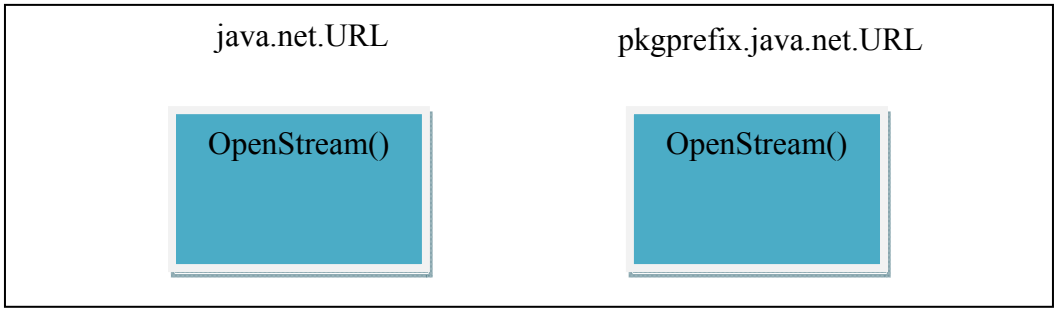

Figure 2.5 Création d'une classe stub

La classe wedge est créée dans un package généré au hasard, concaténé avec le mot clé wedge. De plus, la nouvelle classe va hériter de l'ancienne classe, et toutes les classes dans l'application, qui héritent de cette dernière, vont donc hériter de la classe wedge. Voir figure 2.6.

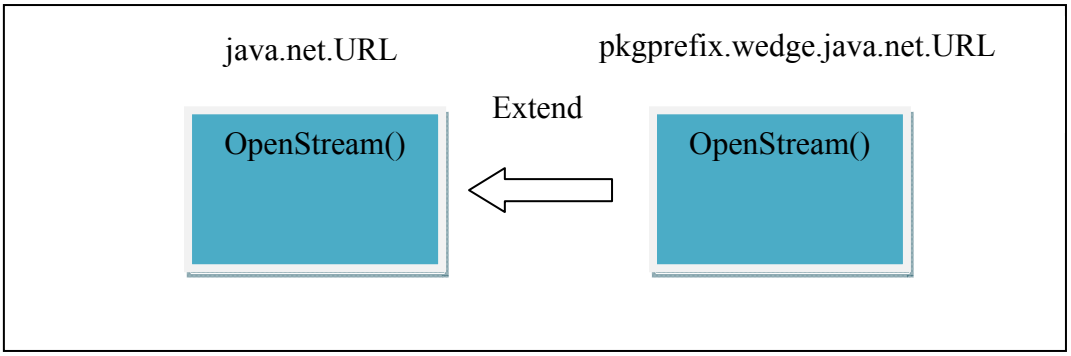

Figure 2.6 Création d'une classe wedge

2.4.1.2 Type des méthodes

Comme nous l'avons déjà mentionné, les arguments de la nouvelle méthode créée dépendent du type de la méthode à modifier. Dans java, il existe plusieurs types de méthodes. Nous présentons dans cette partie la méthode créée pour chaque type de méthode à modifier.

A. Méthodes statiques

Comme nous l'avons expliqué un peu plus haut, la méthode créée est de type statique, quel que soit le type de la méthode à modifier. Donc dans ce cas-là, le passage d'une méthode statique à une autre méthode statique, ne nécessite pas la modification des arguments.

B. Méthodes d'instance

Ce type de méthodes, contrairement aux méthodes statiques, nécessite un objet déjà créé pour permettre leurs appels. Donc la nouvelle classe va dépendre de la classe qui contient la méthode en question. Dans le cas de la création d'une classe stub, la nouvelle méthode va contenir les mêmes arguments que l'ancienne méthode en plus d'un argument rajouté au début. Cet argument est du même type que l'ancienne classe. Donc, si on prend l'exemple de la figure 2.7, l'argument var0 de la nouvelle méthode setContentView(), sera de type Activity. De cette manière, l'ancienne méthode pourrait être appelée dans la nouvelle méthode en utilisant l'expression : var0.setContentView(var1).

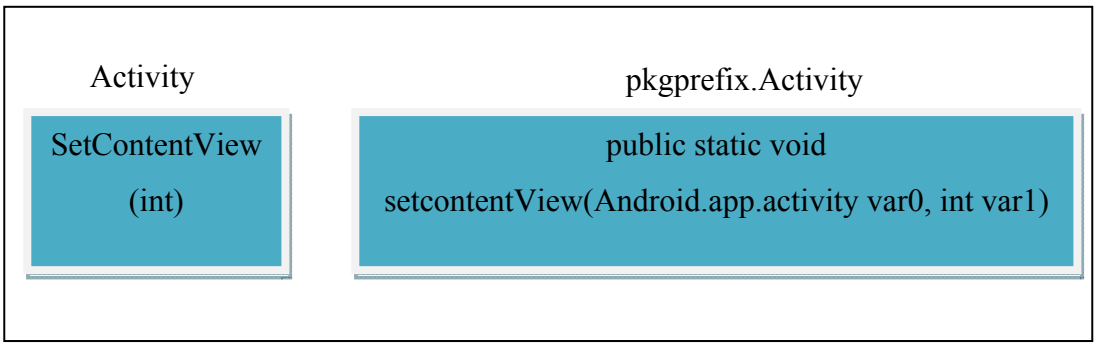

Figure 2.7 Stub d'une méthode d'instance

Dans le cas de la création d'une classe wedge, et comme illustrée dans la figure 2.8, la nouvelle méthode gardera les mêmes arguments que l'ancienne méthode. Puisque la classe wedge hérite de l'ancienne classe, l'ancienne méthode peut-être appelée dans la nouvelle en utilisant l'expression : Super.setContentView(var0).

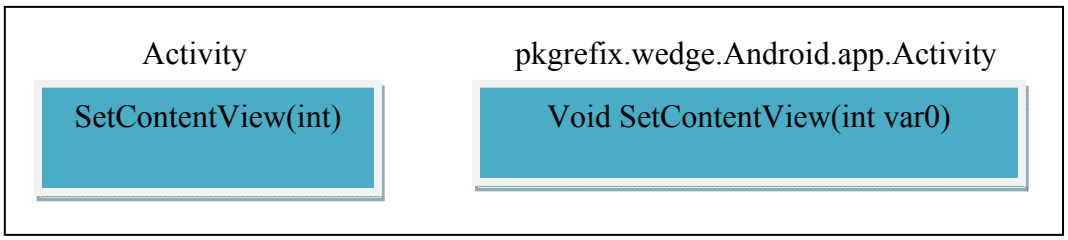

Figure 2.8 Wedge d'une méthode d'instance

C. Les constructeurs de classes

Pour les constructeurs de classes, le patron de conception « Factory » est utilisé. Donc, une méthode avec les mêmes arguments que le constructeur est créée, et elle retourne un objet du même type que l'ancienne classe. Voir figure 2.9.

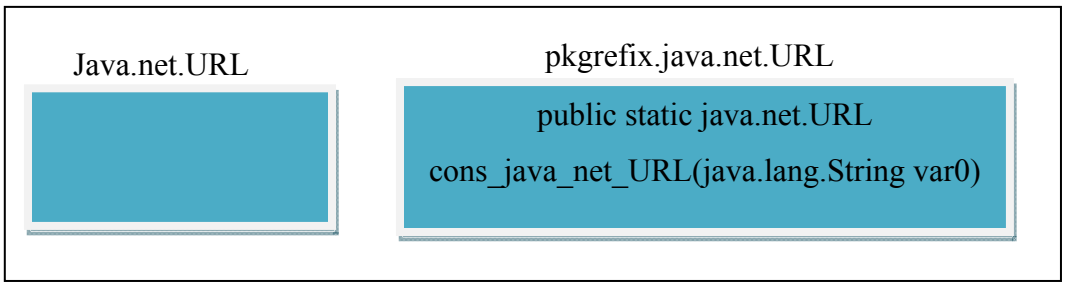

Figure 2.9 Méthode affectée à un constructeur

2.4.1.3 Transformation des appels

L'étape suivante consiste à modifier les appels des méthodes afin de pouvoir appeler les nouvelles méthodes à la place des anciennes. Les méthodes statiques sont invoquées par l'instruction : « invoke-static » et « invoke-static/range ». Comme la nouvelle méthode qui va

être créée est de type statique aussi, le seul détail à changer dans le byte code Smali est la référence vers la nouvelle méthode.

Les méthodes d'instance sont invoquées par l'instruction « invoke-virtual », où l'instruction contient la référence vers la méthode qui doit être appelée et une suite de registres qui représentent les arguments de la méthode dont le premier contient la référence à l'objet dans lequel la méthode doit être appelée. Donc, la modification de cet appel se fera par le remplacement de « invoke-virtual » par « invoke-static », le remplacement de la référence de l'ancienne méthode vers la nouvelle, et aucun changement concernant les registres. Puisque, comme déjà expliqué un peu plus haut, un argument du même type que la classe qui contient la méthode en question, est rajouté, donc ce dernier va contenir l'objet qui correspond à l'ancienne classe.

Concernant les constructeurs de classes, ils sont invoqués par l'instruction « invoke-direct», et une suite de registres. Le premier registre contient la référence vers le lieu où l'instance sera sauvegardée après sa création, et la suite des registres contiennent les arguments du constructeur. Donc ici, « invoke-direct » sera remplacé par « invoke-static », la référence vers la méthode sera changée par la référence à la nouvelle méthode, et une instruction est rajoutée juste après l'instruction « invoke-static ». Cette instruction permet de copier l'instance retournée par la méthode stub, dans le premier registre qui représente l'instance de l'objet créé.

2.4.1.4 Recompilation du code Java

Après avoir récupéré le code java introduit par l'utilisateur, l'outil va donc créer les classes stubs et wedges. Comme nous l'avons déjà montré, ces classes vont contenir le nouveau comportement des méthodes, spécifié par l'utilisateur. Par la suite, et comme le montre la figure 2.10, ce code source va être compilé en byte code dalvik, et va être copié dans le package de l'application. L'APK va donc contenir son propre byte code, plus le byte code des nouvelles classes et méthodes ajoutées.

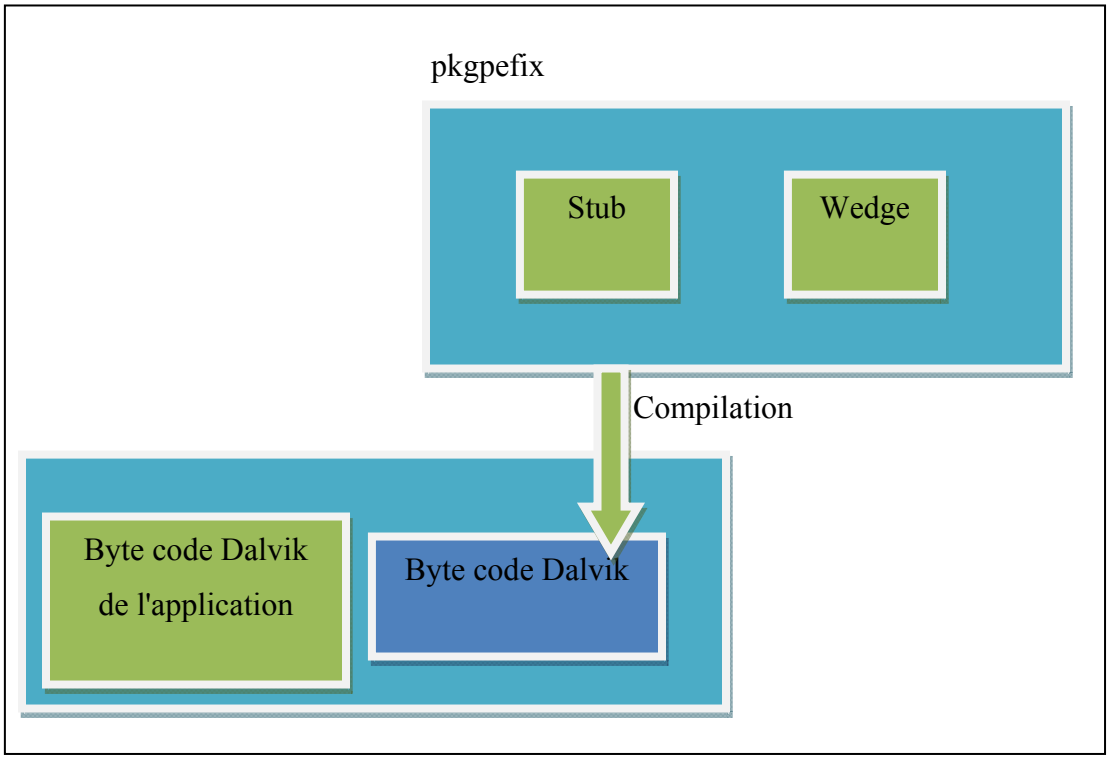

Figure 2.10 Génération d'APK

2.4.2 Injection d'un code de contrôle dans les APK

Certaines études de sécurisation des applications Android injectent un code de contrôle dans l'application elle-même. (Xu et al., 2012) proposent une étude qui empêche les applications APK de s'exécuter directement sur le périphérique en les faisant passer par un outil appelé Aurasium. Ce dernier permet la génération d'une version plus renforcée, qui va être installée par la suite dans le périphérique de l'utilisateur. L'outil ici intercepte donc les interactions entre l'application et le système, et si l'application tente d'appeler une fonctionnalité système, l'outil annonce la manœuvre à l'utilisateur afin d'autoriser ou de refuser l'accès.

Les contrôles visés par Aurasium sont principalement les appels des fonctionnalités système; par exemple, si l'utilisateur tente d'accéder à un serveur distant, l'outil va vérifier l'adresse du serveur dans une liste noire; si l'application tente d'envoyer un SMS, l'outil vérifie si le numéro est un numéro surtaxé. D'autres contrôles des politiques de sécurité sont effectués pour autoriser ou interdire l'accès à des informations privées, comme IMEI, IMSI, SMS stockés, caméra, etc.

Dans cette approche, les mécanismes de contrôles sont injectés dans le code byte de l'application elle-même. Afin de permettre cela, le processus d'injection se divise en deux phases. Une première phase permet la décompilation de l'APK en utilisant le décompilateur apktool. Une deuxième phase pour l'injection des mécanismes de contrôles. Dans cette dernière phase et comme le montre la figure 2.11, tous les composants nécessaires au fonctionnement du monitoring sont injectés dans le package de l'application.

Le code byte permettant le contrôle de l'application est d'abord injecté dans les fichiers Smali. Plusieurs classes Smali représentant le code byte d'Aurasium sont aussi rajoutées à l'application. Par la suite, les ressources nécessaires sont copiées dans le dossier des ressources non compilées. Afin de permettre l'exécution d'Aurasium, le fichier AndroidManifest.xml est modifié d'une manière à permettre à la classe principale du code du monitoring de s'exécuter en premier, et cela en changeant le point d'entrée principal de l'application vers cette dernière.

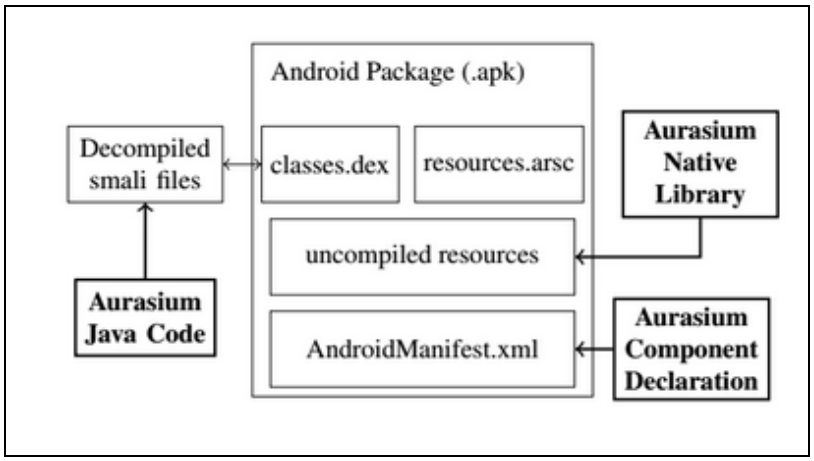

Figure 2.11 Injection des mécanismes de contrôle.
Tirée de (Xu et al., 2012)

2.4.2.1 Politiques de sécurité

Les politiques de sécurité sont injectées au moment de l'injection du code, mais cette étude offre une deuxième alternative. Donc lors de l'exécution de la version renforcée de l'APK (version contenant le code de contrôle), l'application recherche le processus de «

Aurasium's Security Manager » qui est une application semblable à celle injectée dans l'APK, mais qui permet une gestion plus flexible des politiques de sécurité. Cette application peut être installée sur le périphérique de l'utilisateur afin de définir ces propres politiques de sécurité. Donc si ce processus est exécuté, les politiques de sécurité vont être récupérées à partir de cette application. Dans le cas contraire, les politiques qui sont injectées dans l'APK vont être utilisées.

Le code de contrôle injecté comporte plusieurs politiques de sécurité, on cite ici quelques exemples. Lors de l'appel aux fonctionnalités du système, l'application renforcée interrompt l'exécution et consulte l'utilisateur afin de confirmer l'accès :

- ioctl() : est un appel système pour des opérations d'entrée/sortie.
- getaddrinfo() and connect() : responsables des connexions sockets.
- dlopen(), fork() and execvp() : permettent l'exécution du code native.
- read(), write() : permettent l'accès aux fichiers système.

Aurasium permet aussi de contrôler les données personnelles de l'utilisateur, comme le IMEI, IMSI, le numéro du téléphone, ainsi que le contrôle de certaines actions comme l'envoi des SMS, ou l'accès des coordonnées de géolocalisation. Si une application tente donc d'exécuter une action sensible, l'utilisateur sera averti.

2.4.3 Contrôle de divulgation de données privées dans l'application

D'autres études de sécurisation des applications Android sont basées sur le contrôle des données privées des utilisateurs, afin de suivre le cheminement de ces données et détecter toute tentative de divulgation. (Zhang et Yin, 2014) proposent un mécanisme de contrôle appelé Capper, basé sur l'implémentation des politiques de sécurité. Ce mécanisme a pour but de sécuriser les données privées des utilisateurs. Capper (Zhang et Yin, 2014) empêche l'installation des applications Android telles quelles. Comme illustré par la figure 2.12, Capper fait passer les applications par un moteur de réécriture d'applications appelée BRIFT qui convertit le code Dalvik DEX en code JAVA en utilisant l'outil des2jar (Pxb, 2015). Le code source Java sera par la suite converti en une représentation intermédiaire IR en utilisant

le framework Scoot (Corporation., 2010). Cette représentation permet d'effectuer l'analyse statique de flux de données, ainsi que l'instrumentation du code byte.

Capper permet d'analyser la représentation intermédiaire du code de l'application, et mène une analyse statique de flux de données afin de calculer le nombre de portions de code impliquées dans la divulgation des données privées des utilisateurs. Par la suite, il insère des instructions tout au long des portions de code identifiées, afin de garder la trace de propagation des données lors de l'exécution de l'application.

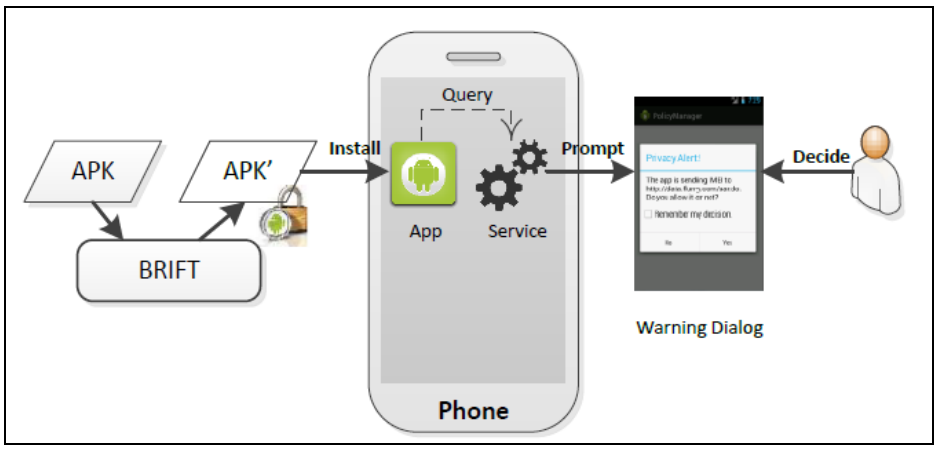

Figure 2.12 Architecture de Capper.
Tirée de (Zhang et Yin, 2014)

Lors de l'exécution de l'application renforcée par Capper, si une fuite de données est détectée, l'utilisateur sera averti afin d'accepter ou interdire l'action en cours.

2.4.4 Purification des APK

Une des méthodes de sécurisation des APK basées sur la réécriture est la purification des APK. Cette approche consiste à éliminer les portions de code malicieuses contenues dans l'application. (Yang et al., 2014) proposent APKLancet, qui est un Framework permettant la réécriture des APK, tout en supprimant les portions de code jugées malicieuses. APKLancet se base sur une base de données prédéfinie contenant les portions de code d'un ensemble de malware, ainsi que leurs caractéristiques. Ces portions de code indésirable sont tout d'abord

repérées via une analyse du code byte de l'application, et ils seront par la suite éliminés sans affecter le comportement de l'application.

APKLanncet permet l'analyse automatique des applications Android, afin de détecter et purifier l'apk des portions de code indésirable, en se basant sur des caractéristiques extraites des bibliothèques publicitaires, des plugins d'analyse, et de 8000 malwares.

2.4.4.1 Code indésirable

Les portions de code malicieuses sont toute portion de code permettant l'exécution d'un comportement indésirable; par exemple : la collecte d'informations sans préavis ni d'autorisation de l'utilisateur. Ces portions de code peuvent se présenter sous plusieurs formes.

2.4.4.2 Code malicieux

Le code malicieux est considéré dans cette étude comme étant le type le plus dangereux des codes indésirables. Ce dernier est défini comme étant un code qui conduit volontairement à un comportement indésirable, qui peut aller de l'envoi d'un simple SMS, jusqu'à l'obtention des privilèges Root par des exploits.

Les attaques par injection de code malicieux sont fréquentes. On peut citer pour exemple, le malware Geinimi qui est le premier malware sophistiqué permettant l'infection des applications Android par l'injection d'un code malicieux en se basant sur le repackaging.

2.4.4.3 Bibliothèques d'annonce et de publicité

Les bibliothèques de publicité représentent les codes tiers le plus utilisés par les applications Android. Environ 50% des applications utilisent au moins une bibliothèque de publicité. Le code permettant l'utilisation de ces bibliothèques peut être considéré comme étant un code indésirable si ce dernier est injecté par une décompilation de l'application.

2.4.4.4 Les plugins d'analyse d'applications

Ces plugins sont mis à la disposition des développeurs afin de recueillir des données sur l'utilisation de leurs applications (nombre d'utilisateurs actifs, localisation des utilisateurs, etc.) et cela pour prendre des décisions concernant le développement des applications. Ces plugins s'exécutent en arrière-plan, et n'ont aucune interaction avec l'utilisateur. Ils peuvent être considérés comme étant un code malicieux s'ils permettent la collecte d'informations sur les utilisateurs pour des fins indésirables.

2.4.4.5 Fonctionnement d'APKLancet

APKLancet (figure 2.13) analyse l'ensemble des fichiers contenus dans l'apk en parcourant le fichier manifest ainsi que les fichiers de ressources afin de collecter les informations sur les déclarations contenues dans l'application. En se basant par la suite sur une base de données comportant des caractéristiques des différents types des portions de code indésirable, elle permet la localisation de ces derniers dans les fichiers Smali de l'application.

La base de données contenant les caractéristiques des portions de code vulnérables est construite en se basant sur 8000 malwares appartenant 184 familles de malware. Deux applications affectées par un ou plusieurs malwares de chaque famille ont donc été sélectionnées au hasard. L'analyse de ces applications permet la détection des caractéristiques de chaque famille de malware.

Les applications sélectionnées sont analysées en se basant sur des portions de code responsables de certains événements ou fonctions reliés au comportement des malwares. Ces portions de code sont appelées les classes index, et elles sont communes à toutes les variantes de la même famille d'un malware.

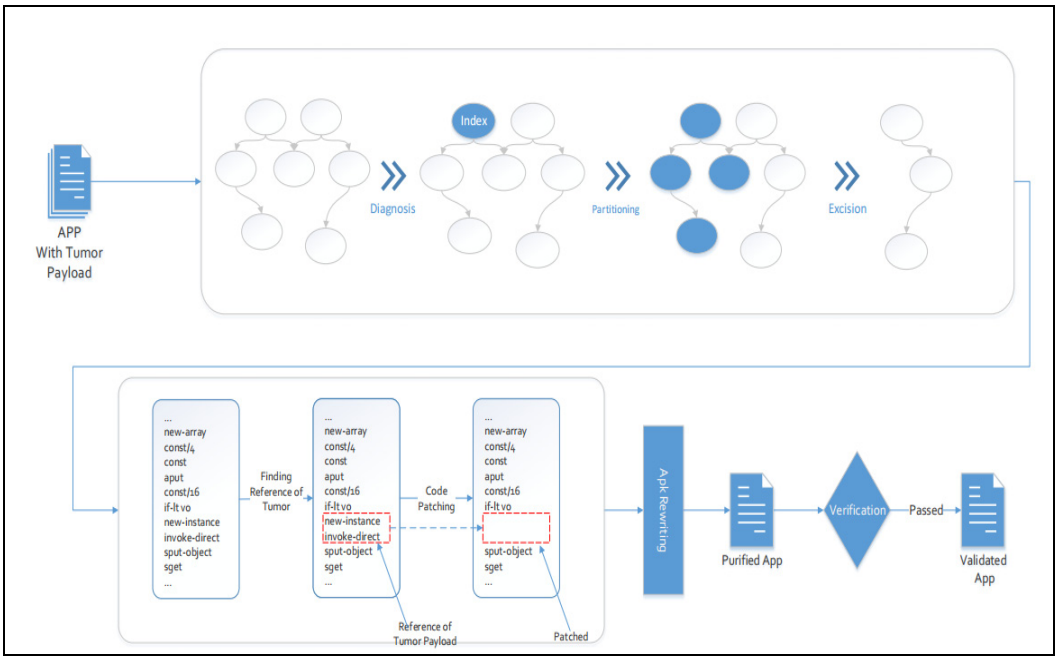

Figure 2.13 Processus de purification de l'apk.
Tirée de (Yang et al., 2014)

Afin que le malware affecte l'ensemble de l'application, il enregistre ces propres classes comme étant des points d'entrées de l'application. Les applications Android peuvent avoir plusieurs points d'entrée. APKLancet se concentre sur cinq types d'entrée qui sont : les activités, les services, broadcast receiver, content provider, et Android.app.Application qui est le point d'entrée principale de l'application.

Concernant les bibliothèques publicitaires et les plugins d'analyse, ces derniers sont généralement bien documentés, elles offrent aux développeurs un ensemble d'interfaces unifiées. L'identification des caractéristiques est donc faite par la sélection d'un ensemble contenant les classes essentielles d'une même bibliothèque, ces dernières sont identifiées comme étant la classe index de la bibliothèque.

Une fois que la base de données est construite, APKlancet vérifie donc toutes les classes de l'application qui font partie des cinq points d'entrée de l'application. Il compare donc le contenu de ces classes avec les caractéristiques contenues dans la base de données afin

d'identifier les portions de code indésirable. APKlancet se base aussi sur la technique de fuzzy hashing (Zhou et al., 2012) afin de contrer la technique du code abfuscation.

2.4.4.6 Identification des portions de code indésirables

Après avoir analysé les points d'entrée de l'application à purifier, APKLancet doit éliminer l'intégralité du code byte de malware dans l'apk, tout en incluant les déclarations dans le fichier manifest, ainsi que les ressources et les bibliothèques injectées. Pour cela, APKlancet utilise différentes stratégies selon le type du code indésirable.

Les bibliothèques publicitaires ainsi que les plugins d'analyse sont indépendants des applications, ce qui facilite leurs éliminations. Leurs ajouts impliquent généralement l'ajout de trois types de ressources qui sont : un fichier JAR, des tags Meta-data et des ajouts de code dans les fichiers XML et java. Si APKlancet identifie des classes d'index suspectes dans ces bibliothèques, ces trois types de ressources seront identifiés en tant que code indésirable.

Contrairement aux bibliothèques publicitaires et aux plugins d'analyse, aucune documentation sur le code malicieux d'un malware n'est disponible, et ce dernier utilise plusieurs techniques afin qu'il soit indétectable. De plus, différemment des bibliothèques, le code malicieux du malware est injecté dans le même package que le code de l'application, ce qui complique sa détection. APKLancet utilise alors un algorithme afin de détecter les portions de code du malware. Cet algorithme est présenté à la figure 2.14, et il prend en entrée l'ensemble "O" de toutes les classes appartenant à l'APK, ainsi qu'un ensemble "E" contenant les classes identifiées comme étant malicieuses.

L'algorithme fournit en sortie un ensemble "M" contenant la totalité de la portion de code malicieuse contenue dans les classes. L'algorithme se base sur la fonction Find_invoke_dest qui permet l'identification de l'ensemble de classes malveillantes en se basant sur le graphe de dépendance de l'application, si une classe déjà identifiée en tant que malicieuse invoque une méthode appartenant à une autre classe, cette dernière est donc ajoutée à l'ensemble des classes malveillantes.

Require:
The set of class in an APK, O;
The set of malicious class in O, E;
Ensure:
The set of malicious code class, M;
1: $M = E$
2: **repeat**
3: $D = \varnothing$
4: **for all** m in M **do**
5: $D \leftarrow Find\_invoke\_dest(m)$;
6: **end for**
7: **for all** d in D **do**
8: **if** d in O **AND** d not in M **then**
9: M ← d
10: **end if**
11: **end for**
12: **until** M is not modified
13: **return** M;

Figure 2.14 Algorithme de détection du code malicieux.
Tirée de (Yang et al., 2014)

2.4.4.7 Élimination des portions de code indésirables

Après avoir identifié les portions de code indésirables, ces derniers ainsi que les fichiers jar doivent être éliminés. Afin que l'application puisse fonctionner suite à ces modifications, une réparation du code de l'application, et du fichier manifest est nécessaire.

Tout comme l'application elle-même, les portions de code malicieuses injectées dans cette derrière nécessitent certaines déclarations dans le fichier manifest, ainsi que des modifications dans les points d'entrée de l'application. Les approches de modifications des points d'entrées les plus utilisées sont :

- L'injection directe dans le point d'entrée : le code malicieux enregistre un nouveau service, un broadcast receiver ou une activité.
- Falsification du point d'entrée principal : certaines portions de code malveillantes changent le point d'entrée principal de l'application (déclaré dans le fichier manifest) vers une autre classe à laquelle il appartient.

- Modification de l'héritage du point d'entrée : certaines portions de code malicieuses cachent le point d'entrée de l'application par un héritage de classe. Le point original d'entrée est donc gardé, et la nouvelle classe malicieuse peut se cacher sans qu'elle soit déclarée dans le fichier manifest. Cependant la relation d'héritage de la classe principale est modifiée. Comme le représente la figure 2.15, le point d'entrée com.normal.activity hérite de la classe Android.app.Activity, mais après l'injection du code malicieux, la super classe est changée à com.mal.Activity qui est une classe malveillante. Dans ce cas, quand le point d'entrée est exécuté, la fonction malicieuse dans la super classe est exécutée en premier.

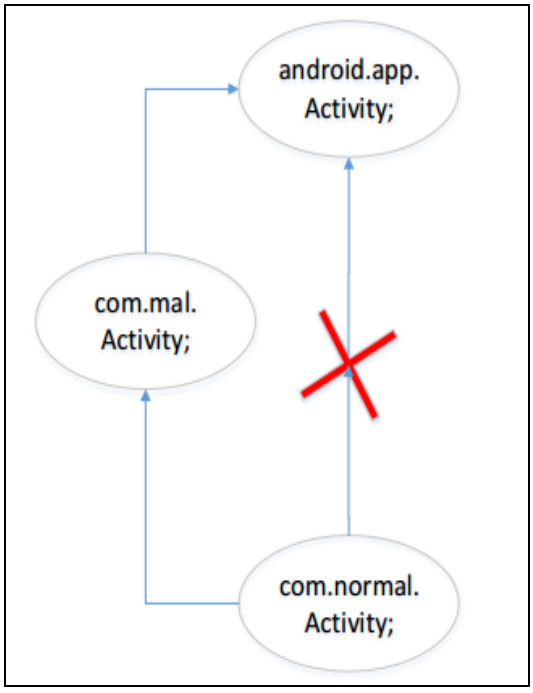

Figure 2.15 Modification de l'héritage du point d'entrée

APKLancet analyse le code Smali de l'application, identifie les points d'invocation des méthodes, et vérifie si la méthode invoquée est une méthode malicieuse. Dans cette situation APKLancet procède selon deux cas de figure :

- La méthode invoquée hérite et modifie une méthode d'une autre classe : dans ce cas-là, APKlancet modifie l'instruction d'invocation, afin d'appeler la méthode de base appartenant à la super classe.

- La méthode invoquée appartient à une classe malveillante : dans ce cas, APKLancet analyse les instructions qui dépendent de la valeur de retour de l'instruction d'invocation, et élimine l'instruction d'invocation, tout en modifiant les instructions dépendant de cette dernière.

De plus, afin que APKLancet puisse réparer le code de l'application après avoir supprimé les portions de code malveillantes, le fichier manifest de l'application doit être contrôlé pour gérer les points d'entrées liés aux codes malveillants. Le premier point à vérifier est le point d'entrée principale de l'application. Si ce dernier a été modifié par un malware, APKLancet essaie de trouver la classe " launcher " afin de l'utiliser en tant que point d'entrée principale. Si cette classe est absente, APKLAncet utilise alors la première activité déclarée dans le fichier manifest comme point d'entrée. Dans le cas d'une modification de l'héritage du point d'entrée, la première super classe du point d'entrée original qui n'est pas identifiée en tant que malicieuse doit être retrouvée, et la relation d'héritage doit être rectifiée.

2.4.5 Comparaison

Nous avons comparé les approches basées sur la réécriture des applications selon les mêmes caractéristiques que les approches modifiant le système Android. Le tableau 2.2 illustre cette comparaison.

Tableau 2.2 Comparaison entre les approches de réécritures des applications

Approche	Méthodologie utilisée	Modifications requises	Nécessité de rooter le périphérique	Empêcher l'exécution du code natif	Empêcher la création du contexte malicieux
I-ARM-Droid (Benjamin Davis 2012)	Réécriture du code Smali	Applications	Non	Non	Non
Aurasium (Xu et al., 2012)	Réécriture du code Java	Applications	Non	Non	Non
Capper (Zhang et Yin, 2014)	Réécriture du code Java	Applications	Non	Non	Non
APKLancer (Yang et al., 2014)	Réécriture du code Smali	Applications	Non	Non	Non

2.4.6 Limitations

Comme illustré par les travaux que nous avons présentés, la méthode de sécurisation d'application Android par leurs réécritures offre plusieurs avantages, mais elle souffre aussi de certaines limitations. La plupart des systèmes basés sur la réécriture d'applications ne peuvent pas résoudre le problème de "Java reflection" qui est utilisé pour appeler des méthodes sans documentation. Ce type d'appel peut accéder à des informations privées, ou encore entrainer des effets secondaires inattendus. De plus, le système Android permet aux applications d'utiliser des codes natifs écrits en C et C++, et cela afin d'accélérer certaines tâches. Le problème avec cette caractéristique du système Android c'est qu'elle ouvre la porte aux attaques, puisque les codes natifs n'obéissent pas aux mêmes règles et politiques de sécurité appliquées sur l'ensemble de l'application.

2.5 Conclusion

Ce chapitre présente une revue de littérature portant sur les approches pour la sécurisation du système Android. Nous avons présenté plusieurs approches appartenant à deux familles de solutions. Une première famille qui vise la modification du système Android afin de rajouter des mécanismes permettant la sécurisation de ce dernier. Et une deuxième famille permettant la réécriture des applications pour appliquer des politiques de sécurité, qui ne nécessitent pas la modification du Framework Andoid.

Les approches présentées ont démontré une importante efficacité pour protéger l'utilisateur contre les menaces. Cependant, elles présentent certains inconvénients, comme l'incapacité de lutter contre l'exécution des codes natifs. De plus, la collaboration entre les applications afin de créer des contextes malicieux n'est traitée par aucune approche.

Nous présentons dans le chapitre suivant notre approche qui permet contrairement aux approches étudiées de lutter contre la collaboration entre les applications, et de contrôler l'exécution des codes natifs.

CHAPITRE 3

ARCHITECTURE ET IMPLÉMENTATION

3.1 Introduction

Ce chapitre présente la solution développée et proposée dans la présente étude. Ladite solution étant centralisée, elle permet la sécurisation des applications Android par l'interception des appels aux méthodes API. Ce chapitre comporte donc les deux axes principaux de notre solution, à savoir, le Framework de réécriture d'applications et le contrôleur centralisé d'applications.

3.2 Solution proposée

Dans ce présent travail, une solution visant à modifier le comportement des applications Android est proposée afin de fournir un mécanisme de contrôle permettant l'application d'un ensemble de politiques de sécurité. Ces dernières sont introduites par l'utilisateur et permettent une exécution sécuritaire de l'ensemble des applications.

Les méthodes API dans le système Android permettent aux développeurs l'utilisation des fonctionnalités les plus poussées. Les applications peuvent donc à travers ces méthodes accéder à différentes propriétés sensibles fournies par le système Android. À titre d'exemple l'envoi des SMS, l'accès à la caméra, l'accès aux coordonnées GPS, etc. Ces fonctionnalités doivent donc être contrôlées afin d'assurer une exécution plus sécuritaire des applications.

Quand une application installée sur le périphérique de l'utilisateur tente d'exécuter une méthode API, l'appel à cette dernière doit être intercepté, afin d'extraire les paramètres nécessaires permettant le contrôle de l'appel. Grâce à ces paramètres ainsi qu'aux politiques de sécurité spécifiées par l'utilisateur, la décision sur l'autorisation ou l'interdiction de l'exécution de la méthode peut être prise. Et pour permettre l'interception des appels, certaines instructions qu'on abordera par la suite seront rajoutées à l'application.

La décision sur l'exécution des méthodes sollicitées est prise lors de l'exécution de l'application, plus précisément, au moment où la méthode est appelée. Cette décision est prise par un contrôleur d'applications conçu dans cette étude. L'application envoie donc les paramètres collectés à partir de l'appel au contrôleur d'applications, qui à son tour, et en se basant sur la liste des politiques de sécurité, prend la décision adéquate et envoie la réponse à l'application. En fonction de la réponse reçue par l'application à partir du contrôleur, cette dernière va autoriser, interdire, ou mettre en pause l'exécution de cette méthode API.

3.3 Choix de conception

L'objectif de cette étude est d'implémenter un système flexible permettant l'application des politiques de sécurité sur les applications Android. Plusieurs alternatives ont été envisagées dans cette étude afin que la solution développée soit efficace et optimale aussi bien en termes de temps d'exécution qu'en consommation de ressources.

A. Modification de la plateforme Android : Une approche basée sur la modification du système Android offre plusieurs avantages. Celle-ci ne nécessite pas la réécriture des applications à chaque fois que ces dernières sont téléchargées par l'utilisateur. De plus, cette approche est en mesure d'intégrer dans les politiques de sécurité des informations système inaccessibles par les applications et permet le contrôle du flux d'informations circulé dans l'ensemble de la plateforme. Cependant, cette approche contient un nombre d'inconvénients. En effet, le fonctionnement des solutions basées sur cette approche requiert des droits administratifs, ce qui implique la nécessité de rooter le périphérique. Cette opération pourrait faire face aux droits d'utilisation du périphérique, ce qui engendre la perte de la garantie de l'utilisateur. De plus, il existe différentes solutions permettant d'avoir plusieurs versions du Framework Android. Dans ce cas, des problèmes de compatibilité peuvent faire surface entre les applications et le système installé, ainsi qu'entre les Frameworks et les types de périphérique. De plus, les contrôles de sécurité peuvent être limités, puisque les applications sont limitées aux politiques de sécurité prises en charge par le

Framework Android modifié. Une approche basée sur la réécriture des applications a donc été adoptée dans cette étude.

B. Décompilation en code source Java : Les développeurs des applications Android programment leurs applications en langage Java, qui sera compilé dans une première phase en code byte Java, puis en code byte Dalvik. La machine virtuelle Java existe depuis beaucoup plus longtemps que la machine virtuelle Dalvik, ce qui implique que les recherches et les outils permettant la décompilation en code source Java sont plus matures. Pour profiter de cette maturité, il a été envisagé dans cette étude de décompiler les applications Android en code source Java afin d'effectuer le traitement nécessaire par la solution développée. Cependant, la machine virtuelle Dalvik est différente de la machine virtuelle Java, puisqu'elle est basée sur les registres et non pas sur les piles. Plusieurs outils comme dex2jar (Pxb, 2015) et ded (Lab, 2014) permettant la conversion du code byte Dalvik en code byte Java. Cependant, la phase de compilation du code byte Java en code byte Dalvik cause la perte de plusieurs informations. Cette perte d'informations peut affecter la phase de recompilation en code byte Dalvik. Dans cette étude il a été opté pour une solution basée sur la décompilation des applications Android en code Smali. Celui-ci étant un code byte intermédiaire. Cette alternative est plus complexe, mais offre un taux de réussite beaucoup plus élevé lors de la phase de recompilation.

C. Nature du Framework de réécriture d'application : Nous avons envisagé deux approches concernant l'emplacement de notre Framework. Une première qui se présente en un serveur distant, tournant sous le système d'exploitation Linux et permet tout aussi bien la réécriture des applications que l'envoi de ces dernières à l'utilisateur. Et une deuxième alternative où le Framework de réécriture est implémenté comme étant une application Android.

   La première alternative a pour avantage de ne pas consommer les ressources du périphérique de l'utilisateur, tandis que la deuxième approche ne nécessite aucun transfert de données vers une partie extérieure. Et afin de tirer avantage des deux

approches et de proposer plus de possibilités à l'utilisateur, les deux variantes ont été implémentées.

D. L'intégration du contrôleur d'applications : La solution présentée dans cette étude est basée sur un contrôleur permettant la gestion des politiques de sécurité. Pour ce faire, deux approches sont envisagées, une première permettant l'intégration du contrôleur dans les applications Android. Dans ce cas, le code de contrôle sera injecté dans le code de l'application à réécrire. De plus, cette méthode ne nécessite pas d'IPC (*InterProcess Communication*) entre l'application en question et le contrôleur. Cependant, cette alternative a été rejetée préférant la deuxième approche qui, elle, se base sur un contrôleur tierce partie. Le contrôleur tierce partie offre l'avantage de gérer plusieurs applications en parallèle. Contrairement à l'approche basée sur l'injection du code de contrôle dans les applications, cette deuxième approche offre l'avantage de lutter contre la collaboration entre multiples applications pour la création d'un contexte malicieux. Une politique de sécurité peut donc être définie afin de gérer la complicité d'un ensemble d'applications. De plus, cette alternative permet une gestion plus flexible des politiques de sécurité grâce à une interface utilisateur

3.4 Architecture globale de la solution proposée

Le présent travail de recherche comporte deux principaux axes, le premier étant un Framework permettant la réécriture de l'application en injectant les portions de code nécessaire, puis un second axe où l'élaboration d'un contrôleur d'application permettant la gestion sécuritaire de l'ensemble d'applications exécutées sur le périphérique est effectuée. L'utilisateur fournit donc le fichier APK de l'application à réécrire et qui va passer par plusieurs étapes afin d'obtenir une version réécrite de l'application capable de communiquer avec le contrôleur. La figure 3.1 montre une représentation très générale de la solution proposée.

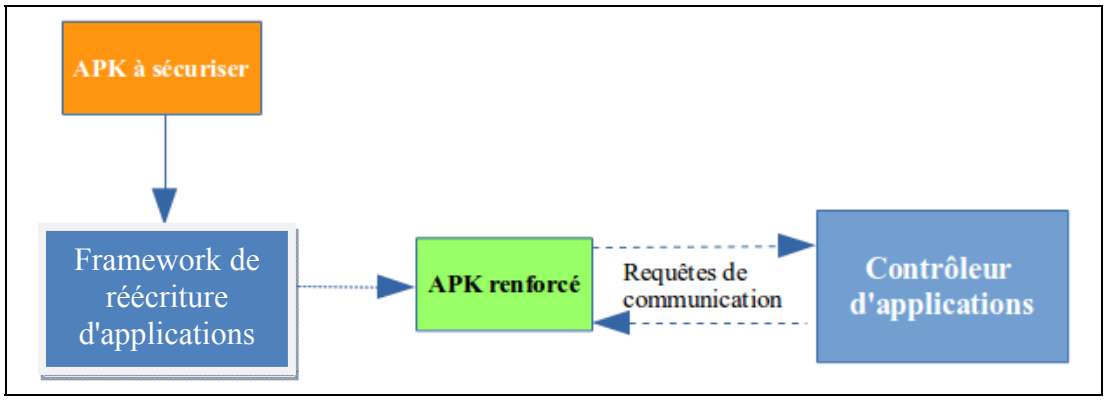

Figure 3.1 Schéma général de la solution proposée

3.5 Principe de réécriture des applications Android

La première partie de recherche, consiste en la réécriture des applications Android. Le but est d'exiger un contrôle sur chaque tentative d'appel à une méthode API par l'application. Ce contrôle se modélise dans la détection des appels aux méthodes API, ainsi que l'envoi des informations nécessaires concernant ces appels au contrôleur d'application, afin qu'il puisse prendre les décisions adéquates sur l'exécution des actions demandées en temps réel.

L'idée principale est, comme le montre la figure 3.2, de rajouter des instructions permettant la communication avec le contrôleur avant chaque appel à une méthode API dans le code de l'application à réécrire. La nature des instructions à rajouter ainsi que leurs emplacements vont être expliqués un peu plus loin dans la suite de ce document.

L'utilisateur doit donc fournir le fichier APK de l'application à sécuriser. Ce dernier va être traité via plusieurs étapes afin d'ajouter les modifications nécessaires. Et afin de permettre ce mécanisme, un Framework se chargeant de la réécriture des applications est proposé.

En premier lieu, le Framework de réécriture conçu dans cette étude est introduit. Celui-ci permet l'automatisation du processus d'injection de code.

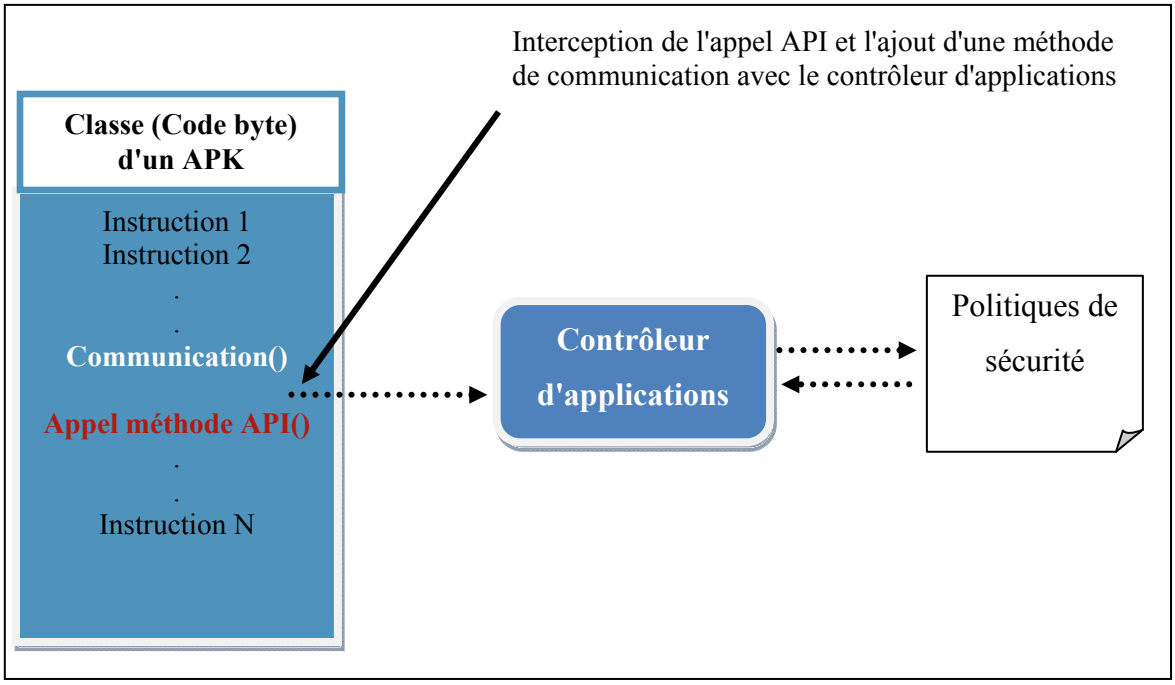

Figure 3.2 L'ajout des méthodes de communication avec le contrôleur

3.6 Framework de réécriture des applications

Dans cette première partie de la présente étude, un Framework permettant la réécriture des applications Android est proposé. Celui-ci a donc pour objectif de traiter les fichiers APK des applications disponibles dans le périphérique de l'utilisateur. La difficulté ici est que le Framework n'a pas accès au code source des applications et doit appliquer la phase de réécriture sur le code byte.

L'algorithme permet donc la localisation des points d'insertion des portions de code permettant la communication avec le contrôleur ainsi que l'injection automatique de ces derniers. Le fichier APK introduit par l'utilisateur va donc subir plusieurs modifications en passant par plusieurs étapes représentées dans la figure 3.3.

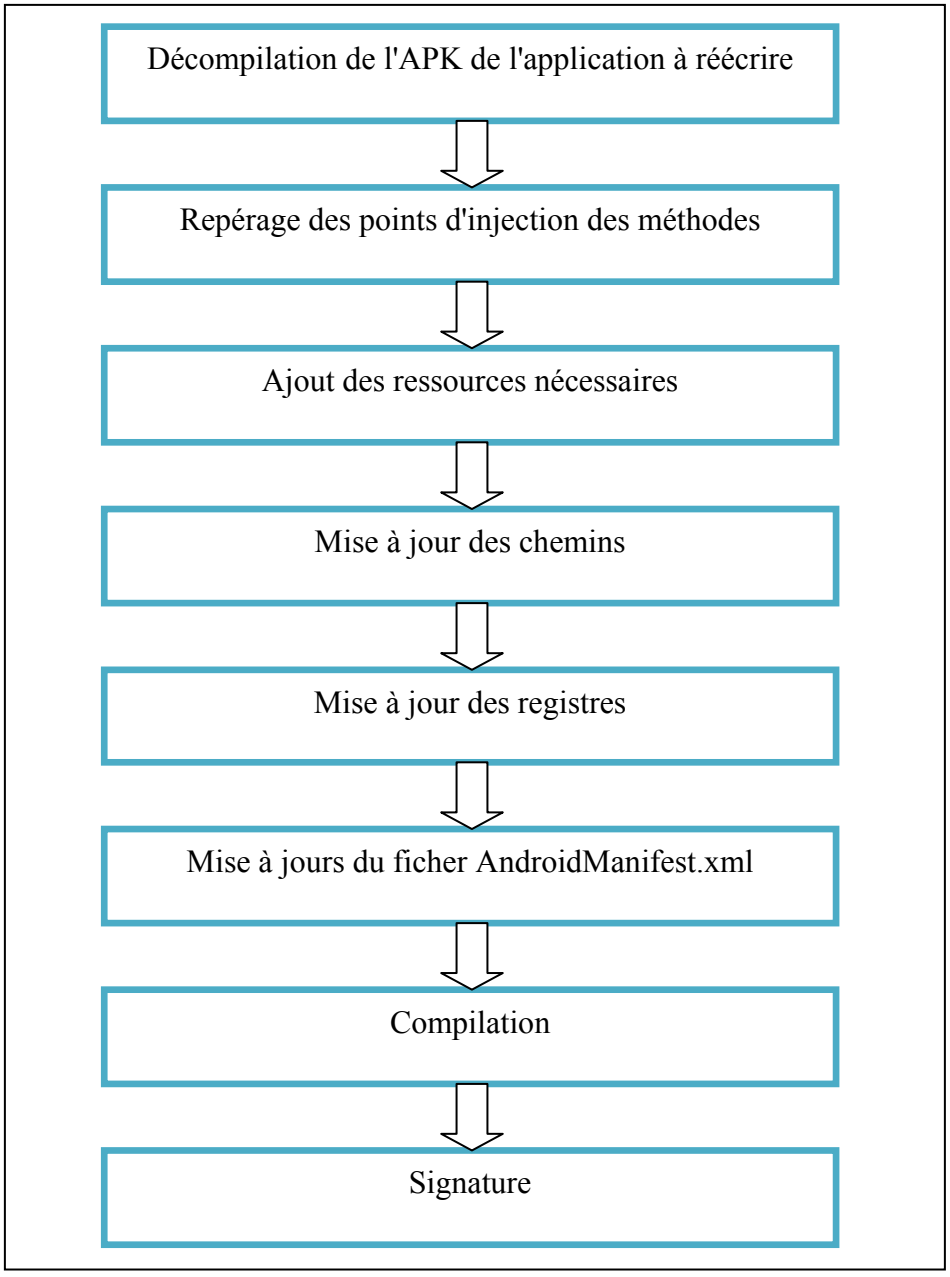

Figure 3.3 Étapes de Renforcement des applications

3.6.1 La nature du Framework de réécriture d'applications

Une fois qu'une application est téléchargée par l'utilisateur, cette dernière doit être réécrite par notre Framework de réécriture d'applications, pour qu'elle puisse communiquer avec le contrôleur. Et afin d'offrir à l'utilisateur un moyen de réécriture pratique et efficace, nous avons implémenté deux variantes du Framework. La première approche se représente en une application Android tierce partie. Cette application est développée en Java et peut être installée dans un périphérique tournant sous le système d'exploitation Android. L'utilisateur peut donc à l'aide de cette application , introduire le fichier APK original de l'application à réécrire, qui va donc être passé par les étapes représentées dans la figure 3.3 afin d'obtenir un APK pouvant communiquer avec le contrôleur. À la fin de la phase de réécriture, le nouveau APK peut être installé directement grâce au Framework de réécriture d'applications.

Cette approche offre à l'utilisateur un moyen de réécriture rapide. Mais elle fait recours aux ressources du périphérique puisqu'elle est installée comme étant une application Android. Nous avons donc envisagé une deuxième alternative permettant à l'utilisateur la réécriture des applications sans l'installation du Framework. Notre deuxième approche se représente donc en une application distante, tournant sur un serveur web. L'utilisateur accède donc à un site web qui lui permet d'introduire son application, et de télécharger l'APK réécrit.

3.6.2 Modification du code des applications

Les développeurs des applications Android écrivent leurs codes sources en java. Les compilateurs sont par la suite utilisés pour convertir les codes sources en codes exécutables dalvik (.Dex) afin qu'ils puissent être exécutés par la machine virtuelle dalvikvm. La façon idéale d'apporter des modifications à l'application est donc de modifier le code source, puis le recompiler en un nouveau exécutable dalvik. Mais malheureusement ce mécanisme n'est possible que pour les AOSP Roms (LEE, 2013). Dans le cas des OEM Roms (LEE, 2013) l'utilisateur n'a pas accès au code source, tout ce dont l'utilisateur a accès est le fichier exécutable dalvik, qui n'est pas lisible à l’œil nu, d’où la nécessité de convertir les fichiers exécutables dalvik en un code byte intermédiaire Smali (voir chapitre 1), qui offre une

lecture plus compréhensible tout en supportant les modifications. Cette conversion est appelée bachSmaling.

3.6.3 Décompilation des applications

La première étape réalisée par notre Framework de réécriture des applications est donc l'adaptation du code byte des applications afin de permettre les modifications. Une phase de décompilation du fichier APK de l'application à réécrire est donc nécessaire. Celle-ci permet l'extraction des fichiers Smali, supportant les modifications. Cette opération est réalisée en faisant recours à l'outil APKTOOL. Le fichier APK introduit par l'utilisateur, va donc être décompilé et les fichiers Smali générés vont être parcourus un par un afin de subir les modifications nécessaires qui sont principalement des portions de code permettant la communication avec le contrôleur.

3.6.4 Détection des points d'injections

Pour que le Framework puisse injecter les portions de code nécessaire permettant la communication avec le contrôleur d'applications, les points d'insertion des instructions à rajouter doivent être localisés. Dans cette étude, et comme ci-avant mentionné, l'intérêt est porté principalement aux méthodes faisant partie des bibliothèques Android et, afin de permettre le contrôle sur ces méthodes, tous les appels aux méthodes API dans les fichiers Smali de l'application sont ciblés. Chaque fichier Smali de l'application va donc être inspecté afin de localiser les points d'injections des communicateurs.

D'abord, tous les appels aux méthodes contenus dans le code Smali sont ciblés, puisque les méthodes API sont appelées de la même manière que les autres méthodes, puis une méthode de sélection est appliquée afin de n'extraire que les méthodes API. Une fois que toutes ces méthodes sont localisées, l'injection des portions de code nécessaire est entamée. Toutefois, avant d'expliquer la procédure de tri adoptée dans la présente étude, la nature des méthodes dans le langage Smali est, tout d'abord, introduite.

3.6.5 Invocation des méthodes dans le langage Smali

Dans le code Smali, il existe trois principaux types de méthodes qui peuvent être cités, et qui sont : les méthodes statiques, les méthodes d'instanciation, et les constructeurs.

3.6.5.1 Les méthodes statiques

Les méthodes statiques sont appelées dans le code Smali par les deux instructions : Invoke-static et invoke-static/range. Et pour que la machine virtuelle puisse invoquer la méthode demandée, certains paramètres sont nécessaires. Ces derniers sont écrits en code Smali sous le format suivant :

(p1), *Lpackage/Name/ObjectName; →MethodName(parameter1,….,parameterN)Z*

Où :

- Lpackage : représente le package.
- Name : est le nom de l'application.
- ObjectName : est le nom de l'objet qui fait recours à la méthode demandée.
- MethodName : est le nom de la méthode sollicitée.
- Parameter1….ParameterN : sont les types des paramètres attendus par la méthode demandée. Exemple pour le type String, le paramètre est sous le format : Ljava/lang/String.
- Z : représente le type de l'objet retourné par la méthode.
- (p1): représente le registre qui contient la valeur du paramètre 1.

De plus, un registre est réservé pour chaque paramètre attendu par la méthode sollicitée.

3.6.5.2 Les méthodes d'instanciation

Les méthodes d'instanciation sont appelées dans le code Smali par l'instruction invoke-virtual. Ces méthodes sont sollicitées par le même format que les méthodes statiques, mais en plus des registres nécessaires pour contenir les paramètres attendus par la méthode, un

registre de plus est indispensable. Celui-ci va contenir la référence à l'objet dans lequel la méthode va être appelée.

3.6.5.3 Les constructeurs

Les constructeurs sont appelés par le même format que les autres types de méthodes, avec le mot clé : Invoke-direct. La différence concernant les registres est que dans le cas des constructeurs, le premier registre est réservé comme référence à la nouvelle instance d'objet qui va être créé.

3.6.6 Pistage des méthodes API

En premier lieu, tous les fichiers Smali générés par la phase de décompilation vont être parcourus. Une recherche est faite dans chaque fichier en utilisant les mots clés d'invocation de chaque type de méthode. Cette phase permet de repérer tous les appels aux méthodes contenus dans l'application. La phase suivante est la phase de sélection et qui va permettre d'extraire le sous-ensemble des méthodes API à partir de l'ensemble global des méthodes retrouvées.

L'algorithme développé dans cette étude fait référence à un fichier qui contient les signatures des appels aux méthodes API qui y sont traitées. Lesdites signatures sont constituées du type de la méthode, le package, le nom de la méthode ainsi que de la classe contenant la méthode.

Dans la présente étude, on s'intéresse aux 20 méthodes API les plus fréquentes dans les applications malicieuses, ces dernières sont illustrées dans la figure 3.3 (Aafer, Du et Yin, 2013). Où pour un ensemble d'applications, la barre rouge représente le pourcentage d'utilisation de ces méthodes API par les applications malicieuses, et la barre blanche représente leurs disponibilités dans des applications bénignes.

Le fichier de référence aux méthodes API contient donc les signatures de ces méthodes. Il est important de noter que notre fichier de référence de méthode API est flexible. En effet, celui-ci pouvant accepter toute modification ou insertion de nouvelles signatures de méthode API.

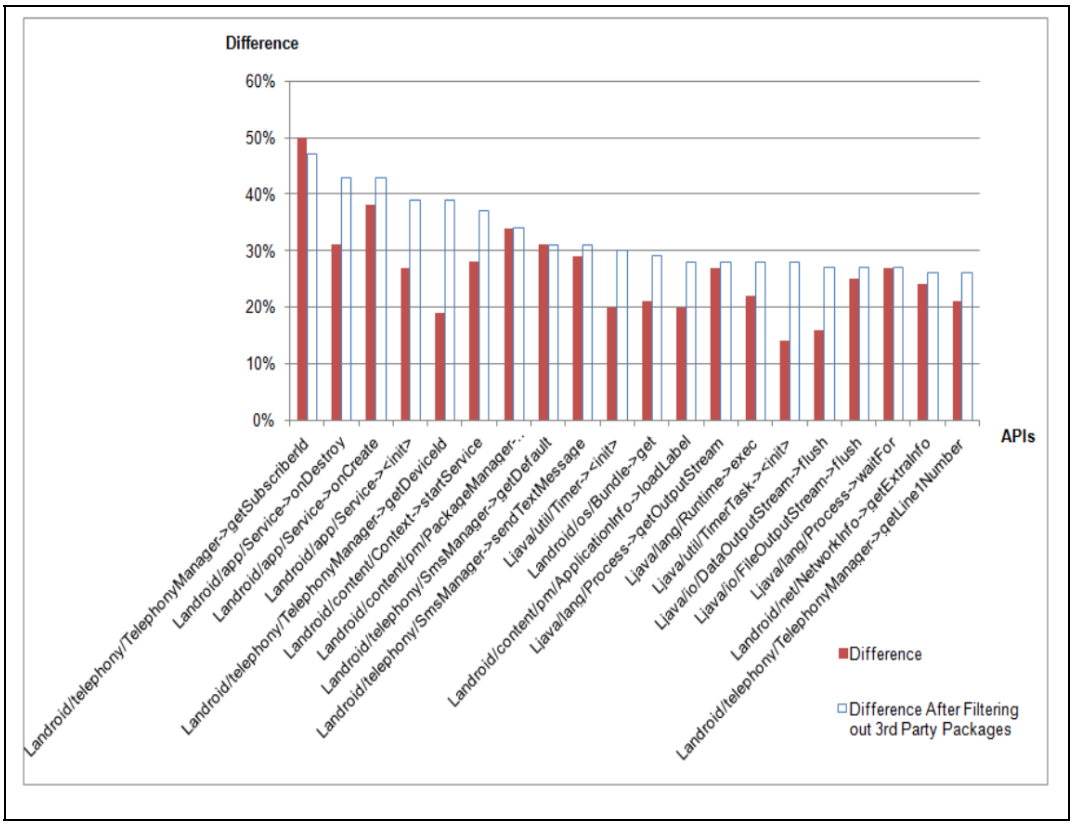

Figure 3.4 Top 20 méthodes API les plus fréquentes dans les applications malicieuses. Tirée de (Aafer, Du et Yin, 2013)

Une fois donc que l'ensemble des méthodes contenues dans l'application sont repérées, cette liste est comparée à l'ensemble des signatures contenues dans le fichier de référence des méthodes API. Ainsi, à la fin de cette opération, l'algorithme développé dans cette étude ne garde trace que des méthodes API contenues dans l'application ainsi que de leur emplacement dans le code byte.

La prochaine étape est d'injecter les portions de code qui permettent la communication avec le contrôleur avant chaque appel à une méthode API repérée par l'algorithme.

3.6.7 Ajout de ressources nécessaires à l'application

Afin que l'application renforcée par notre algorithme puisse contrôler l'exécution des méthodes API, cette dernière doit faire recours au contrôleur d'applications, en fournissant

plusieurs paramètres permettant à ce dernier de prendre la décision sur l'exécution de la méthode sollicitée. Cette communication entre l'application réécrite et le contrôleur est assurée par plusieurs méthodes. Ces dernières, ayant été mises en œuvre dans cette étude, sont contenues dans un ensemble de classes. Cependant, avant de présenter les classes qui sont rajoutées à l'application réécrite, la nature de la communication est d'abord présentée.

3.6.7.1 La nature de la communication

La communication entre les applications réécrites par le Framework développé dans cette étude et le contrôleur d'applications est principalement basée sur les Intents implicites (voir chapitre 1). En effet, au moment où l'application doit solliciter le contrôleur, un nouveau Intent sera créé par l'application réécrite, comme le montre la figure 3.5. Cet Intent va faire communiquer certaines informations au contrôleur d'application. Ces dernières sont principalement les paramètres de la méthode API à invoquer, plus deux arguments qui permettent au contrôleur d'identifier l'application qui le sollicite, ainsi que le type de la requête.

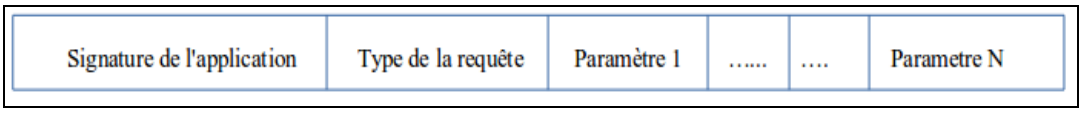

Figure 3.5 Paramètres envoyés via l'intent

La Signature de l'application est composée du package ainsi que du nom de l'application. Cet argument permet au contrôleur d'identifier l'application qui l'interroge. Celui-ci va, par la suite, garder trace de cette signature afin qu'il puisse cibler cette application au moment de la réponse et ce, en renvoyant cette signature qui va être interceptée par l'application concernée.

Le contrôleur d'application de la présente étude traite deux types de requêtes. Ces dernières étant l'exécution des méthodes API et l'exécution des applications. L'argument type de la requête permet donc au contrôleur d'identifier la nature de la requête.

Les autres arguments communiqués par les Intents sont les paramètres attendus par la méthode API. Lesdits paramètres sont donc envoyés au contrôleur afin qu'il puisse identifier le comportement de l'action demandée.

3.6.7.2 Les arguments des Intents

Les Intents sont donc créés et utilisés dans cette étude dans deux situations. La première étant lors de l'envoi d'une requête par l'application réécrite vers le contrôleur tandis que la deuxième est lors de l'envoi de la réponse par le contrôleur. La différence entre les deux requêtes est donc précisée par le champ Action. La figure 3.6 illustre l'ensemble de champs constituant l'intent. Où, le champ données représente le type de données circulées par l'intent,. Dans le cas de la présente étude, un tableau de *string* contenant les paramètres ci-avant cités est envoyé. Le champ Extras contient les données à circuler. Quant au champ *Flags* contient un ensemble de variables que nous utilisons afin de modifier le comportement de l'intent en cas d'erreurs.

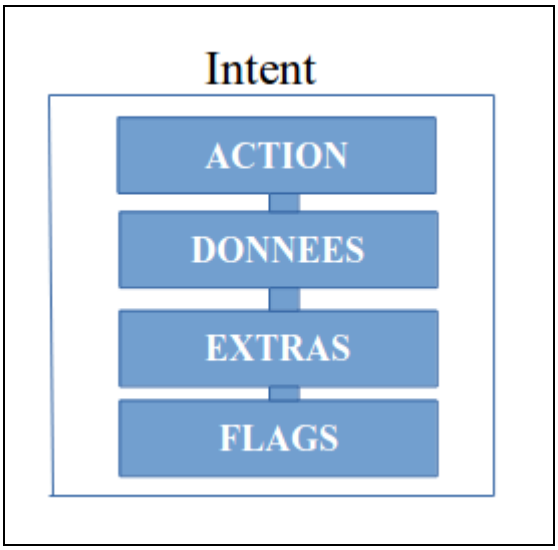

Figure 3.6 Champs composant un intent

3.6.7.3 Les actions des intents

Le fonctionnement des intents implicites nécessite la définition des actions. Ces dernières permettent la définition du type de traitement nécessaire aux données circulées par l'intent. Pour cela, deux actions ont été créées dans cette étude. Une première appliquée sur les intents utilisés pour envoyer les données vers contrôleur, et une deuxième appliquée sur les intents qui servent à envoyer les données vers les applications réécrites. De cette manière, quand une application réécrite envoie une requête au contrôleur, ce dernier saura que cette dernière lui est destinée (grâce à l'action de l'intent). Par conséquent, le contrôleur traitera cette requête et enverra la réponse via un intent. L'application concernée, à son tour, récupérera la réponse grâce à l'action appliquée sur l'intent ainsi que la signature de l'application envoyée par le contrôleur.

De plus, pour que les applications réécrites puissent intercepter les bonnes actions, des filtres doivent être créés dans les fichiers manifestes de ces dernières. Cette création des filtres est effectuée par l'algorithme développé dans la présente étude.

3.6.7.4 Le filtrage des intents

Quand une réponse est envoyée par le contrôleur, les applications réécrites doivent récupérer la donnée communiquée par l'intent. Ce mécanisme est, comme ci-avant expliqué, assuré par l'utilisation des actions. Mais pour que Android communique l'intent à la bonne application, chaque application réécrite doit déclarer un filtre dans son fichier manifest. Ce filtre permet de préciser à Android que cette application peut gérer un type précis d'actions. Notre algorithme va donc modifier le fichier manifest de l'application réécrite, afin de rajouter le nœud <intent-filter>. Ce dernier va contenir l'action <This.From_Monitor> que préalablement définie. Celle-ci permet à Android de comprendre que cette application peut gérer les intents reçus par le contrôleur.

3.6.8 La nature des ressources à rajouter

Une fois que les appels aux méthodes API contenues dans l'application sont repérés par l'algorithme développé dans cette étude, ce dernier doit injecter les portions de code nécessaires et ce, afin de contacter le contrôleur avant chaque tentative d'exécution d'une méthode API. Pour ce faire, plusieurs classes écrites en langage Smali (byte code) ont été développées dans cette étude. La collaboration entre ces classes permet la communication des arguments nécessaires au contrôleur d'applications, l'attente de la réponse et la prise de décision selon la réponse reçue. Par la suite, les classes Smali à rajouter seront présentées et ce, après avoir d'abord expliqué le mécanisme général de la communication entre l'application réécrite et le contrôleur d'application.

3.6.9 Mécanisme de communication

La figure 3.7, illustre le mécanisme de communication entre le contrôleur et les applications réécrites par l'algorithme développée dans la présente étude. Ce dernier va donc ajouter un appel à une méthode de contrôle avant chaque appel à une méthode API. Plus de détails seront donnés dans le présent mémoire quant à ces méthodes. Mais nous illustrons dans la figure 3.7 un exemple de la méthode de contrôle requête(), contenue dans la classe Smali *methods*, développées dans cette étude. Cette méthode est donc exécutée juste avant l'appel de la méthode API. Il est à noter qu'au moment de cette exécution, cette dernière va créer une nouvelle activité permettant de gérer la communication avec le contrôleur. Ladite activité est illustrée par le fichier Smali communicateur et fait recours à une autre classe Smali appelée *BroadCastReceiver.* Cette dernière implémente les méthodes nécessaires à la gestion des intents, qui permettent de les envoyer et les intercepter.

L'activé Communicateur va donc générer un nouvel intent, et l'envoyer au contrôleur d'application grâce à la classe *BroadCastRecevier*, et attendre la réponse. Une fois que cette dernière est reçue, l'activité Communicateur consulte la réponse afin de prendre la décision adéquate.

L'algorithme développé dans la présente étude va donc copier toutes ces classes ayant été développées en Smali dans le nouveau dossier qui contient la version décompilée de l'application à réécrire.

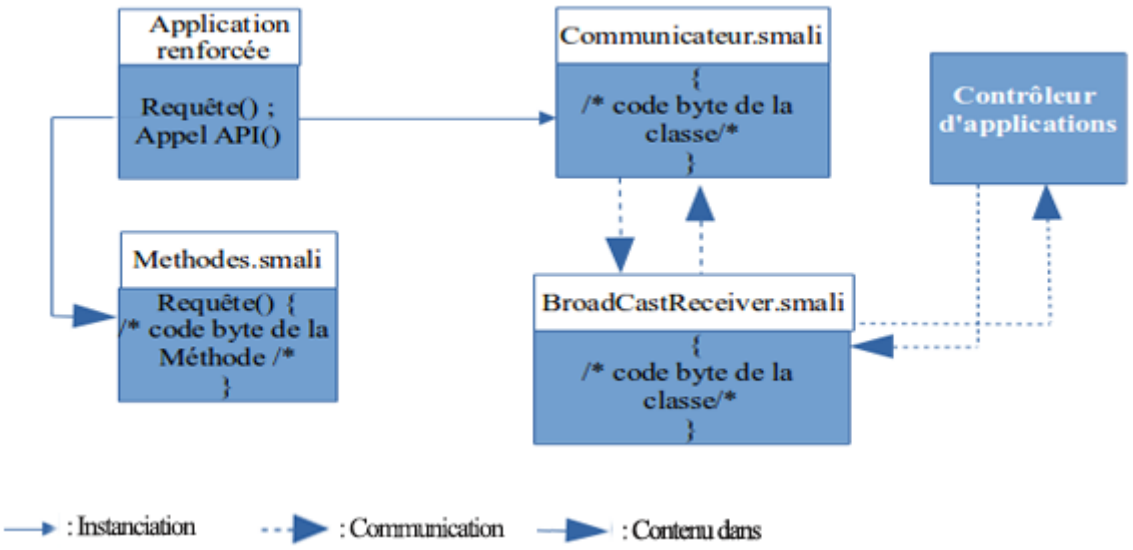

Figure 3.7 Mécanisme de communication contrôleur-applications réécrites

3.6.9.1 La classe Méthodes.smali

Cette classe contient donc les méthodes qui permettent d'initier la communication avec le contrôleur d'application. L'algorithme injecte l'appel à une des méthodes contenues dans cette classe. Deux méthodes principales existent :

Une première qui permet la gestion de l'exécution de l'application. L'appel à cette dernière est ajouté par l'algorithme dans l'activité main de l'application à réécrire. La première instruction à exécuter par l'application serait donc l'appel à cette méthode. Celle-ci prend en argument le type de la requête «RunApplication » ainsi que la signature de l'application.

Le type de cette requête permet au contrôleur de savoir que l'application demande le droit d'être exécutée. Alors que la signature de l'application permet au contrôleur d'identifier l'application en question. Une fois que le contrôleur reçoit ce type de requête, une

vérification est réalisée en faisant recours aux politiques de sécurité introduites par l'utilisateur ainsi, une réponse sera envoyée.

La deuxième méthode contenue dans l'application permet la gestion de l'exécution des méthodes API, l'appel à cette méthode est injecté avant chaque appel à une méthode API. Cette méthode prend en arguments le type de la requête « RunAPI ». Celui-ci permet au contrôleur de savoir que l'application tente d'exécuter une méthode API, la signature de l'application, la signature de la méthode API à exécuter, et un ensemble d'arguments qui représente les arguments de la méthode API. Notre algorithme identifie donc les arguments passés à la méthode API, et les envoi au contrôleur via notre méthode de contrôle.

3.6.9.2 La classe Communicateur.smali

Cette classe est une activité qui hérite de la classe *ActionBarActivity*. Elle est instanciée par les méthodes de contrôles contenues dans la classe *Méthodes.smali* et permet de gérer la communication avec le contrôleur. Cette classe permet donc d'instancier l'intent qui contient la requête à envoyer au contrôleur, d'attendre la réponse et de prendre la décision adéquate selon la réponse reçue.

Trois types de réponses sont identifiées dans la présente étude, à savoir :

- Accès refusé : Si ce type de réponse est reçu, l'activité va forcer l'application à se terminer.
- Accès autorisé : Ce type de réponse permet à l'application ou à l'API de s'exécuter. L'activité va alors se fermer, en débloquant l'accès à l'application pour s'exécuter normalement.
- Accès temporairement refusé : Si le contrôleur décide que la ressource demandée par l'application est temporairement inaccessible, cette réponse sera envoyée à l'application. Cette dernière va être interceptée par l'activité qui, à son tour, va renvoyer une deuxième requête au contrôleur après l'écoulement d'un laps de temps.

Afin que cette classe puisse gérer les communications avec le contrôleur, l'activité doit être exécutée dans un nouveau *Thread*. De cette manière, l'application peut être temporairement suspendue pendant le temps de contrôle tout en communiquant avec le contrôleur grâce au nouveau *Thread*. Cette communication est effectuée en faisant recours à la classe *BroadCastReceiver.smali*.

3.6.9.3 La classe BroadCastReceiver.smali

Cette classe permet la gestion des intents, en héritant de la classe *BroadCastReceiver* et en implémentant la méthode *OnReceive*. Cette méthode a été ré-implémentée afin qu'elle puisse identifier les intents envoyés par le contrôleur. Ces derniers, étant destinés à l'application en question.

La méthode va alors récupérer la signature de l'application envoyée par le contrôleur et la comparer avec la signature de l'application en question. Si les deux signatures sont identiques, la réponse envoyée par l'intent sera récupérée et envoyée à l'activité *Communicateur.Smali* afin d'effectuer le traitement nécessaire.

3.6.10 Injection des ressources dans l'application

Les classes Smali ci-avant représentées, sont rajoutées à l'application à réécrire. Pour ce faire, l'algorithme va créer un nouveau package dans le répertoire contenant les fichiers Smali de l'application. Ainsi, les ressources nécessaires vont être copiées dans ce nouveau package et ce, tout en les séparant des autres classes Smali de l'application. La figure 3.8 illustre l'insertion du package de contrôle dans une application décompilée.

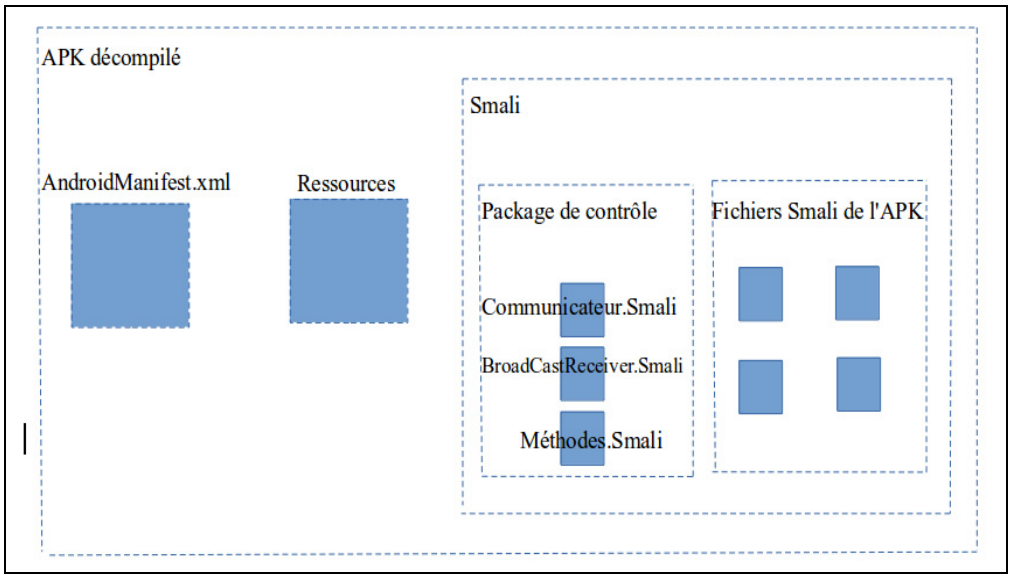

Figure 3.8 Insertion du package de contrôle dans l'application

3.6.11 Modification des chemins

Après avoir copié les ressources nécessaires dans l'application décompilée, l'algorithme développé dans la présente étude va procéder à la modification des chemins dans les fichiers Smali rajoutés. Les classes développées en Smali sont des classes standards, prêtes à être copiées dans n'importe quelle APK. Cependant, une étape très importante est à considérer Celle-ci s'agissant de faire le lien entre l'application et les ressources copiées.

L'algorithme va, par conséquent, adapter les appels de contrôles rajoutés dans les fichiers Smali de l'application à réécrire afin que ces derniers parviennent à appeler les méthodes situées dans le package de contrôle ayant été rajouté. Les signatures des méthodes dans les instructions d'invocation vont alors être modifiées. Le suivant appel de la méthode requête() est pris à titre d'exemple:

invoke-static (p1), LCOM/Methods;-> requête()V

Celui-ci peut être adapté à une application spécifique sous le format :

invoke-static (p1), LCOM/Facebook/w/z/ComAPP/Methods;-> requête()V

De plus, plusieurs chemins dans les classes standards rajoutées doivent être modifiés. L'algorithme développé dans cette étude parcourt alors les classes Smali rajoutées dans l'application réécrite et adapte les chemins selon cette dernière.

3.6.12 Adaptation des registres

Une des phases critiques de la phase de réécriture dans cette étude est celle des adaptation des registres. En effet, les applications Android sont développées de sorte qu'elles puissent fonctionner sur une machine Dalvik. Or, cette machine est basée sur l'utilisation des registres. Les appels aux méthodes en langage Smali spécifient donc les registres à utiliser. Lors de l'exécution d'une application, les données nécessitées par cette dernière peuvent circuler d'un registre à un autre.

L'ajout des appels aux méthodes de contrôles adoptées dans cette étude nécessite l'ajout des registres afin que lesdites méthodes puissent fonctionner. L'algorithme développé a alors pour tâche d'analyser les registres utilisés dans l'application afin de repérer le registre attendu par chaque appel.

Si, par exemple, l'appel suivant est rajouté dans le code de l'application :

invoke-static (p1), LCOM/Methods;-> requête(LAndroid/content/context;)V

Il est remarqué que la méthode requête utilise une variable contenant le contexte de l'application. Ledit contexte doit alors être fourni par le registre p1. Si cette ligne de code est rajoutée dans l'application, il faudra s'assurer que le registre p1 représente bien le contexte de l'application Si cela n'est pas le cas, l'algorithme exécutera une analyse du fichier Smali en question afin de récupérer le registre correspondant au contexte de l'application.

Le problème avec l'utilisation des registres dans le langage Smali est, en fait, que les applications ont tendance à circuler les données via plusieurs registres En effet, une valeur A enregistrée dans un registre p0, peut à tout moment être enregistrée dans un autre registre afin de libérer le registre p0 et ainsi de suite. L'algorithme développé permet donc le pistage de

ces différentes affectations de variable afin de repérer les registres contenant les variables ciblées.

3.6.13 Modification du fichier AndoidManifest.xml

Le fichier *AndroidManifest.xml* est nécessaire pour le bon fonctionnement de l'application, il permet de décrire tous les composants de l'application (activités, BroadCast, Receivers…), les permissions, les bibliothèques utilisées dans l'application, etc. Celui-ci fournit donc les informations à propos de l'application au système Android.

L'algorithme développé dans la présente étude affecte plusieurs modifications à l'application lors de la phase de réécriture. Certaines de ces modifications doivent être déclarées dans le fichier *AndroidManifest.xml* de l'application. L'algorithme permet donc la modification automatique du fichier manifest de l'application afin d'apporter les modifications nécessaires Ces dernières sont principalement :

- La déclaration de l'activité Communicateur implémentée par la classe *Communicateur.Smali*. L'algorithme permet aussi d'informer le fichier AndroidManifest.xml que cette activité doit s'exécuter dans un nouveau Thread.
- La déclaration du BroadCastReceiver implémenté par la classe *BroadCastReceiver.smali*
- L'ajout du filtre « *This.from_Monitor* » qui permet à l'application de reconnaître les actions des intents envoyés par le contrôleur.

3.6.14 Compilation de l'application

Une fois que toutes les modifications nécessaires sont apportées à l'application, cette dernière doit être recompilée afin de générer un nouveau APK. L'algorithme développé dans cette étude procède donc à la compilation de tous les fichiers Smali contenus dans l'application réécrite et ce, en utilisant l'outil APKTOOL.

L'ensemble des fichiers Smali seront alors recompilés afin de générer le fichier *.dex* qui représente l'ensemble de l'application. Les ressources non compilées seront rajoutées à ce dernier afin de générer un nouvel APK représentant la version réécrite de l'application.

3.6.15 Signature de l'application

Android exige que toutes les applications doivent être signées avec un certificat pour qu'elles puissent être installées. Android utilise alors ce certificat afin d'identifier l'auteur de l'application.

La signature de l'application se trouve au sein de son archive APK dans le répertoire WEB-INF. L'algorithme développé dans cette étude permet donc la signature du nouveau APK généré par la compilation de l'application réécrite.

3.7 Le contrôleur des applications

La première partie de la présente étude a été de réécrire les applications Android afin que ces dernières puissent communiquer avec le contrôleur d'application en cas de besoin. Pour cela, un algorithme permettant la modification automatique des applications ainsi que la génération des nouveaux APK réécrits prêt à être installés a été développé.

La deuxième partie de la présente étude est le contrôle d'application Autrement dit, l'implémentation d'une tierce partie permettant le contrôle centralisé des actions effectuées par les applications installées. Pour cela, un contrôleur d'applications capable de prendre les décisions nécessaires a été développé et ce, tout en modifiant le comportement des applications au besoin.

3.7.1 L'authentification des applications

Le contrôleur d'applications développé dans cette étude permet la gestion des applications réécrites installées sur le périphérique. Pour ce faire, chaque application installée doit être enregistrée dans le contrôleur. Ce mécanisme est assuré par le contrôleur lui-même de manière à ce que quand il reçoit une demande d'exécution de la part d'une application (requête « Exécution »), il ira vérifier si l'application est déjà enregistrée dans son catalogue. Si cela n'est pas le cas, le contrôleur procédera à l'enregistrement de cette application tout en

gardant trace de sa signature. De cette manière l'utilisateur peut appliquer ces politiques de sécurité sur l'application en question.

Le contrôleur d'application fait recours à un fichier catalogue qui comporte la liste des applications enregistrées avec leurs signatures au sein du contrôleur. Ce fichier est chargé au lancement du contrôleur dans une structure de donnée de type « tableau » et sera conservé tout au long de l'exécution du contrôleur. Quand l'utilisateur reçoit une requête de type « Exécution », les vérifications seront faites sur la structure de données. Si l'application n'est pas déjà enregistrée, la structure de données et le fichier catalogue seront alors mises à jour. La figure 3.9 illustre un exemple d'enregistrement d'une application au sein du contrôleur.

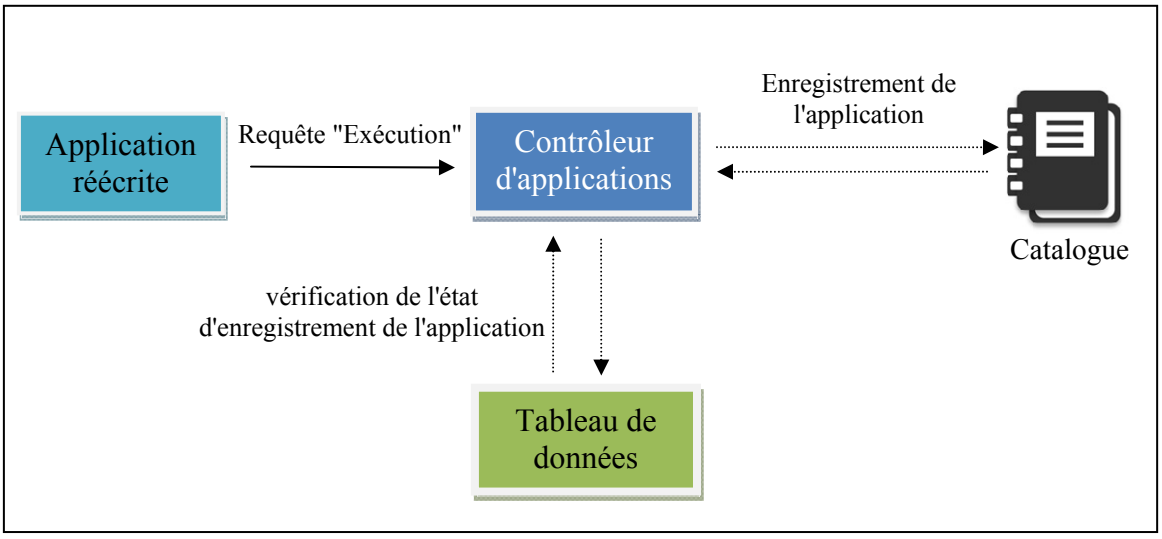

Figure 3.9 Enregistrement d'une application au sein du contrôleur

3.7.2 Contrôle d'exécution des applications

D'un côté, le contrôleur d'applications se base principalement sur des règles de sécurité introduites par l'utilisateur afin de prendre les décisions adéquates pour chaque action effectuée par les applications installées. D'un autre côté, celui-ci offre à l'utilisateur la possibilité d'interdire l'exécution des applications sans définir les règles. L'utilisateur accède donc via l'interface graphique du contrôleur à la liste d'applications installées où il peut définir l'ensemble d'applications à interdire. De cette manière, quand une application, non

autorisée par l'utilisateur, tente de s'exécuter, elle envoie une requête de type « Exécution ». Ainsi, le contrôleur interdira l'exécution de l'application et la force à se terminer.

3.7.3 Les règles de sécurité

Le rôle du contrôleur d'application est de prendre les décisions adéquates pour chaque requête émise par une application réécrite. Le contrôleur traite alors cette requête et envoie la réponse à l'application en question. Cette prise de décision est basée sur un ensemble de règles de sécurité introduites par l'utilisateur. Le contrôleur offre alors une interface graphique permettant à l'utilisateur de définir ses règles.

Il a été opté dans cette étude, pour une solution qui permet la création des règles de sécurité en faisant recours à un ensemble de conditions. Ces dernières permettent à l'utilisateur d'introduire les scénarios d'exécution souhaités.

3.7.3.1 Les conditions

Dans la présente étude, la détermination des conditions est à effectuer par l'utilisateur lui-même. Lesdites conditions servent principalement à la création des règles de sécurité. Dans la présente étude, cinq types de conditions de base sont définis. Celles-ci seront utilisées pour construire des conditions plus générales. Ces conditions de base sont:

- Intervalle de temps : Cette condition permet à l'utilisateur de définir un intervalle de temps pendant lequel un ensemble d'applications, ou des méthodes API peuvent être exécutées.
- Date : La condition date permet de gérer l'exécution des applications ou des méthodes API, dans un intervalle défini par des dates.
- Emplacement : Cette condition permet à l'utilisateur d'empêcher l'exécution des applications ou des méthodes API dans des emplacements définis. Ces emplacements sont basés sur les coordonnées GPS.
- Applications : Cette condition permet la définition d'un ensemble d'applications, en utilisant les opérateurs logiques « OU » et « ET ». Si l'utilisateur veut donc appliquer une

règle de sécurité qui dépend d'un ensemble d'applications, cette condition serait donc sous le format « APP1 ET APP2 ET APP3, etc. ».

- Batterie : Cette condition permet à l'utilisateur de gérer l'exécution des applications ou des méthodes API, tout en prenant compte de l'état de la batterie.

L'utilisateur du contrôleur d'applications développé dans la présente étude, peut alors créer un ensemble de conditions basées sur les cinq types ci-avant cités. L'utilisateur affecte un nom à chaque condition, cette dernière sera enregistrée dans un fichier catalogue en spécifiant son nom et type.

Ledit contrôleur permet à l'utilisateur d'introduire des conditions plus génériques en combinant les conditions déjà créées par l'utilisateur. De cette manière, l'utilisateur est capable de créer des règles qui dépendent de plusieurs types de conditions à la fois. De plus, chaque condition créée (basique ou générique) peut être combinée avec d'autres conditions.

De plus, ce contrôleur d'applications permet la combinaison des conditions en se basant sur les opérateurs logiques "ET" et "OU". L'utilisateur peut alors définir n'importe quel scénario logique. La figure 3.10 illustre des exemples de création de conditions au sein de notre moniteur.

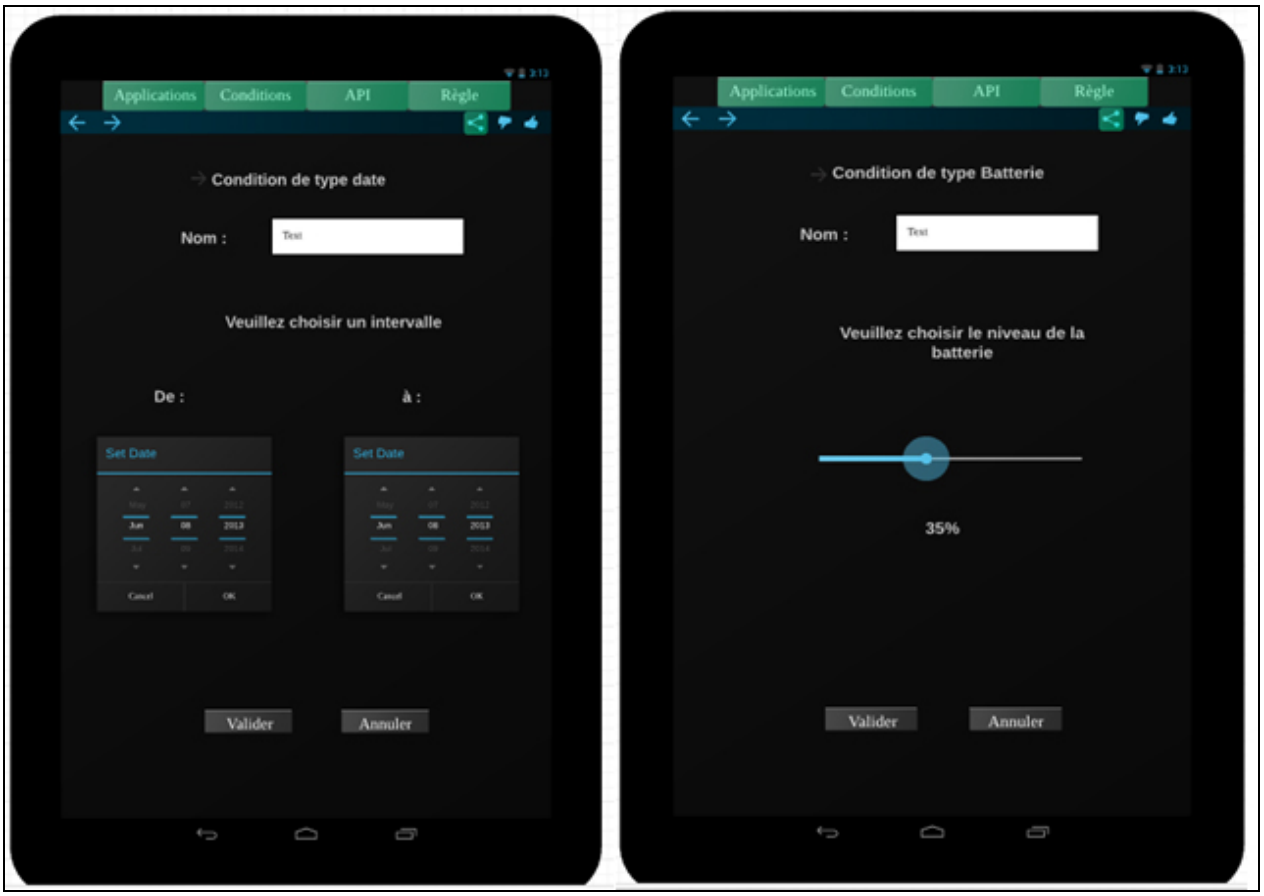

Figure 3.10 Exemples de création de conditions

3.7.3.2 La création des règles de sécurité

Les règles de sécurité sont introduites par l'utilisateur et servent à spécifier le comportement souhaité par l'utilisateur pour chaque application réécrite. Le contrôleur d'applications fait donc recours à ces règles lors de chaque tentative d'exécution d'une action suspecte.

Les règles de sécurité sont principalement définies en utilisant les conditions créées par l'utilisateur. Ces dernières sont combinées afin de définir le scénario souhaité par l'utilisateur. Si, par exemple, l'utilisateur veut appliquer la règle de sécurité suivante :

« Il est interdit d'exécuter un ensemble d'applications si l'utilisateur du périphérique est en réunion ».

Afin que cette règle de sécurité soit créée, l'utilisateur doit, d'abord, procéder à la création des conditions qui définissent la règle. Si par exemple, la réunion est faite deux fois par semaine, de 14h à 15h, chaque lundi au sein de la société. Plusieurs conditions doivent être définies. À savoir :

- C1 : Condition de type intervalle de temps contenant l'heure de début et de fin de la réunion.
- C2 : Condition de type Date contenant la date de la réunion..
- C3 : Condition de type Emplacement contenant les coordonnées GPS de la société où se trouve la salle de réunion.
- C4 : Condition de type applications contenant l'ensemble d'applications, exemple : ((Application 1 ET Application 2) OU Application 3 OU Application 4).

Ces quatre conditions sont prises en compte afin de définir la règle qui permet d'empêcher le scénario défini par l'utilisateur. Il est à noter qu'aucune application ou ensemble d'applications de la règle C4 n'est exécuté si les trois conditions (C1 & C2 & C3) sont vérifiées. La figure 3.11 illustre un exemple de la façon de combiner des conditions au sein du contrôleur d'application.

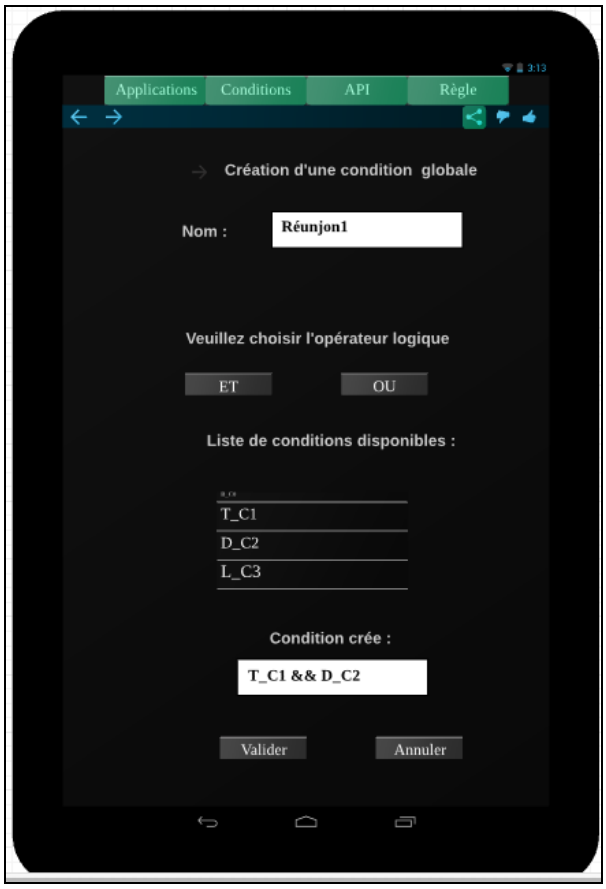

Figure 3.11 Combinaison de conditions

3.7.3.3 L'application des règles de sécurité

Les règles de sécurité peuvent être construites afin de définir des scénarios d'exécution. Ces dernières sont, non seulement, dépendant des applications, mais aussi des méthodes API. Le contrôleur d'applications développé dans cette étude offre une manière flexible de construire lesdites règles en se basant sur des méthodes API.

L'interface du contrôleur d'applications et comme la montre la figure 3.12 permet l'affichage de toutes les applications installées sur le périphérique et sont déjà enregistrées au sein du contrôleur. L'utilisateur peut, dans ce cas, soit, gérer les méthodes API spécifiques à chaque application, soit les gérer à l'ensemble des applications.

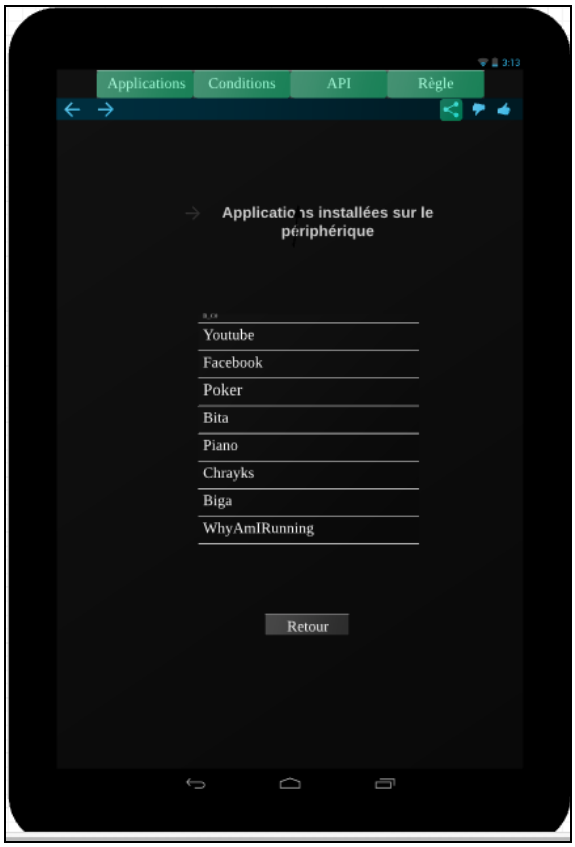

Figure 3.12 Liste d'applications réécrites

Quand l'utilisateur choisit une application spécifique (valable aussi pour l'ensemble des applications), les méthodes API utilisées par l'application seront donc affichées. Cette liste de méthodes est récupérée lors de la phase de réécriture de l'application. Le Framework développé dans cette étude identifie les méthodes API à partir des fichiers Smali, et la liste sera envoyée au moniteur d'application lors de la phase d'authentification de l'application.

L'utilisateur peut interdire l'exécution d'une méthode API par l'application choisie (ou l'ensemble des applications) en désactivant la méthode dans l'interface utilisateur comme illustrée dans la figure 3.13.

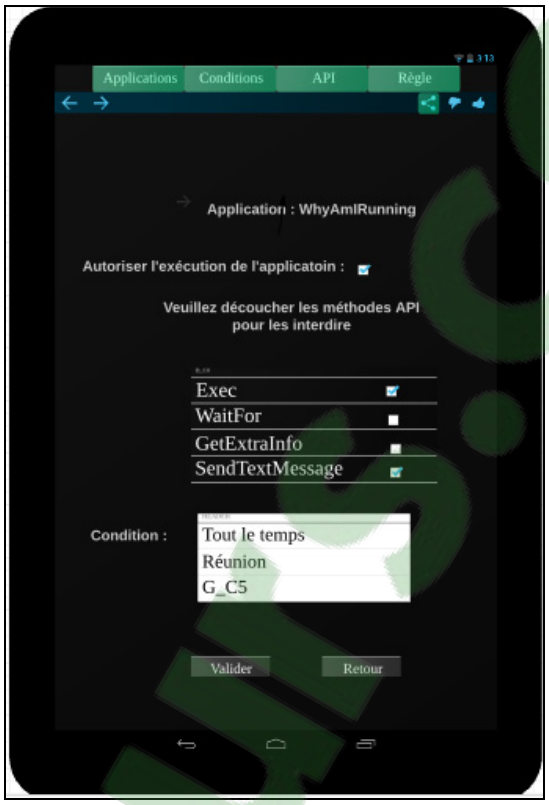

Figure 3.13 Gérer les méthodes API d'une application

Si l'application tente d'exécuter une méthode désactivée par l'utilisateur, le contrôleur d'application empêchera cette action, car la condition « Tout le temps » est par défaut activée. L'utilisateur peut donc personnaliser la règle en changeant la condition « Tout le temps » et en la remplaçant avec une condition spécifique déjà créée.

Si, par exemple, l'utilisateur veut appliquer la règle de sécurité suivante :

« Il est interdit qu'une application puisse enregistrer une séquence vidéo si l'utilisateur du périphérique se trouve en réunion ».

L'utilisateur va donc créer les conditions nécessaires afin de définir cette règle. Ces conditions seront donc combinées afin de définir une condition plus générique qui va être appliquée sur la méthode API d'enregistrement vidéo désactivé par l'utilisateur. De cette manière, si l'utilisateur n'est pas en réunion, le contrôleur d'application autorisera l'exécution

de la méthode API d'enregistrement vidéo par l'application en question (ou par l'ensemble d'applications).

Plusieurs règles peuvent donc être définies, dépendamment des situations définies par les conditions. Ces règles peuvent alors être activées ou désactivées par l'utilisateur selon le besoin.

3.7.3.4 L'exclusion entre les méthodes API

Le contrôleur d'applications offre à l'utilisateur la capacité de définir des méthodes API exclusives à une ou à un ensemble d'applications. L'utilisateur affecte via l'interface graphique du contrôleur, les méthodes API exclusives aux applications choisies. De cette manière toutes les applications qui n'appartiennent pas à l'ensemble défini, ne peuvent exécuter ces méthodes API. Dans ce cas, l'utilisateur pourrait toujours appliquer des conditions pour définir des scénarios plus spécifiques à cette exclusion. Ces conditions sont créées de la même manière expliquée auparavant.

3.7.3.5 Sauvegarde des règles de sécurité

Une fois que les règles de sécurité sont définies par l'utilisateur, celles-ci seront sauvegardées par le contrôleur d'applications. Un fichier sera, par conséquent, créé pour chaque application. Ce fichier contiendra les règles de sécurité propres à cette dernière. Le contrôleur fait recours à deux autres fichiers, à savoir, un premier qui contient les conditions créées par l'utilisateur et un deuxième qui contient les règles de sécurités qui sont appliquées à l'ensemble des applications. Ces fichiers sont mis à jour à chaque modification effectuée par l'utilisateur.

Les règles de sécurité sont enregistrées en utilisant la syntaxe suivante :

« Action 1 à interdire, Action 2 à interdire, etc. : Condition 1 ET/OU Condition 2 ET/OU, etc. »

Cette règle mentionne que l'action 1 et l'action 2 sont interdites si la condition 1 ET/OU la condition 2 sont vérifiées. Deux exemples concrets peuvent être cités dans ce cas :

- Exec,SendTextMessage : C1 ET (C2 ou C3).
- BNC, RBC : Facebook Ou ScreenShooter.

La première règle interdit l'exécution des deux méthodes API : Exec et SendTextMessage si la condition C1 et l'une des conditions C2 ou C3 sont vérifiées.

Quant à la deuxième règle, elle interdit l'exécution des deux applications BNC, et RBC si l'une des applications Facebook ou ScreenShooter est exécutée.

3.7.4 Fonctionnement du contrôleur d'applications

Le contrôleur d'applications se base sur les règles de sécurités introduites par l'utilisateur et enregistrées dans des fichiers de sauvegarde. Cependant, afin d'accélérer le traitement, ces fichiers ne sont pas sollicités à chaque fois qu'une requête est reçue par une application réécrite.

Le contrôleur d'applications charge les fichiers de sauvegarde lors de son démarrage. Il procédera, par la suite, à l'étape de traitement, où toutes les règles seront analysées. L'algorithme permet donc au contrôleur de décomposer des règles en éléments de base. Ces derniers sont principalement des méthodes API, des noms d'applications, et des conditions.

Une fois que les règles sont analysées par le contrôleur d'application, le traitement sera réalisé en combinant la logique de toutes les règles activées par l'utilisateur. À la fin de ce traitement, le contrôleur d'applications générera des structures de données de type tableau, ces tableaux de références seront utilisés tout au long de l'exécution du contrôleur.

Le premier tableau de référence contiendra, comme le montre la figure 3.14, la liste des méthodes API interdites par l'ensemble des applications. Quant au deuxième tableau de référence, il contiendra dans chaque case indexée par le nom d'une application; une structure représentée par un booléen qui permet de savoir si l'exécution de l'application est permise ou

pas. Le deuxième tableau contiendra aussi la liste des méthodes API interdite par l'application en question.

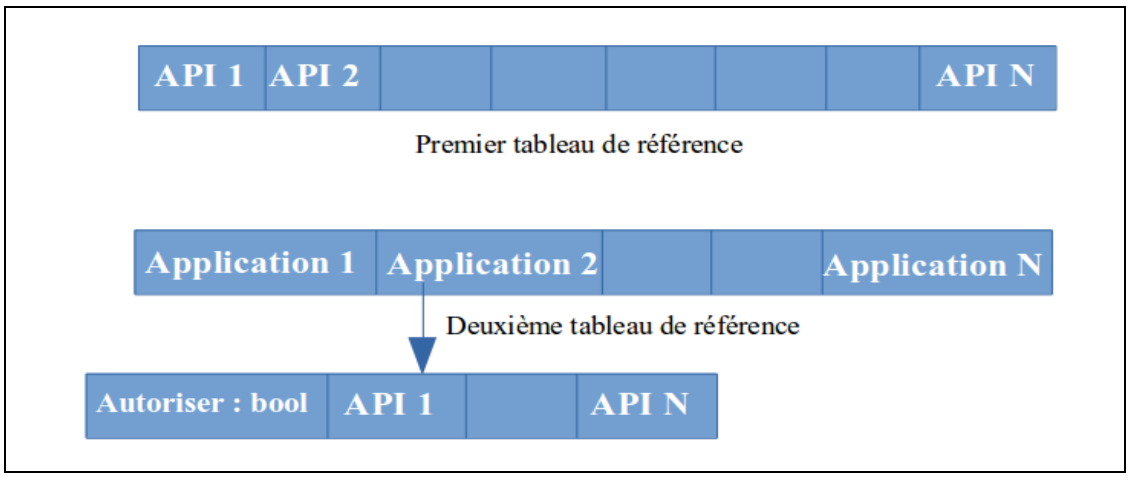

Figure 3.14 Tableaux de références

Quand une application envoie une requête au contrôleur, celui-ci consulte les deux tableaux de références, afin de savoir si l'application où la méthode API demandée a le droit de s'exécuter. Cette information sera transmise à l'application en question.

3.7.5 Rafraîchissement des tableaux de références

Les tableaux de références sont mises à jour par le contrôleur d'applications, dans deux cas de figure. Le premier cas s'agit de la modification ou de l'ajout des règles de sécurité par l'utilisateur. Dans ce cas-ci, les fichiers de sauvegarde seront mises à jour et les tableaux de références seront rafraîchis en tenant compte des modifications apportées.

Le deuxième cas de figure s'agit du changement d'état d'une ou de plusieurs conditions. Dans ce cas, si une règle définie dépend des conditions qui ont changées, les tableaux de références doivent être mises au courant de ce changement.

Si, par exemple, une règle de sécurité interdit l'accès à une application dépendant d'une condition de type intervalle de temps. Quand cette condition est vérifiée, autrement dit quand l'heure actuelle sera incluse dans cet intervalle, l'application devra être interdite. Ainsi, le

booléen dans le deuxième tableau de référence devra être mis à «faux». Afin de permettre ce mécanisme, le contrôleur développé dans la présente étude définit certains services permettant le contrôle des conditions.

3.7.6 Le contrôle des conditions

Afin que les conditions déclarées par l'utilisateur et utilisées dans les règles activées puissent être à jour. Il a été opté pour une solution basée sur l'utilisation des services. Pour cela, un service pour chaque type de condition (Intervalle de temps, Date, Emplacement, Application, Batterie) est implémenté. Chacun de ces services contrôle les conditions appartenant à son type.

Quand une règle de sécurité activée par l'utilisateur dépend d'une ou de plusieurs conditions, ces dernières seront envoyées aux différents services. Chaque service intercepte les conditions homogènes à son type :

- Service de temps : Intercepte les conditions de type Intervalle de temps et contrôle le changement de temps.
- Service de date : Intercepte les conditions de type date et contrôle les changements des dates.
- Service d'emplacement : Intercepte les conditions de type emplacement et contrôle les variations des coordonnées GPS.
- Service d'application : Intercepte les conditions de type application et contrôle l'exécution des applications.
- Service de batterie : Contrôle l'état de la batterie.

Quand un des cinq services détecte qu'une condition change d'état (valide, invalide), ce dernier envoie une notification à une classe observatrice préalablement implémentée. Celle-ci permet le monitorage des conditions ainsi que la mise à jour des deux tableaux de références. Cette classe est enregistrée comme « observateur » des événements envoyés par les cinq services. La figure 3.15 illustre le mécanisme de contrôle de conditions par le contrôleur d'applications.

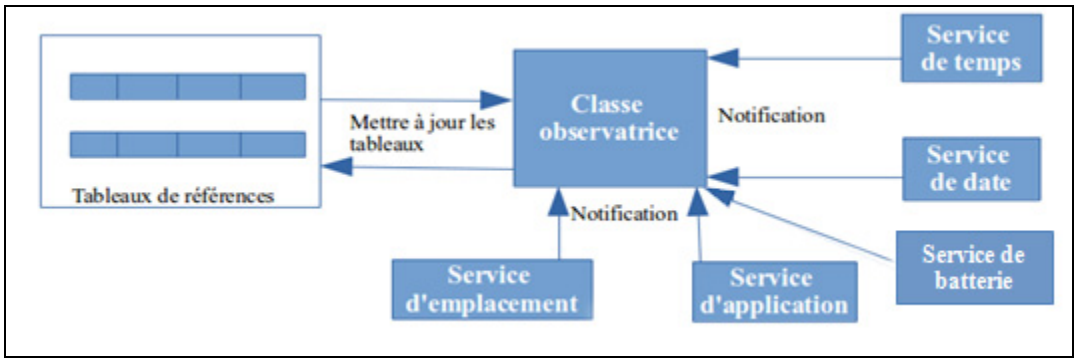

Figure 3.15 Mécanisme de contrôle de conditions

3.7.7 Communication de la réponse

Une fois que le contrôleur d'applications prend la décision adéquate à la requête envoyée par une application réécrite, en consultant les tableaux de références, cette réponse doit être communiquée à l'application en question. Cette communication est assurée par les intents. Le contrôleur va alors créer un nouvel intent, et appliquer l'action « This_From_Monitor ». De cette manière, l'intent pourrait être intercepté par les applications réécrites.

Comme le montre la figure 3.16, la réponse communiquée via l'intent se compose de trois champs. Le premier champ est la signature de l'application demandant l'accès à la ressource. Cette application va alors récupérer cette signature et la comparer avec la sienne, afin de vérifier si le message lui est bien destiné. Si la signature est identique, l'application va, dans ce cas, récupérer le deuxième champ qui représente le numéro de la requête envoyée par cette application. De cette manière, si l'application demandant l'accès se compose de plusieurs Threads qui sollicitent le contrôleur en même temps, cette application saura quelle réponse appartient à quelle requête envoyée.

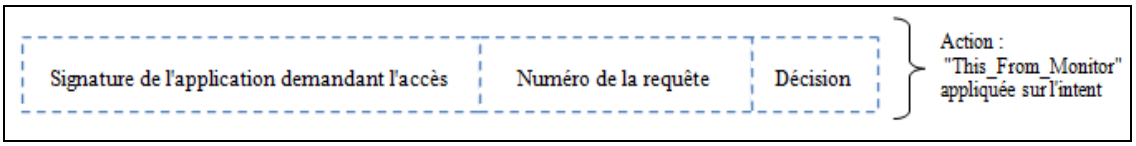

Figure 3.16 Réponse du contrôleur d'applications

Une fois que le Thread qui a envoyé la requête est identifié, l'application réécrite récupérera la décision. Cette dernière représente le droit d'accès à la ressource demandée (Accès refusé, Accès autorisé, accès temporairement refusé). L'utilisateur, comme le représente la figure 3.17, sera averti si l'application devra être fermée conformément à la décision du contrôleur d'applications.

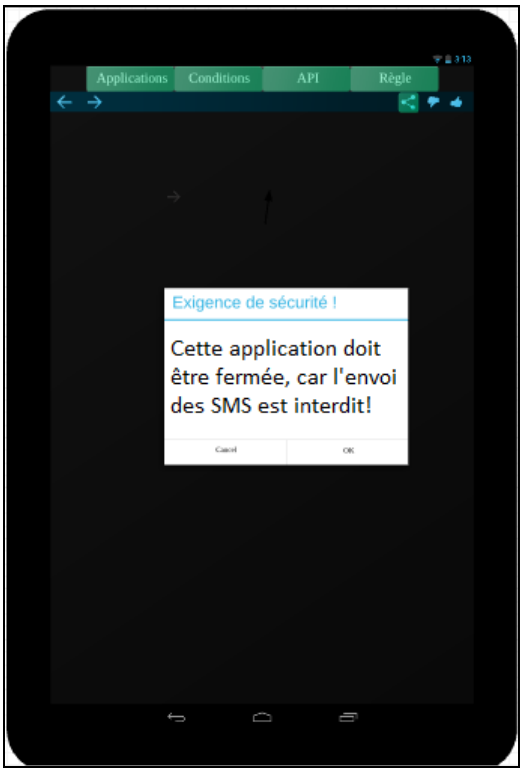

Figure 3.17 Avertissement de l'utilisateur

3.7.8 L'ajout des méthodes API

Comme précédemment mentionné, la présente étude s'est focalisée sur le top 20 des méthodes API les plus fréquentes dans les applications malveillantes. Par conséquent, le contrôleur d'applications développé ne traite que ces méthodes. Cependant, l'utilisateur peut toujours, à sa guise, ajouter plus de méthodes à intercepter. Ledit contrôleur propose via une interface graphique, comme illustré dans la figure 3.18, un moyen d'introduire d'éventuelles méthodes API que l'utilisateur souhaiterait contrôler.

L'utilisateur introduit alors l'appel à la méthode API en langage JAVA. Ainsi, le moniteur s'occupe par la suite de la conversion des instructions java en instruction Smali et rajoute l'appel en question à la liste d'appels déjà introduites. De cette manière, le nouvel appel sera intercepté par le Framework dans la phase de réécriture.

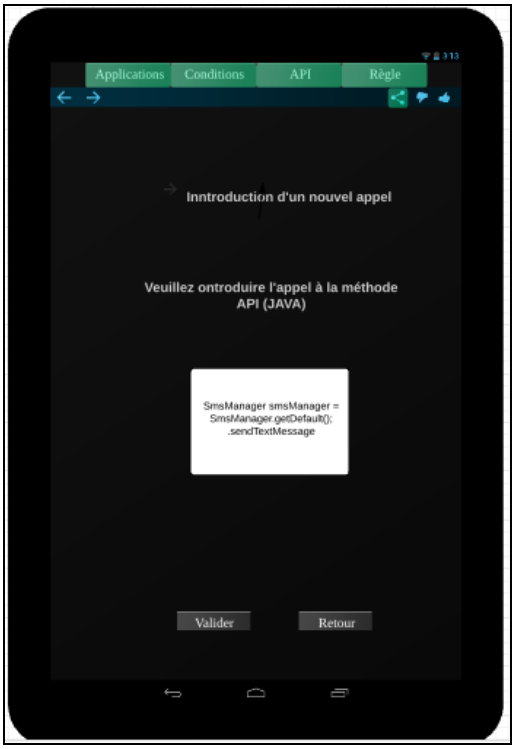

Figure 3.18 Ajout d'un nouvel appel API

3.8 Conclusion

Nous avons présenté dans ce présent chapitre l'architecture des deux axes de la solution proposée, à savoir, le Framework de réécriture d'applications, et le contrôleur d'applications. Nous illustrons dans le chapitre suivant les tests et expérimentations élaborés.

CHAPITRE 4

TEST ET EXPÉRIMENTATIONS

4.1 Introduction

L'objectif des expériences menées est de fournir les résultats des évaluations du système de sécurisation des applications Android élaboré. Dans ce chapitre, les tests et résultats réalisés pour chacun des deux axes de notre solution sont présentés, à savoir, le Framework de réécriture des applications et le contrôleur d'applications.

Tout d'abord, les tests de fonctionnalités propres au Framework de réécriture des applications sont mis en évidence. Où sont présentés les résultats des tests réalisés concernant la réécriture d'applications, l'augmentation de la taille des applications réécrites, ainsi que les tests d'insertion des appels de contrôles.

Par la suite, les différents tests réalisés sont présentés afin d'évaluer le contrôleur d'applications développé dans cette étude. En effet, un ensemble de tests est illustré. Celui-ci, assurant aussi bien le bon fonctionnement des mécanismes de contrôles que celui des politiques de sécurité. Par la suite, les calculs de l'overhead de temps d'exécution de la solution développée dans cette étude seront présentés.

Les expériences ont été réalisées sur un téléphone Samsung Galaxy S3 exécutant le système d'exploitation Android 4.1.1, avec 2G de RAM, un processeur dual-core cadencé à 1,5 GHz et une mémoire interne de 12G.

Dans la présente étude, la variante du Framework, fonctionnant sur le serveur web, est installée sur un serveur tournant sous le système d'exploitation Linux ubuntu 14.04, comportant un processeur core i5-4200U, 8GB DDR3 RAM. L'application fonctionne sous le serveur web Apache 2.

4.2 Framework de réécriture d'applications

Dans cette première partie seront présentés les tests concernant le Framework de réécriture des applications.

4.2.1 Réécriture des applications

Afin d'évaluer l'efficacité du Framework de réécriture d'applications développé dans cette étude, des tests sur un ensemble de 950 applications Android ont été exécutés. Lesdites applications ont été choisies au hasard à partir de Google Play et ce, ajoutés à 450 applications malicieuses (Mila, 2011). Il est à noter que l'ensemble des applications ont été réécrites en utilisant notre Framework.

Les applications malicieuses utilisées appartiennent à différentes catégories: jeux, applications de musique, réseaux sociaux, etc. Ces applications contiennent des malwares appartenant à plusieurs familles. La taille de ces applications varie entre 1 Mo et 50 MO.

Le tableau 4.1 illustre le taux de réussite de la réécriture d'applications par le Framework. Il est à noter que les Logs fournis par le Framework de réécriture d'applications ont été récupérés afin d'analyser les messages d'erreurs dus à l'échec lors de la phase recompilation. Après l'analyse de ces logs, il a été constaté que les erreurs sont générées par apktool (Tumbleson, 2015) et n'ont aucun rapport avec les modifications apportées aux applications.

En effet, Apktool n'arrive pas dans certains cas à recompiler une application même si aucune modification n'est apportée à cette dernière. Il génère dans la plupart des cas des erreurs en relation avec les ressources de l'application (Lacombe, 2009).

Tableau 4.1 Taux de réussite de réécriture d'applications

Type d'applications	Nombre d'applications	Taux de sucés de la réécriture
Applications tirées de Google Play	950	99,78% (948)
Application malicieuse	450	99,77% (449)
L'ensemble des applications	1400	99.78% (1397)

Comme représenté sur le tableau 4.1, le Framework de réécriture d'applications développé dans la présente étude assure un taux de réussite de 99.78 % lors de la phase de recompilation. Autrement dit, la compilation de 3 applications a échoué sur un ensemble de 1400 applications. Ce taux de réussite est considérablement élevé par rapport à d'autres approches étudiées. L'étude présentée par (El-Harake et al., 2014) qui est une approche de réécriture d'applications basée sur un compilateur AspectJ (Ajc, 2015), par exemple, illustre un taux de réussite de 90% lors de la phase de recompilation. Aurasium, l'approche proposée par (Xu et al., 2012), utilise le même compilateur apktook (Tumbleson, 2015) utilisé dans le développement de la solution présentée dans cette étude. Cette approche, assure un taux de réussite de 96 %, soit un taux similaire à celui de la compilation de l'approche Capper présenté par (Zhang et Yin, 2014).

4.2.2 Temps de réécriture d'applications

Comme déjà illustré dans le chapitre précédent, l'approche de réécriture d'applications adoptée dans la présente étude propose deux alternatives. La première étant une application Android tierce partie tandis que la deuxième étant une solution implémentée sur un serveur web.

Dans cette partie, le temps de traitement nécessaire pour la réécriture des applications par les deux variantes du Framework est présenté. Un ensemble de 1350 applications a été réécrit par les deux variantes. La taille de ces applications varie entre 1 et 100 MO.

Le tableau 4.2 illustre un échantillon de dix applications où, le temps de réécriture de chaque application par les deux variantes du Framework a été calculé. Le tableau illustre aussi la différence de temps d'exécution entre la variante implémentée sur le périphérique mobile et celle hébergée sur le serveur web.

Tableau 4.2 Temps de réécriture des applications

Application	Taille de l'application (MO)	Temps d'exécution sur le serveur (s)	Temps d'exécution sur le périphérique (s)	Temps d'exécution(s) (périphérique-serveur)
LE Monde	1	40	62	22
ConnectTheDots	1	30	50	20
MassengerPlayer	4	79	104	25
ITubePro	4	38	61	23
VLC	12	60	85	25
PafLeChien	12	75	94	19
GamerZ	26	76	96	20
SubwaySurf	28	79	96	17
TempleRun	48	81	101	20
ClashRoyale	84	75	101	26

Il peut être remarqué à partir du tableau 4.3 que le temps de réécriture ne varie pas en fonction de la taille des applications et ce, pour deux raisons. La première est que apktool ne recompile que les fichiers codebyte. En effet, les ressources de l'application comme les image, les fichiers XML, etc. ne seront pas pris en compte. De ce fait, une application peut avoir une taille considérable à cause des ressources non compilées. Ce pendant, celle-ci ne compote pas un grand nombre de fichiers Smali, ce qui accélère sa compilation et décompilation par apktool.

La deuxième raison est que le Framework développé dans cette étude ne traite que les fichiers Smali. Par conséquent, le temps de réécriture déprend uniquement de la taille de ces derniers. La figure 4.1 illustre pour les dix échantillons, le temps de réécriture par chacune des variantes du Framework en fonction de la taille des applications. Il y peut être remarqué également que le Framework de réécriture d'application nécessite plus de temps sur le périphérique mobile que sur le serveur web. Ceci est dû aux ressources limitées du périphérique.

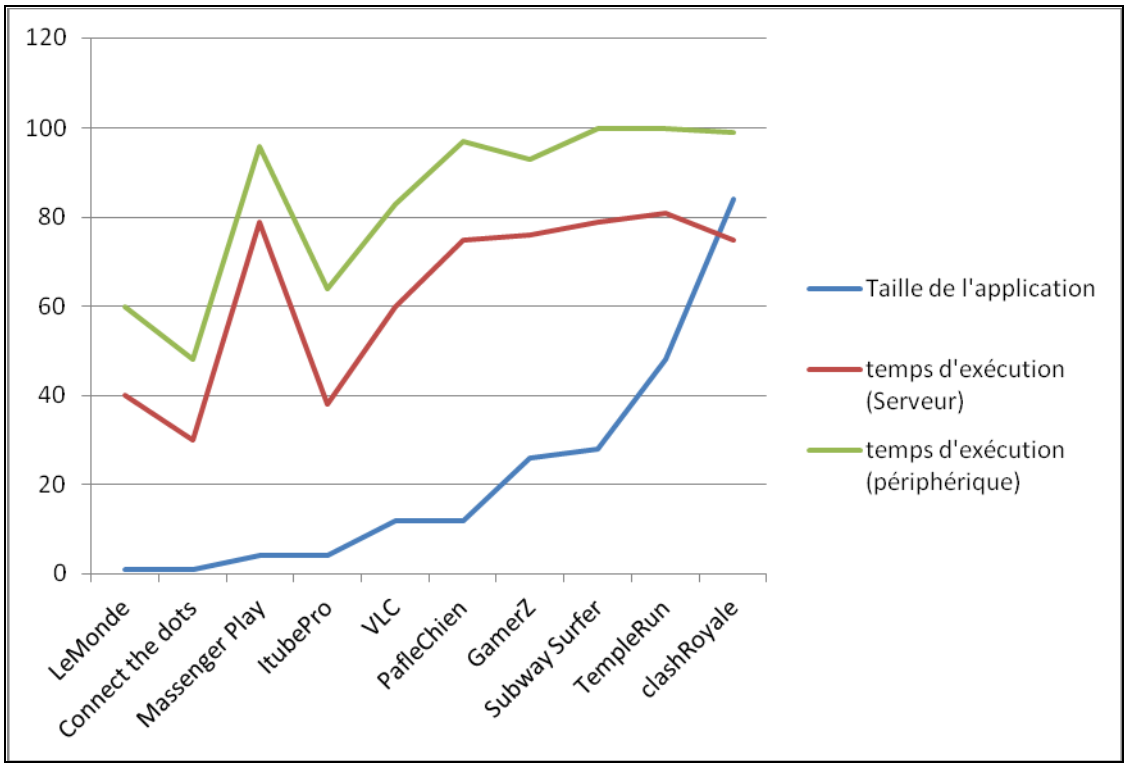

Figure 4.1 Variation du temps de réécriture en fonction de la taille des applications

Dans la présente étude, pour l'ensemble des 1350 applications, la moyenne de la différence de temps entre les deux variantes a été calculée. Cette valeur est de 21s. Le Framework exécuté sur le périphérique mobile nécessite donc en moyenne 21s de plus que la variante implémentée sur le serveur web.

De plus, pour l'ensemble des 1350 applications le temps moyen de réécriture a également été calculé. Celui-ci est de 51s pour la variante serveur web et de 72 s pour la variante implantée dans le périphérique. La figure 4.2 illustre le temps de réécriture pour l'ensemble des applications pour chacune des deux variantes. Il peut être également remarqué à travers cette figure que juste 20% des 1350 applications nécessitent un temps de réécriture de 80s sur le serveur et 100s sur le périphérique mobile. Les autres applications nécessitent un temps de réécriture moins considérable.

(Zhang et Yin, 2014) illustrent le temps de traitement de Capper. Celui-ci étant de 60s. Cependant, ces derniers n'incluent en aucun cas le temps de décompilation par Dex2jar qui, lui, peut prendre plusieurs secondes. Cette tâche représente en moyenne plus de 40% de la phase de réécriture de la présente étude. Le temps de réécriture par le Framework pourrait alors être réduit de 40% si le temps de décompilation n'est pas pris en compte.

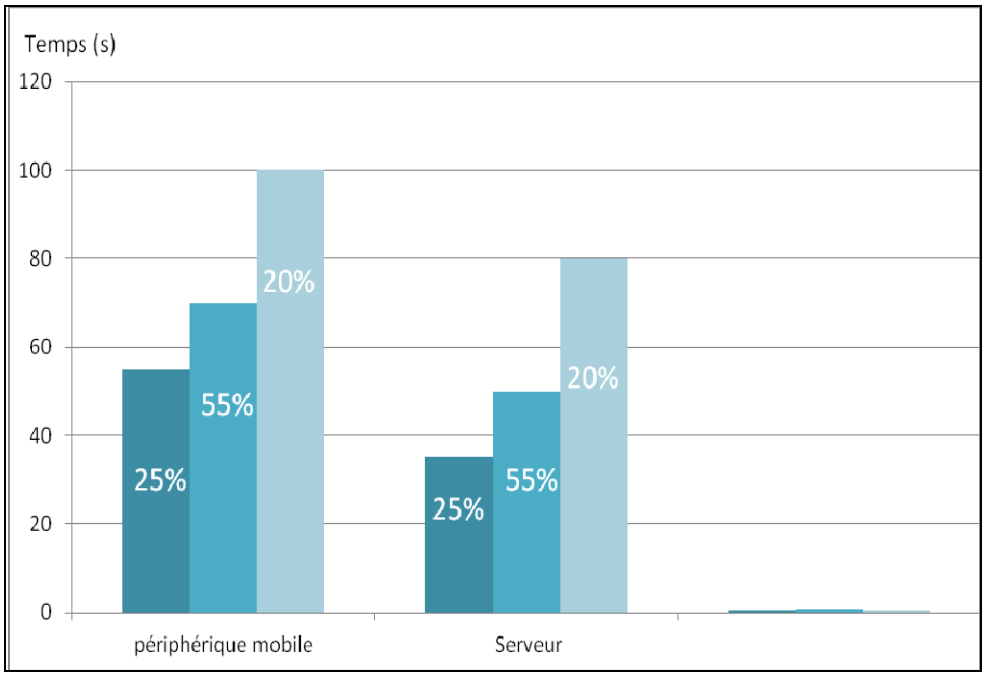

Figure 4.2 Temps de réécriture de l'ensemble des applications

4.2.3 Taille des applications

Les périphériques mobiles souffrent de la contrainte des ressources limitées. Il est donc important de mesurer l'impact de la réécriture des applications sur le système.

Il est à noter que apktool (Tumbleson, 2015) étant l'outil utilisé par le Framework afin de compiler et décompiler les applications, ne conserve pas la même taille de l'application une fois que cette dernière est décompilée puis recompilée. La nouvelle compilation peut augmenter ou diminuer la taille des applications. Ainsi, afin d'avoir une meilleure illustration de cette variation de taille, chaque application a été décompilée en utilisant apktool (Tumbleson, 2015) puis recompilée sans effectuer aucune modification à ces dernières. Les résultats de cette opération sont illustrés dans le tableau 4.3.

Dans la présente étude, les portions de codes rajoutées par le Framework varient d'une application à l'autre. Le Framework développé dans cette étude copie, tout d'abord, les classes Smali nécessaires à la communication avec le contrôleur d'applications dans l'application en question. Par la suite, il parcourt le code byte de l'application et procède à l'insertion des instructions appelant les méthodes de ces classes selon le type d'API sollicité. De ce fait, certaines applications peuvent nécessiter plus de code byte à rajouter que d'autres.

Cette variation affecte donc la taille des applications. Afin d'avoir une idée plus claire sur la variation des tailles, l'ensemble de 1350 applications Android ont été réécrites en utilisant notre Framewok et ce, en observant leurs tailles avant et après la phase de réécriture. Le tableau 4.3 illustre les différentes tailles pour un ensemble de dix applications.

Le tableau 4.3 représente aussi la différence de taille entre l'APK original de l'application et l'APK généré par le Framework de réécriture d'applications. Cette valeur inclut la modification de taille apportée par apktool.

Les pourcentages illustrés dans le tableau 4.3 représentent le pourcentage de taille rajoutée à chaque application (par le Framework) par rapport à sa taille originale et ce, en incluant la taille rajoutée par apktool, ainsi que le pourcentage de taille rajoutée par le Framework en

excluant la taille rajoutée par apktool. Cette dernière représentant la taille du code byte permettant le contrôle des appels aux méthodes API.

Tableau 4.3 Tailles des applications réécrites par le Framework

	Originale	Recompilée avec Apktool (o)	Renforcée (Framework) (o)	Renfoncée-Recompilée (o)	Taille rajoutée (o)	Pourcentage de Taille de code byte rajouté	Pourcentage de taille rajoutée
FBMessenger	23494355	23671070	23 673 491	2421	179136	0.01%	0,76%
Traffic Rider	94019694	94472732	94 502 477	29745	482783	0.03%	0,51%
World Chef	64631472	65798804	65 981 462	182658	1349990	0.28%	2.09%
BatterySaver	8295601	8344668	8 357 203	12535	61602	0.15%	0,74%
Color Switch	14051589	14099235	14 102 189	2954	50600	0.02%	0.36 %
Instagram	11322148	11877285	11 996 209	118924	674061	1.05%	5,95%
Netflix	15761539	15993453	16156718	163265	463265	1.03%	2,51%
SpotifyMusic	27368605	27985605	27 986 774	1169	51020	0.004%	2.26%
Snapchat	34545629	34629675	34688877	59202	59202	0.17%	0,41%
Piano Tiles 2	18869921	18912502	18 935 851	23349	65930	0.12%	0,35%

Pour l'ensemble de 1350 applications choisies, la taille moyenne rajoutée a été calculée. Celle-ci, étant de 98822 octets (0.098 Mo), soit une taille relativement petite par rapport aux tailles des applications. De plus, le pourcentage moyen de taille rajoutée pour l'ensemble des 1350 applications a été calculé. Celui-ci étant de 1.2%. (Benjamin Davis 2012) estiment le pourcentage de taille rajoutée par i-arm-droid à 2%. Alors que (Xu et al., 2012) estiment quant à eux la taille rajoutée par Aurasium à 52000 octets.

La figure 4.3 illustre les différentes tailles des dix applications étudiées, dans chacun des trois états : original, après recompilation avec Apktool et après réécriture par notre Framework.

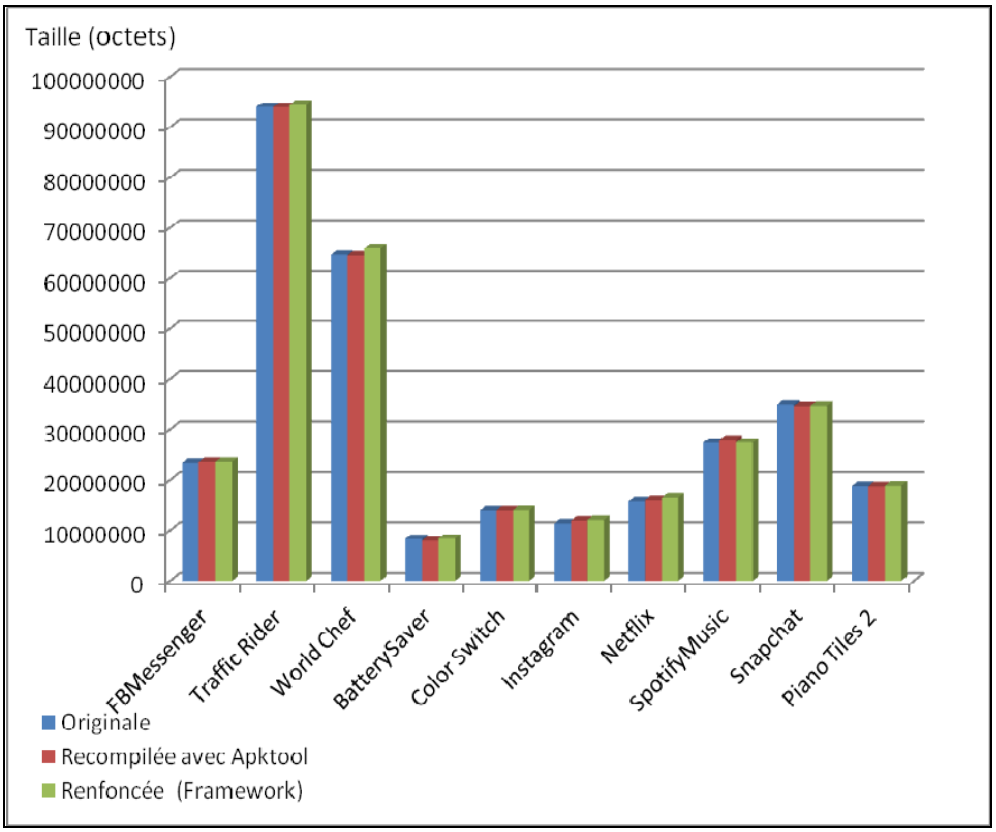

Figure 4.3 Tailles de dix applications réécrites par le Framework

La figure 4.4 représente les différentes tailles (originale, après compilation et après réécriture) pour un ensemble de 134 applications traitées. Nous remarquons que la taille rajoutée par notre Framework est très petite par rapport à la taille originale des applications.

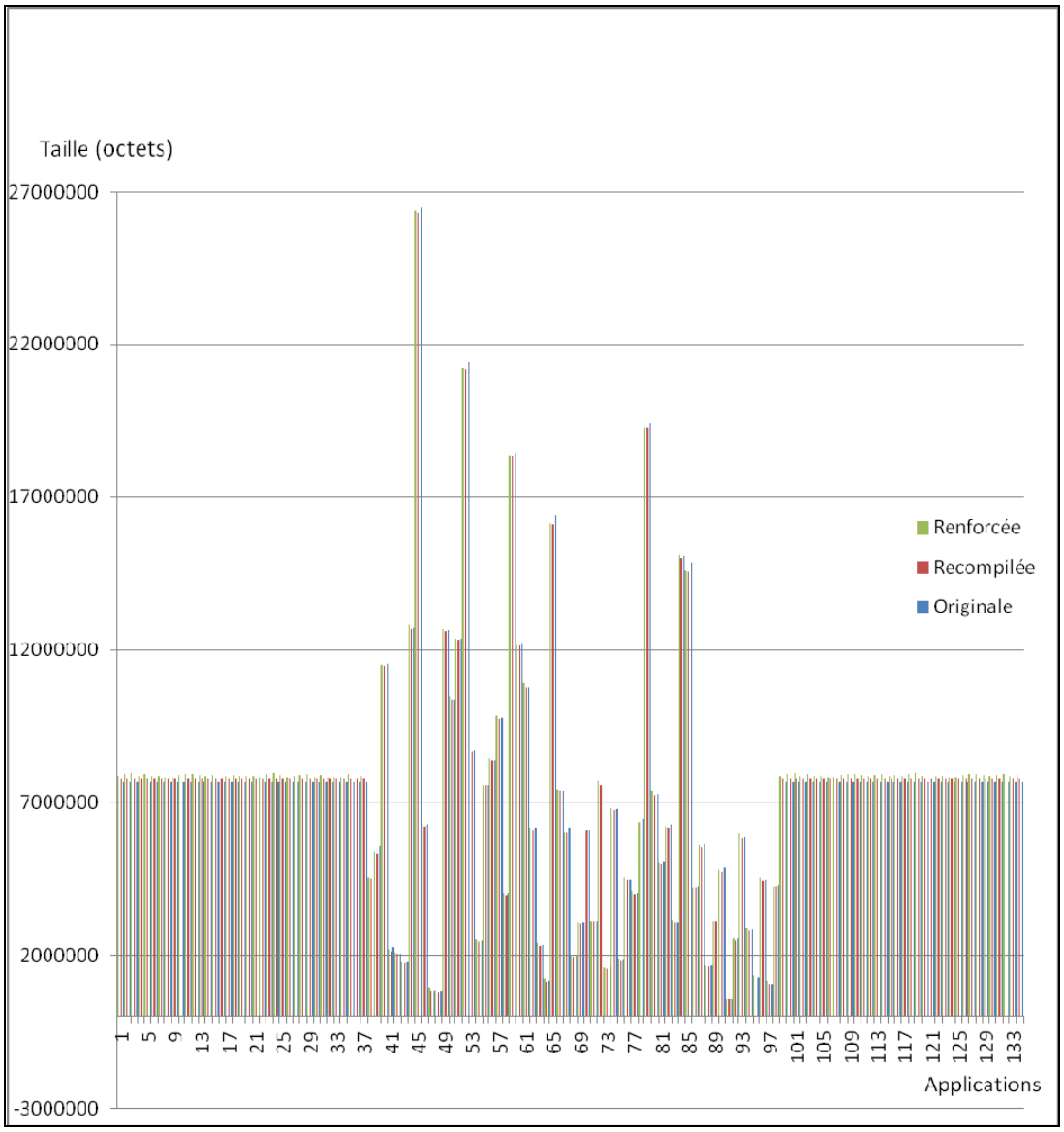

Figure 4.4 Tailles de 134 applications réécrites par le Framework

4.2.4 Insertion des appels de contrôles

Le Framework de réécriture d'applications développé dans cette étude a deux tâches principales, à savoir, l'ajout des classes Smali permettant la communication avec le contrôleur et l'insertion des appels de contrôle d'API (sollicitant les classes). Afin de vérifier l'intégralité de ces insertions, dix applications ont été réécrites par le Framework. Ainsi, les

versions générées par le Framework ont été analysées manuellement afin de vérifier si les appels de contrôle ont bien été rajoutés avant chaque appel aux méthodes API dans l'ensemble du code byte de chaque application.

Grâce à cette analyse, il a pu être constaté que l'algorithme d'insertion des méthodes de contrôle développé dans cette étude assure un taux de succès de 100%. Autrement dit, pour chaque appel à une méthode API, le Framework a rajouté l'appel à la méthode de contrôle spécifique à l'API.

Comme déjà expliqué dans le chapitre précédent, l'évaluation est basée sur les vingt méthodes API les plus fréquentes dans les applications malicieuses (Aafer, Du et Yin, 2013). Le tableau 4.4 illustre les API retrouvés dans les dix applications malicieuses choisis au hasard. Ledit tableau contient 12 méthodes API sollicitées par les applications en question. Les abréviations desdites méthodes ainsi que leurs explications sont les suivantes:

- EXC: Ljava/Lang/RunTime->Exec(), permet d'exécuter une commande shell.
- SMS : Landroid/telephony/SmsManager->SendTextMessage(), permet d'envoyer un message texte (SMS)
- NET : java.net.HttpURLConnection->openConnection(), permet d'ouvrir une connexion HTTP.
- ID : Landroid/telephonyManager/getDeviceId(), permet de récupérer l'identifiant du périphérique.
- SS : Landroid/content/Context->startService(), permet de démarrer un service.
- PM : Landroid/content/pm/PackageManager->GetPackageManager(), retourne une instance de packageManager contenant des informations sur le package global.
- GOS : Ljava/lang/Process ->GetOutputStream(), retourne le OutputStream connecté au InputStream standard.

- WF Ljava/lang/Process-> WaitFor(), permet d'attende la fin de l'exécution d'un processus.
- GLN Landroid/telephony/TelephonyManager->GetLine1Number(), retourne le numéro de téléphone de la première ligne téléphonique.
- LL Landroid/content/pm/ApplicationInfo->LoadLabel(), permet de récupérer l'étiquette textuelle courante associée à un objet .
- FL Ljava/io/FileOutputStream->Flush(),permet d’écrire toutes les données en mémoire vers la sortie (OutPutStream, etc.)

Tableau 4.4 Méthodes API utilisés par les applications malicieuses

	EXC	SMS	CA	NET	ID	SS	PM	GOS	WF	GLN	LL	FL
ADRD			✓	✓		✓						
AnserverBot	✓	✓	✓	✓								
Asroot						✓	✓					
CoinPirate		✓								✓		
Crusewin		✓									✓	
DogWars								✓				✓
DroidCoupon			✓	✓					✓			
DroidDeluxe	✓				✓							
DroidKungFu	✓					✓	✓					✓
Jifake						✓	✓					

Les tests réalisés ont été effectués sur les 20 méthodes API citées dans le chapitre précédent. Les méthodes non représentées dans le tableau 4.4 n'ont pas été retrouvées dans l'ensemble des dix applications sélectionnées. L'analyse manuelle visant à rechercher ces derniers confirme que ces méthodes n'existent pas dans le code des applications.

4.3 Contrôleur d'applications

Dans cette deuxième partie, les tests relatifs au contrôleur d'applications développé sont présentés.

4.3.1 Tests de fonctionnement

Afin de s'assurer que les modifications apportées par le Framework ne nuisent pas au fonctionnement des applications, un ensemble de vingt applications a été réécrit par le Framework. Aussi, lesdites applications sont exécutées dans le périphérique de test tout en observant les logs générés par les applications en question et le contrôleur.

Les logs générés par les applications sont principalement des traces des appels API sollicités par ces dernières. Le contrôleur, à son tour, génère un ensemble de logs représentant les appels API interceptés. Les politiques de sécurité implémentées dans le contrôleur ont été désactivées dans un premier lieu afin d'observer le fonctionnement normal des applications.

Les applications installées dans le périphérique exécutent leurs événements grâce à Monkey (Contagiodump, 2015). Celui-ci étant une application fonctionnant sous Adnroid permettant de générer des événements utilisateurs au hasard. Comme, par exemple, cliquer sur les boutons ainsi que les autres composantes de l'application. Dans cette étude, l'application Monkey (Contagiodump, 2015) a été utilisée afin de stresser les applications d'une manière aléatoire. Les logs ont permis de vérifier si les applications fonctionnent normalement ou cessent de fonctionner à cause des modifications apportées.

Grâce aux logs, il a été constaté que les modifications apportées par notre Framework de réécriture n'influencent en aucun cas le bon fonctionnement des applications. De Plus, les logs générés par les applications ont été comparés avec les logs du contrôleur afin de s'assurer que tous les appels aux méthodes API sollicitées par les applications ont bien été interceptés par le contrôleur.

4.3.2 Politiques de sécurité

Comme expliqué dans le chapitre précédent, le contrôleur d'applications développé dans la présente étude se base sur un ensemble de politiques de sécurité introduites par l'utilisateur afin de contrôler les applications installées dans la plateforme Android. En effet, des tests de fonctionnement ont été réalisés afin de s'assurer que ledit contrôleur analyse et implémente les politiques de sécurité. Les tests ont été réalisés grâce à un ensemble d'applications réécrites par le Framework. Ces dernières ont été installées sur le périphérique de test et leurs événements ont été lancés d'une manière aléatoire grâce à Monkey (Contagiodump, 2015).

Dans cette étude, un ensemble de politiques de sécurité a été choisi et implémenté dans le contrôleur a été introduit. Ainsi, les logs, générés par ce dernier et par les applications, ont été analysés. Le but de cette manœuvre est de s'assurer que les méthodes API sollicitées par les applications ont bien été contrôlées conformément aux politiques de sécurité.

Les politiques de sécurité, introduites dans le contrôleur, dépendent des applications exécutées ainsi que des méthodes API sollicitées, des contraintes de temps, de date, de localisation, et de batterie. Voici quelques politiques de sécurité utilisées lors des tests :

- Interdiction de lancer l'application A, si : (Application B et Application C) ou (Application B et Application D) sont exécutées.
- Interdiction d'utiliser (API1 et API2) par les applications A et B, Si : (API1 est utilisé par n'importe quelle application, Où AP2 est utilisé par les applications (D ou G)).
- Interdiction d'utiliser API1 par toutes les applications si : ((API2 est utilisé par l'application A et les coordonnées de géolocalisation=X), Ou (API 3 est utilisé par les applications C et D et F et (Intervalle de date = D Ou Intervalle de temps = T)).
- Interdiction d'utiliser API1 et API2 : si (Batterie < 50%, Et application A est exécutée).

La liste des politiques présentée n'illustre que 4 politiques sur un ensemble de 30 politiques de sécurités implémentées pour la réalisation des tests. Grâce à l'analyse des logs, le comportement de chaque application a pu être identifié. Ainsi, il a été constaté que le contrôleur gère tout l'ensemble des applications exécutées et permet d'exiger un contrôle sur tout appel aux méthodes API afin de s'assurer que ces dernières respectent bien les politiques de sécurité définies par l'utilisateur.

Ci-après, un cas d'étude permettant l'introduction de la politique de sécurité est cité :

« Il est interdit à toutes applications de prendre une capture d'écran si les applications bancaires RBC et BNC sont exécutées, et cela du lundi au jeudi, de 13h à 16h »

Tout d'abord, la politique de sécurité dans le contrôleur d'application est introduite en définissant les trois conditions suivantes:

- T1 : intervalle de temps de 13h à 16h
- D1 : intervalle date du lundi au jeudi
- E1 : l'application BNC est exécutée ou l'application RBC est exécutée

Par la suite, la condition globale est définie:

- G : T1 & D1 & E1

L'API "ScreenShot" a été par la suite interdit sur l'ensemble des applications installées si la condition G est vérifiée.

L'application "ScreenShotTaker" a été développée pour permettre de prendre une capture d'écran toutes les 15 secondes. Ainsi, celle-ci a été réécrite grâce à notre Framework. La figure 4.2 illustre les logs récoltés lors de l'exécution du contrôleur ainsi que les deux applications : ScreenShotTaker et RBC.

Dans un premier temps, le contrôleur et l'application "Screenshoter" sont lancés à 12h 59 minute. Trente secondes plus tard, l'application "BNC" a été exécutée. Comme la figure 4.5

le montre, l'application "ScreenShotTaker" n'a pas été arrêtée puisque la condition T1 n'est pas vérifiée.

```
12.59.03 : Application ScreenShoter démarrée.
12.59.03 : API "ScreenShot" Sollicité.
12.59.18 : API "ScreenShot" Sollicité.
12.59.33 : API "ScreenShot" Sollicité.
12.59.48 : API "ScreenShot" Sollicité.
13.00.03 : API "ScreenShot" Sollicité.
13.00.18 : API "ScreenShot" Sollicité.
13.00.33 : API "ScreenShot" Sollicité.
13.00.48 : API "ScreenShot" Sollicité.
13.01.03 : API "ScreenShot" Sollicité.
13.01.18 : API "ScreenShot" Sollicité.
13.01.18 : Application fermée par le contrôleur "API ScreenShot interdit".
```

Figure 4.5 Logs générés par l'application "ScreenShotTaker"

À 13h 01, nous avons relancé l'application BNC, et comme le montre la figure 4.6 l'application ScreenShoter a bien été arrêtée par le contrôleur d'application (condition G vérifiée).

```
12.58.00 : Controleur d'applications démarrée.
12.59.03 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
12.59.18 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
12.59.31 : Application RBC lancée (Autorisé).
12.59.33 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
12.59.48 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
13.00.03 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
13.00.18 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
13.00.33 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
13.00.48 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
13.01.03 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
13.01.04 : Application RBC lancée (Autorisé).
13.01.18 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Interdit).
13.01.18 : Fermeture de l'application "ScreenShoter".
```

Figure 4.6 Logs générés par le contrôleur d'applications

4.3.3 Temps d'exécution

Le contrôleur d'applications intercepte tout appel à une méthode API afin de l'analyser. Un temps d'analyse est donc rajouté au temps d'exécution de chaque méthode API pour l'application réécrite. Afin de mesurer ce temps d'exécution, six applications sollicitant différentes méthodes API ont été créées. Ainsi, le temps d'exécution nécessaire pour l'exécution de chaque méthode a été calculé.

Dans cette étude, les instructions de calcul de temps d'exécution sont donc placées dans le byte code des applications avant et après chaque appel à une méthode API. Afin de calculer le temps d'exécution rajouté par la phase d'analyse, six applications sont lancées dans un premier temps en sollicitant un ensemble de méthodes API (avant réécriture).

La figure 4.7 illustre les logs fournis par les applications lors de chaque appel à une méthode API où le temps d'exécution a été calculé après l'appel.

```
16.30.31 : API "ScreenShot" Sollicité par l'application "ScreenShoter" (Autorisé).
Temps d'exécution : 106ms.
16.31.02 : API "GPS" Sollicité par l'application "GetLoc" (Autorisé).
Temps d'exécution : 113ms.
16.31.59 : API "SMS" Sollicité par l'application "SMSSender" (Autorisé).
Temps d'exécution : 120ms.
16.32.04 : API "PackageManager" Sollicité par l'application "GamerG" (Autorisé).
Temps d'exécution : 111ms.
16.32.19 : API "GetLineNumber1" Sollicité par l'application "GamerG" (Autorisé).
Temps d'exécution : 203ms.
16.32.19 : API "Flush" Sollicité par l'application "PokerZ" (Autorisé).
Temps d'exécution : 199ms.
16.32.48 : API "StartService" Sollicité par l'application "SMSSender" (Autorisé).
Temps d'exécution : 312ms.
16.32.56 : API "PackageManager" Sollicité par l'application "VideoReader" (Autorisé).
Temps d'exécution : 136ms.
```

Figure 4.7 Logs générés par les applications de tests (sans contrôleur)

Par la suite, les six applications sont réécrites afin qu'elles puissent communiquer avec le contrôleur. Ainsi, les logs générés sont récupérés par ces dernières. La figure 4.8 illustre les nouveaux temps d'exécution calculés.

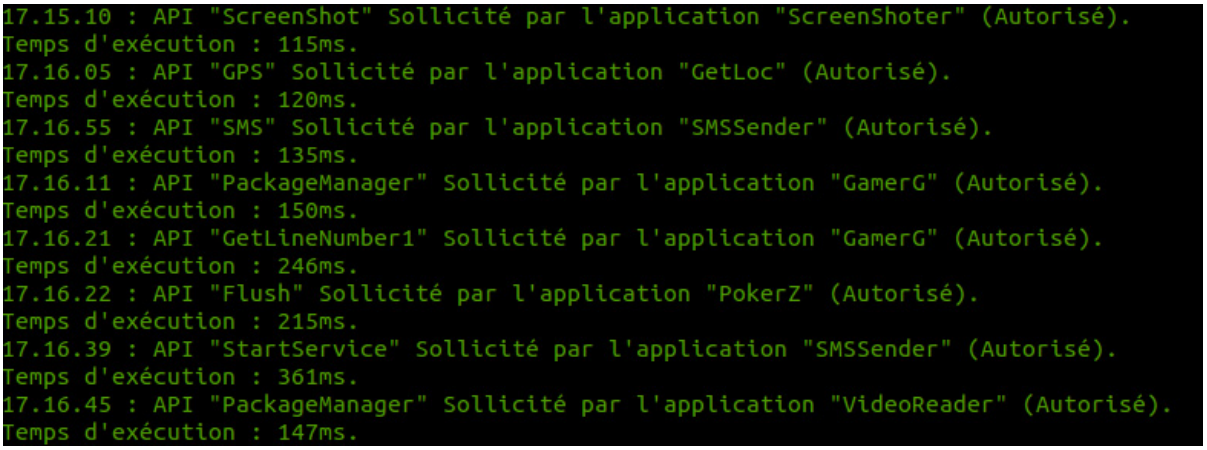

Figure 4.8 Logs générés par les applications de tests (avec contrôleur)

Le tableau 4.5 illustre la différence entre le temps d'exécution des méthodes API avant et après réécriture des applications par le Framework développé dans la présente étude. Il peut être également remarqué à travers ce tableau que pour cet ensemble de six applications, le temps d'analyse varie entre 7ms et 49ms Ceci représente un temps relativement négligeable. Le pourcentage représentant le temps d'analyse rajouté lors des appels aux méthodes API a été calculé et représenté dans le tableau 4.5.

Tableau 4.5 Temps d'exécution des méthodes API

Application	Méthode API	Temps (Ms) avant réécriture	Temps (Ms) après réécriture	Temps d'analyse (Ms)	Pourcentage de temps d'analyse rajouté (Ms)
ScreenShoter	ScreenShot	106	115	9	8.5%
GetLoc	GPS	113	120	7	6.19%
SMSSender	SMS	120	135	15	12.5%
GamerG	PackageManager	111	150	39	35.13%
GamerG	GetLineNumber1	203	246	43	21.18%
PokerZ	Flush	199	215	14	7.03%
SMSSender	StartService	312	361	49	15.70%
VideoReader	PackageManager	136	147	11	8.08%

La figure 4.9 illustre le temps d'exécution des méthodes API par les six applications étudiées, avant et après la réécriture des applications. Nous remarquons que le temps d'exécution rajouté par la phase d'analyse de notre contrôleur d'applications est relativement petit par rapport au temps d'exécution de l'appel sans contrôle. Il représente 14.37% pour l'ensemble des huit méthodes API illustrées dans la figure 4.9.

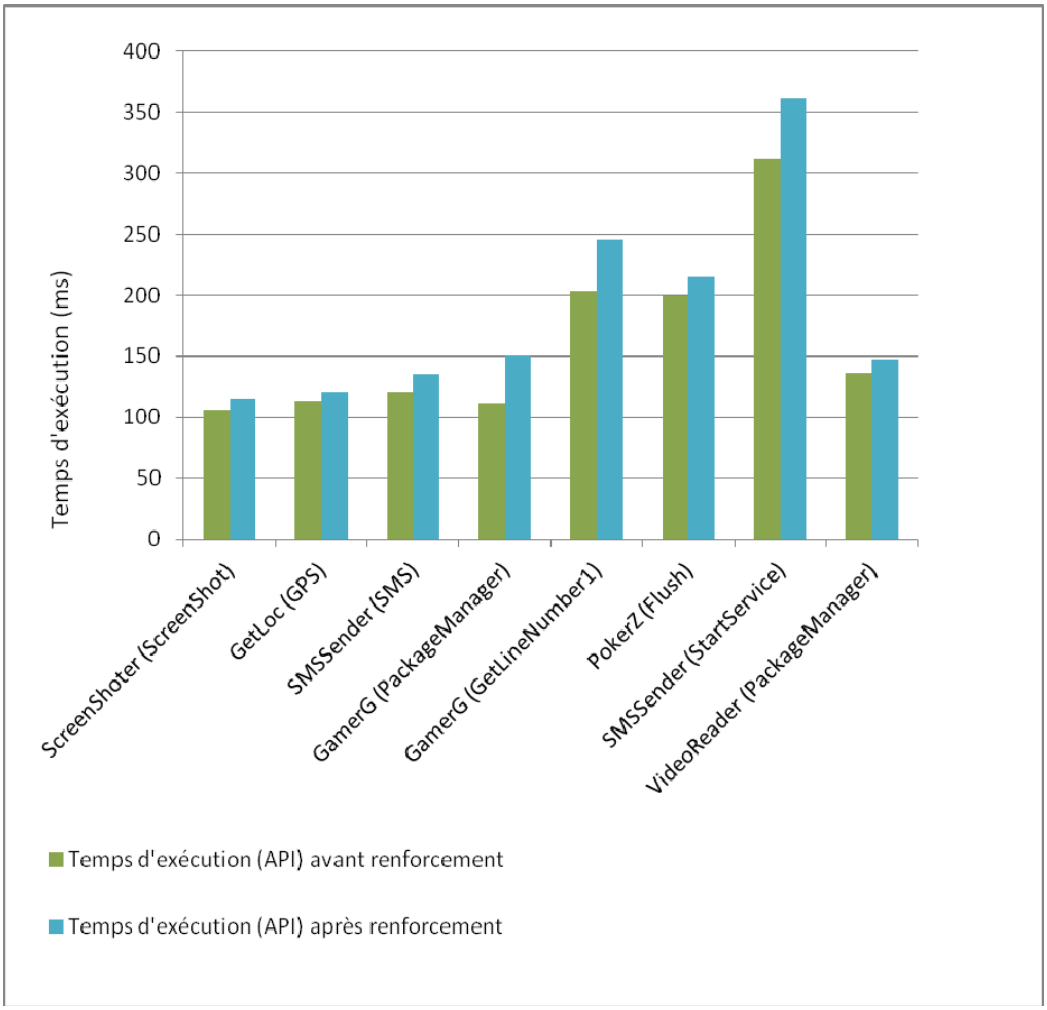

Figure 4.9 Temps d'exécution des méthodes API

De plus, des tests ont été effectués sur un ensemble de 20 appels API afin de calculer l'overhead de temps d'exécution rajouté par le contrôleur d'applications et qui représente 20% par rapport au temps d'exécution de l'appel API sans contrôle. Il est à noter que (Xu et al., 2012) se sont, à leur tour, aussi intéressés aux contrôles des méthodes API. Ces derniers estiment l'overhead rajoutée par Aurasium à 35%.

4.4 Discussion

L'approche que nous avons proposée afin de contrôler l'exécution des méthodes API, se compose de deux axes principaux, à savoir, le Framework de réécriture d'applications et le contrôleur d'applications. Les deux axes de notre solution ont démontré une importante efficacité durant les expérimentations élaborées. Notre Framework assure un taux de réussite de réécriture de 99.78%. Ce taux de réussite est considérablement élevé comparé aux autres approches étudiées. L'étude présentée par (El-Harake et al., 2014) qui est une approche de réécriture d'applications basée sur un compilateur AspectJ (Ajc, 2015), par exemple, illustre un taux de réussite de 90%, Urasium (Xu et al., 2012) et Caper (Zhang et Yin, 2014) garantissent quant à eux un taux de 96%.

Le temps de réécriture des applications a été calculé pour les deux variantes de notre Framework. Ce dernier est de 51s pour la variante exécutée sur le serveur web et de 72 s pour la variante implantée dans le périphérique. (Zhang et Yin, 2014) illustrent le temps de traitement de Capper. Celui-ci étant de 60s. Cependant, ces derniers n'incluent en aucun cas le temps de décompilation par Dex2jar qui, lui, peut prendre plusieurs secondes. Cette tâche représente en moyenne plus de 40% de la phase de réécriture de la présente étude. Le temps de réécriture par le Framework pourrait alors être réduit de 40% si le temps de décompilation n'est pas pris en compte.

Les périphériques mobiles souffrent de la contrainte des ressources limitées. Il est donc important de mesurer l'impact de la réécriture des applications sur le système. Nous avons donc calculé la taille moyenne rajoutée par notre Framework de réécriture, celle-ci, étant de 98822 octets (0.098 Mo), soit une taille relativement petite par rapport aux tailles des applications, et qui représente un pourcentage moyen de 1.2% par rapport à la taille originale de l'application. (Benjamin Davis 2012) estiment le pourcentage de taille rajoutée par i-arm-droid à 2%. Alors que (Xu et al., 2012) estiment quant à eux la taille rajoutée par Aurasium à 52000 octets.

Notre contrôleur d'application exige un contrôle sur tout appel aux méthodes API, un temps d'analyse est donc rajouté au temps d'exécution des méthodes API. Des tests ont donc été

effectués sur l'ensemble de 20 appels API étudiés (Aafer, Du et Yin, 2013) afin de calculer l'overhead de temps d'exécution rajouté par le contrôleur d'applications, et qui représente 20% par rapport au temps d'exécution de l'appel API sans contrôle. Il est à noter que (Xu et al., 2012) se sont, à leur tour, aussi intéressés aux contrôles des méthodes API. Ces derniers estiment l'overhead rajoutée par Aurasium à 35%.

Notre contrôleur d'applications offre un moyen simple et efficace pour l'introduction des politiques de sécurité. Une interface graphique est donc mise en disposition de l'utilisateur, et qui se base sur la décomposition des politiques de sécurité en conditions basiques, et offre la possibilité de combinaison des conditions afin de définir des règles de sécurité complexes.

De plus, d'autant que l'on peut dire à partir de la littérature, notre solution est la seule à pouvoir lutter contre la collaboration entre applications pour la création des contextes malicieux. Les politiques de sécurité introduites dans notre contrôleur peuvent être définies sur un ensemble comportant plusieurs applications et méthodes API. Le contrôleur étant une application tierce partie, peut donc surveiller et arrêter toute tentative de collaboration malsaine entre les applications.

Contrairement aux approches étudiées, notre solution a la capacité d'arrêter les tentatives d'exécution des codes natives par applications malveillantes, puisque l'exécution de ces derniers doit solliciter des méthodes API. La définition des politiques de sécurité dépendant de ces méthodes API peut être une solution efficace contre ce type d'attaque.

Le tableau 4.6 offre une comparaison entre notre approche et les différentes approches étudiées.

Tableau 4.6 Comparaison entre notre approche et les approches étudiées

Approche	Méthodologie utilisée	Modifications requises	Nécessité de rooter le périphérique	Empêcher l'exécution du code natif	Empêcher la création du contexte malicieux
Notre Approche	Analyse dynamique Renforcement par politique de sécurité Réécriture du code Java	Applications	NON	OUI	OUI
(Feth et Pretschner, 2012)	Analyse dynamique	Système Android	Oui	Non	Non
Apex (Nauman, Khan et Zhang, 2010)	Renforcement par politique de sécurité	Système Android	Oui	Non	Non
Tissa (Zhou et al., 2011)	Contrôle d'accès aux ressources	Système Android	Oui	Non	Non
AppFence (Hornyack et al., 2011)	Analyse dynamique et contrôle d'accès aux ressources	Système Android	Oui	Non	Non
I-ARM-Droid (Benjamin Davis 2012)	Réécriture du code Smali	Applications	Non	Non	Non
Aurasium (Xu et al., 2012)	Réécriture du code Java	Applications	Non	Non	Non
Capper (Zhang et Yin, 2014)	Réécriture du code Java	Applications	Non	Non	Non
APKLancer (Yang et al., 2014)	Réécriture du code Smali	Applications	Non	Non	Non

CONCLUSION

La popularité de la plateforme Android 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 Android, affecte non seulement la vie privée d'un très grand nombre d'utilisateurs, mais aussi l'économie des organisations intégrant Android dans leurs solutions logicielles. Par conséquent, ceci rend la sécurité de cette plateforme une des préoccupations les plus importantes et prioritaires.

Plusieurs solutions visant à protéger l'utilisateur contre les différentes attaques ont été élaborées. De celles qui identifient les portions de code malveillant dans les applications, jusqu'à celles qui détectent le comportement étrange des applications. Ces deux familles de solutions peuvent apporter certains avantages, mais elles sont loin d'être parfaites. Car le plus grand problème des solutions de sécurité apportées aux appareils mobiles c'est les contraintes de performances limitées des périphériques, à savoir, le processeur, la batterie, la mémoire, etc.

De plus, les malwares peuvent faire recours à des méthodes API pour des fins malveillantes. L'appel à ces dernières n'est pas forcement détecté comme étant une action malveillante. Surtout si ces méthodes sont sollicitées par une collaboration entre applications pour la création d'un contexte malicieux.

Les méthodes API dans le système Android permettent aux développeurs l'utilisation des fonctionnalités les plus poussées. Les applications peuvent donc à travers de ces méthodes accéder à différentes propriétés sensibles fournies par le système Android. À titre d'exemple, l'envoi des SMS, l'accès à la caméra, l'accès aux coordonnées GPS, etc. Ces fonctionnalités doivent donc être contrôlées afin d'assurer une exécution plus sécuritaire des applications

La présente étude s'est focalisée sur une catégorie d'applications qui ne sont malicieuses que dans certains contextes. Ces contextes malicieux peuvent être créés en exploitant les méthodes API par une application, ou par la collaboration entre plusieurs applications.

Tout d’abord, une étude approfondie sur l’architecture et la sécurité du système Android est établie afin d’identifier ses problèmes et ses lacunes. La présente étude cible les approches basées sur la modification du système Android, ainsi que celles basées sur la réécriture des applications. Puis, une analyse comparative des plus importants travaux connexes est effectuée, ce qui a permis d’identifier les principaux défis de la sécurisation de la plateforme Android par le contrôle des méthodes API.

En se basant sur cette revue de littérature, les principales caractéristiques de l’approche proposée et adoptée ont été identifiées. Ladite solution est centralisée et basée sur la réécriture des applications pour le contrôle des appels aux méthodes API.

L'approche proposée dans cette étude comporte deux axes principaux. Le premier, étant un Framework permettant la réécriture des applications en injectant les portions de code nécessaire pour la communication avec le contrôleur d'application. Ce contrôleur représente, en fait, le second axe de la solution proposée. Celui-ci, permettant la gestion de l'ensemble d'applications exécutées sur le périphérique ainsi que le contrôle des méthodes API sollicitées par ces applications.

Le contrôle exigé par notre contrôleur d'applications se base principalement sur un ensemble de politiques de sécurité introduites par l'utilisateur. Ces derniers permettent la définition des scénarios malicieux en se basant sur plusieurs conditions, à savoir, le temps, la date, les coordonnées GPS, le niveau de la batterie, les méthodes API, et les applications. Le contrôleur d'applications développé dans cette étude offre un moyen simple et efficace pour l'introduction des politiques de sécurité. L'utilisateur introduit donc les scénarios malicieux en se basant sur des conditions basiques, ces dernières peuvent être combinées pour la création des politiques de sécurité les plus complexes.

Les résultats expérimentaux montrent que la solution développée offre un moyen efficace de contrôle des appels API, ainsi qu'un meilleur compromis entre le taux de contrôle et la consommation des ressources du périphérique. En effet, le Framework assure un taux de réussite de réécriture de 99.78%. Ce taux de réussite est considérablement élevé comparé aux autres approches étudiées. De plus, le temps de réécriture des applications par le Framework

a été calculé, ce dernier est de 51s. Aussi, la taille moyenne rajoutée par le Framework de réécriture a également été calculée. Celle-ci, étant de 98822 octets (0.098 Mo), soit une taille relativement petite par rapport aux tailles des applications. Cette taille représente un pourcentage moyen de 1.2% par rapport à la taille originale de l'application.

Le contrôleur d'application exige un contrôle sur tout appel aux méthodes API, un temps d'analyse est donc rajouté au temps d'exécution des méthodes API. Il a été constaté que ce temps d'analyse représente 20% par rapport au temps d'exécution de l'appel API sans contrôle.

De plus, comme il peut être constaté à travers la comparaison avec la revue de littérature, la solution développée dans cette étude est la seule à pouvoir lutter contre la collaboration entre applications pour la création des contextes malicieux. Par ailleurs, les politiques de sécurité introduites dans le contrôleur développé dans cette étude peuvent être définies sur un ensemble comportant plusieurs applications et méthodes API. Enfin, ledit contrôleur, étant une application tierce partie, peut alors surveiller et arrêter toute tentative de collaboration malsaine entre les applications.

Contrairement aux approches étudiées, la solution développée dans cette étude a la capacité d'arrêter les tentatives d'exécution des codes natives par applications malveillantes, puisque l'exécution de ces derniers doit solliciter des méthodes API. La définition des politiques de sécurité dépendant de ces méthodes API peut être une solution efficace contre ce type d'attaque.

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Aafer, Yousra, Wenliang Du et Heng Yin. 2013. « DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android ». In *Security and Privacy in Communication Networks: 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, sous la dir. de Zia, Tanveer, Albert Zomaya, Vijay Varadharajan et Morley Mao. Aafer2013. p. 86-103. Cham: Springer International Publishing. < http://dx.doi.org/10.1007/978-3-319-04283-1_6 >.

Ajc. 2015.«the AspectJ compiler/weaver ».<https://eclipse.org/aspectj/doc/next/devguide/ajc-ref.html >.

B.V, Snow. 2013. *The LPIC-2 Exam Prep*. LPIC.

Benjamin Davis , Ben Sanders, Armen Khodaverdian, and Hao Chen. 2012. « I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications ». *Mobile Security Technologies*.

Bougarmoud, abdelaali ait. 2013. *Developpement d'une application de reservation pour la plateforme android*.

Caruel, Wilfried. 2015. « Rewriting applications ». < http://www.scoop.it/t/securite-infos-et-web >.

Chien-Wei, Chang, Lin Chun-Yu, King Chung-Ta, Chung Yi-Fan et Tseng Shau-Yin. 2010. « Implementation of JVM tool interface on Dalvik virtual machine ». In *VLSI Design Automation and Test (VLSI-DAT), 2010 International Symposium on*. (26-29 April 2010), p. 143-146.

Chilowicz, Michel. 2012. « IPC sous Android ». <http://igm.univ-mlv.fr/chilowi/teaching/subjects/java/android/ipc/index.html >.

Contagiodump. 2015. « Monkey ». <developer.android.com/guide/developing/tools/monkey.html >.

Conti, Mauro, Vu Thien Nga Nguyen et Bruno Crispo. 2011. « CRePE: context-related policy enforcement for android ». In *Proceedings of the 13th international conference on Information security*. (Boca Raton, FL, USA), p. 331-345. 1949355: Springer-Verlag.

Cooper, V. N., H. Shahriar et H. M. Haddad. 2014. « A Survey of Android Malware Characterisitics and Mitigation Techniques ». In *Information Technology: New Generations (ITNG), 2014 11th International Conference on*. (7-9 April 2014), p. 327-332.

Cooper, V. N., Shahriar, H., & Haddad, H. M. 2014. « Information Technology: New Generations (ITNG), 2014 11th International Conference ». In., p. 37-54.

Corporation., Amazing Force. 2010 «Scooter Framework ».

Crussell, Jonathan, Clint Gibler et Hao Chen. 2012. « Attack of the Clones: Detecting Cloned Applications on Android Markets ». In *Computer Security – ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, sous la dir. de Foresti, Sara, Moti Yung et Fabio Martinelli. Crussell2012. p. 37-54. Berlin, Heidelberg: Springer Berlin Heidelberg. < http://dx.doi.org/10.1007/978-3-642-33167-1_3 >.

Dhanesh. 2015. « securityxploded ». < http://securityxploded.com/android_reversing.php >.

DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. 2011. « Quire: lightweight provenance for smart phone operating systems ». In *Proceedings of the 20th USENIX Conference on Security*.

Dini, Gianluca, Fabio Martinelli, Andrea Saracino et Daniele Sgandurra. 2012. « MADAM: A Multi-level Anomaly Detector for Android Malware ». In *Computer Network Security: 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security, MMM-ACNS 2012, St. Petersburg, Russia, October 17-19, 2012. Proceedings*, sous la dir. de Kotenko, Igor, et Victor Skormin. Dini2012. p. 240-253. Berlin, Heidelberg: Springer Berlin Heidelberg. < http://dx.doi.org/10.1007/978-3-642-33704-8_21 >.

El-Harake, Khalil, Yliès Falcone, Wassim Jerad, Mattieu Langet et Mariem Mamlouk. 2014. « Blocking Advertisements on Android Devices Using Monitoring Techniques ». In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, sous la dir. de Margaria, Tiziana, et Bernhard Steffen. El-Harake2014. p. 239-253. Berlin, Heidelberg: Springer Berlin Heidelberg. < http://dx.doi.org/10.1007/978-3-662-45231-8_17 >.

El-Serngawy, M., et C. Talhi. 2015. « CaptureMe: Attacking the User Credential in Mobile Banking Applications ». In *Trustcom/BigDataSE/ISPA, 2015 IEEE*. (20-22 Aug. 2015) Vol. 1, p. 924-933.

Enck, William, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel et Anmol N. Sheth. 2010. « TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones ». In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. (Vancouver, BC, Canada), p. 1-6. 1924971: USENIX Association.

Feng, Yu, Saswat Anand, Isil Dillig et Alex Aiken. 2014. « Apposcopy: semantics-based detection of Android malware through static analysis ». In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. (Hong Kong, China), p. 576-587. 2635869: ACM.

Feth, D., et A. Pretschner. 2012. « Flexible Data-Driven Security for Android ». In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. (20-22 June 2012), p. 41-50.

Gannes, Liz. 2007. « http://recode.net ». < http://recode.net/2014/05/19/google-buys-an-enterprise-android-company-divide-formerly-known-as-enterproid/ >.

Garin, Florent. 2009. *ANDROID Développer des applications mobiles pour les Google Phones*. Dunod.

Google. 2015. « developer android»<http://developer.android.com/guide/components/intents-filters.html >.

Gruver, Ben. 2004. « smali/baksmali ». < https://github.com/JesusFreke/smali >.

Guignard, Damien. 2010. *Programmation Android De la conception au déploiement avec le SDK Google Android 2*. Eyrolles.

Hogarth, Dennis C. 2013. *Les appareils mobiles face aux cybermenaces*.

Hornyack, Peter, Seungyeop Han, Jaeyeon Jung, Stuart Schechter et David Wetherall. 2011. « These aren't the droids you're looking for: retrofitting android to protect data from imperious applications ». In *Proceedings of the 18th ACM conference on Computer and communications security*. (Chicago, Illinois, USA), p. 639-652. 2046780: ACM.

Jean-Michel. 2014. « developpez ». < http://jmdoudoux.developpez.com/cours/developpons/java/chap-securite.php >.

Jiang, Xuxian, et Yajin Zhou. 2013. *Android Malware*. Springer Publishing Company, Incorporated, 58 p.

Lab, SIIS. 2014. « ded: Decompiling Android Applications ». < http://siis.cse.psu.edu/ded/ >.

Lacombe, Eric. 2009. *sécurité des noyaux de systèmes d'exploitation*.

LEE, MAX. 2013. « What is AOSP ». < http://highonandroid.com/android-roms/what-is-aosp/ >.

Mila. 2011. « Take a sample, leave a sample. Mobile malware mini-dump ». < http://contagiodump.blogspot.ca/2011/03/take-sample-leave-sample-mobile-malware.html. >.

Moulu, André. 2015. « Sécurité des applications Android constructeurs et construction de backdoors ciblées ».

Mulliner, C., Robertson, W., & Kirda, E. 2014. « Virtualswindle: An automated attack against in-app billing on android ». In *In Proceedings of the 9th ACM symposium on Information, computer and communications security*. p. 459-470.

Nauman, Mohammad, Sohail Khan et Xinwen Zhang. 2010. « Apex: extending Android permission model and enforcement with user-defined runtime constraints ». In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. (Beijing, China), p. 328-332. 1755732: ACM.

Pxb. 2015. « dex2jar ». < https://sourceforge.net/projects/dex2jar/ >.

Rahul, Manekari. 2013. *Android is Open to Attacks According to a Survey From NQMobile (2013)*.

Reddy, N. 2011. *ACPLib: App-centric Security Policies on Unmodified Android*.

Ruff, Nicolas. 2014. *Sécurité du sysèeme Android*.

Sanz, Borja, Igor Santos, Xabier Ugarte-Pedrero, Carlos Laorden, Javier Nieves et Pablo García Bringas. 2014. « Anomaly Detection Using String Analysis for Android Malware Detection ». In *International Joint Conference SOCO'13-CISIS'13-ICEUTE'13: Salamanca, Spain, September 11th-13th, 2013 Proceedings*, sous la dir. de Herrero, Álvaro, Bruno Baruque, Fanny Klett, Ajith Abraham, Václav Snášel, C. P. L. F. André Carvalho, García Pablo Bringas, Ivan Zelinka, Héctor Quintián et Emilio Corchado. Sanz2014. p. 469-478. Cham: Springer International Publishing. < http://dx.doi.org/10.1007/978-3-319-01854-6_48 >.

ScriptTol. 2015. « ScriptTol ». < http://www.scriptol.fr/programmation/dalvik.php >.

Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., & Dolev, . 2009. *Google android: A state-of-the-art review of security mechanisms*.

Silicium, Open. 2014. « connected diamond ». < http://connect.ed-diamond.com/Open-Silicium/OS-010/Lepton-systeme-d-exploitation-temps-reel-pour-l-embarque-enfouis-une-approche-detaillee >.

Sophos. 2014. « Mobile Security Threat Report 2014 ».

Stark, Clinton. 2013. « Google Android vs. Apple iOS ». < http://www.starkinsider.com/2013/04/google-android-vs-apple-ios.html >.

Tumbleson, C. 2015. « A tool for reverse engineering Android apk files ». < https://ibotpeaches.github.io/Apktool/. >.

Xu, Rubin, Hassen Sa, #239, di et Ross Anderson. 2012. « Aurasium: practical policy enforcement for Android applications ». In *Proceedings of the 21st USENIX conference on Security symposium*. (Bellevue, WA), p. 27-27. 2362820: USENIX Association.

Yang, Wenbo, Juanru Li, Yuanyuan Zhang, Yong Li, Junliang Shu et Dawu Gu. 2014. « APKLancet: tumor payload diagnosis and purification for android applications ». In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. (Kyoto, Japan), p. 483-494. 2590314: ACM.

Yerima, S. Y., S. Sezer, G. McWilliams et I. Muttik. 2013. « A New Android Malware Detection Approach Using Bayesian Classification ». In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*. (25-28 March 2013), p. 121-128.

ZDNet. 2016. « Chiffres clés : les ventes de mobiles et de smartphones ». < http://www.zdnet.fr/actualites/chiffres-cles-les-ventes-de-mobiles-et-de-smartphones-39789928.htm >.

Zhang, Mu, et Heng Yin. 2014. « Efficient, context-aware privacy leakage confinement for android applications without firmware modding ». In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. (Kyoto, Japan), p. 259-270. 2590312: ACM.

Zhou, W., Zhou, Y., Jiang, X., & Ning, P. 2012. « Detecting repackaged smartphone applications in third-party Android marketplaces ». In *In Proceedings of the second ACM conference on Data and Application Security and Privacy* p. 1-12.

Zhou, Wu, Xinwen Zhang et Xuxian Jiang. 2013. « AppInk: watermarking android apps for repackaging deterrence ». In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. (Hangzhou, China), p. 1-12. 2484315: ACM.

Zhou, Wu, Yajin Zhou, Xuxian Jiang et Peng Ning. 2012. « Detecting repackaged smartphone applications in third-party android marketplaces ». In *Proceedings of the second ACM conference on Data and Application Security and Privacy*. (San Antonio, Texas, USA), p. 317-326. 2133640: ACM.

Zhou, Yajin, Xinwen Zhang, Xuxian Jiang et Vincent W. Freeh. 2011. « Taming Information-Stealing Smartphone Applications (on Android) ». In *Trust and Trustworthy Computing: 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*, sous la dir. de McCune, Jonathan M., Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse et Yolanta Beres.

Zhou2011. p. 93-107. Berlin, Heidelberg: Springer Berlin Heidelberg. < http://dx.doi.org/10.1007/978-3-642-21599-5_7 >.