ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA MAÎTRISE EN GÉNIE
CONCENTRATION TÉCHNOLOGIES DE L'INFORMATION
M. Ing.

PAR
Amine HADHIRI

SÉCURITÉ DES APPLICATIONS ANDROID : MENACES ET CONTRE-MESURES

MONTRÉAL, LE 27 AVRIL 2012

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

M. Jean-Marc Robert, directeur de mémoire
Département de génie logiciel à l'École de technologie supérieure

M. Abdelouahed Gherbi, président du jury
Département de génie logiciel à l'École de technologie supérieure

M. Chamseddine Talhi, membre du jury
Département de génie logiciel à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 20 AVRIL 2012

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

La réalisation de ce mémoire a été rendue possible grâce à la participation financière du Conseil de recherches en sciences naturelles et génie du canada (CRSNG) dans le cadre du *Programme partenarial d'engagement* ainsi qu'à la participation de l'École de technologie supérieure (ÉTS).Ma gratitude leur est acquise.

Je tiens à témoigner ma reconnaissance et ma gratitude à mon directeur de mémoire, Jean-Marc Robert qui m'a accordé sa confiance en acceptant de diriger mon travail. Ses qualités scientifiques et humaines m'ont toujours particulièrement impressionné. Promoteur de ce mémoire, il n'a cessé de m'encourager et m'apporter une aide précieuse grâce à son écoute attentive, ces conseils, sa patience et sa rigueur scientifique. Puissent ces ligne être l'expression de ma reconnaissance la plus profonde.

Mes remerciements sont également adressés aux membres du centre de recherche et développement de Gemalto de Montréal, et particulièrement à Mylène Lefebvre et Stéphane Chrétien pour leurs disponibilités et leur aide technique et logistique à l'occasion des procédures expérimentales.

Au terme de ces remerciements, j'aimerai rendre hommage à mes parents, à mon frère Helmi et ma sœur Dorra dont l'encouragement et le soutien interminable m'ont été d'une grande aide tout au long de mes études.

SÉCURITÉ DES APPLICATIONS ANDROID: MENACES ET CONTRE-MESURES

Amine HADHIRI

RÉSUMÉ

De nos jours, les téléphones intelligents sont omniprésents dans notre vie quotidienne. Le nombre d'utilisateurs de ces appareils ne cesse de croitre à travers le monde. En effet, les tâches que les téléphones intelligents sont capables d'offrir sont quasiment comparables à celles offertes par les ordinateurs conventionnelles. Cependant, il serait aberrant de tenir la comparaison lorsqu'il s'agit de la sécurité. En effet, les utilisateurs des téléphones intelligents peuvent consulter leurs courriels, leurs comptes bancaires et accéder à leurs comptes des réseaux sociaux. Cependant, la plupart de gens ne semblent pas réaliser qu'ils peuvent être victime des cyberattaques mettant en péril les trois aspects de sécurité de leurs données : la confidentialité, la disponibilité et l'intégrité

Plusieurs systèmes d'exploitation sont disponibles pour les téléphones intelligents. Selon les récentes statistiques, Android – lancé par Google en 2007 - serait le système le plus répandu sur le marché des téléphones intelligents et ne cesse de gagner en popularité. Cette popularité est croissante non seulement parmi les utilisateurs, mais aussi parmi les développeurs des applications mobiles profitant de l'ouverture de Android qui est un système source libre (Open source). Malheureusement, la sécurité dans Android ne suit pas la même courbe de croissance de ce système d'exploitation mobile. La popularité d'Android a fait en sorte qu'il devienne une cible de choix pour les cyberattaques. Par conséquent, le nombre des applications Android malveillantes s'est multiplié au cours des dernières années. Contrairement à ce que la majorité des utilisateurs des téléphones intelligents pensent, les impacts de ces attaques n'ont rien de moins comparé aux impacts causés par les logiciels malveillants ciblant les ordinateurs conventionnels.

Dans le cadre de cette recherche, nous nous sommes intéressés en un premier temps à caractériser les types des applications malveillantes ciblant le système Android en présentant une nouvelle classification de ceux-ci en fonction des vecteurs d'attaque qu'ils exploitent. Cette classification nous permet de mieux comprendre le mode de fonctionnement des applications malveillantes ciblant Android. Ensuite, nous proposons de nouvelles méthodes de développement d'applications pour Android que nous pensions capables de limiter les risques des applications malveillantes visant à exploiter malicieusement les applications légitimes installées sur le téléphone Android. Les méthodes proposées sont inspirées de plusieurs solutions de sécurité pour Android présentes dans la littérature, avec l'avantage de ne pas apporter de modifications au système Android.

Mots-Clés : Android, Sécurité, Vulnérabilité, Attaque, Application, Développeur

ANDROID APPLICATIONS SECURITY: THREATS AND COUNTERMEASURES

Amine HADHIRI

ABSTRACT

Nowadays, smartphones are ubiquitous in our daily lives. The number of users of these devices continues to grow worldwide. Currently, the tasks that smartphones are able to offer are almost comparable to those offered by conventional computers. However, it would be absurd to hold the comparison when it comes to security. Indeed, users of smartphones can access their email, their bank accounts and access their accounts of social networks. However, most people do not seem to realize they can be a victim of cyber attack threatening the three aspects of their data security: confidentiality, availability and integrity.

Several operating systems were introduced for smartphones. According to recent statistics, Android - launched by Google in 2007 - is the most widely used system in the smartphones market and is steadily gaining in popularity. This popularity is increasing not only among users but also among developers of mobile applications taking advantage of the openness of Android which is fully open source. Unfortunately, security in Android does not follow the same growth curve of the mobile operating system. In fact, the popularity of Android has made it become a prime target for cyber-attacks. Thus, the number of malicious applications has been multiplied in recent years. Contrary to what most smartphones users think, the impacts of these attacks have nothing less than the impacts caused by malware targeting conventional computers.

As part of this research, we were interested in a first step to characterize the different types of malicious applications targeting the Android operating system by introducing a new classification thereof based on the attack vectors they exploit. This classification would allow us to better understand the *modus operandi* of malicious applications targeting Android. We then propose new secure methods to develop applications for Android that we estimate capable of limiting the risk of malicious applications exploiting legitimate applications installed on the Android phone. The proposed methods are inspired by several security solutions for Android reported in the literature, with the advantage of not making changes to the Android system.

Keywords: Android, Security, Vulnerability, Cyber-attack, Application, Developer

TABLE DES MATIÈRES

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ADT	Android Development Tools
API	Application Programming Interface
CA	Certification Authority
DAC	Discretionary Access Control
DDos	Distributed Denial of Service
GID	Group IDentification number
GPS	Global Positioning System
IDE	Integrated Development Environment
IMEI	International Mobile Equipment Identity
JNI	Java Native Interface
JVM	Java Virtual Machine
LSM	Linux Security Module
MAC	Mandatory Access Control
NFC	Near Field Communication
PID	Process IDentification number
RBAC	Role-Based Access Control
SDK	Software Development Kit
SMS	Short Message Service
SQL	Structured Query Language
URI	Uniform Resource Identifier
UID	User IDentification number
XML	eXtensible Markup Language

INTRODUCTION

1) Mise en contexte

De nos jours, l'utilisation des téléphones intelligents pour des fins personnelles et professionnelles est omniprésente. Les téléphones intelligents sont de plus en plus utilisés dans les services bancaires en ligne, la consultation de courriel et réseaux sociaux ainsi que dans les achats en ligne. Avec 700 000 téléphones Android vendus par jour, Google est considéré comme le leader de l'industrie des téléphones intelligents. Le modèle économique de Google permet aux utilisateurs de télécharger des applications diverses permettant d'augmenter la valeur de leurs téléphones mobiles. Ces applications fournissent des fonctionnalités comparables à celles offertes par les ordinateurs conventionnels, mais avec une augmentation de la flexibilité et la portabilité. Malheureusement, la sécurité est une caractéristique essentielle qui est toujours manquante dans Android par rapport aux ordinateurs conventionnels. En fait, la plupart des ordinateurs conventionnels sont équipés d'une panoplie d'antivirus, coupe-feu et autres solutions de sécurité. Cependant, la sécurité des téléphones Android semble avoir du mal à suivre l'expansion rapide de ceux-ci.

2) Objectifs de la recherche et réalisations

L'expansion croissante et rapide des téléphones Android a fait en sorte qu'ils deviennent une cible de choix pour les attaques malicieuses. En effet, le nombre d'applications malveillantes ciblant Android s'est multiplié au cours des dernières années. Les motifs, les impacts et le mode de fonctionnement de ces attaques sont diversifiés et deviennent de plus en plus sophistiqués. La croissance exponentielle des téléphones Android utilisés à travers le monde combinée à l'augmentation du nombre d'attaques ciblant Android soulève la nécessité d'aborder le problème de la sécurité de ce système d'exploitation afin de préserver la confidentialité, disponibilité et intégrité des informations et données personnelles des utilisateurs et prévenir l'utilisation abusive des téléphones Android.

Dans le cadre de ce travail, nous nous sommes intéressées en premier lieu à caractériser les vulnérabilités du système d'exploitation Android. À l'issue de cette caractérisation, nous

présentons une nouvelle classification des applications malveillantes ciblant Android selon les vulnérabilités qu'elles exploitent. En second lieu, nous nous sommes intéressés aux nouvelles propositions de mécanismes de sécurité pouvant être intégrés au sein de la plateforme Android. Ces mécanismes proposés dans la littérature scientifique doivent être (1) acceptés par Google (2) intégrés par ceux-ci impliquant un changement à a plateforme et (3) déployés dans les versions d'Android par les divers fournisseurs de service (p. ex., Telus, Bell, Rogers, etc.) ainsi que par les fabricants d'équipements (p. ex., Samsung, HTC)

Après avoir fait une revue critique de la littérature, l'objectif principal de ce travail de recherche s'est concentré à introduire des mécanismes de sécurité pouvant être intégrés au cœur même des applications Android, empêchant toute exploitation malicieuse de celles-ci. Ces mécanismes présentent l'avantage de ne pas apporter de modification à la plateforme Android, contrairement à la majorité des solutions qui ont été proposées dans la littérature. Notre contribution se résume à la proposition d'un nouveau mécanisme de certification des applications Android ainsi qu'un modèle d'authentification pouvant être utilisé pour authentifier les applications Android les unes une contre les autres, renforçant ainsi la sécurité des interactions entre celles-ci. La certification quant à elle permet à une application légitime d'évaluer le degré de fiabilité et de sécurité de l'interaction avec une application externe en se basant sur des politiques bien définies régissant les interactions entre les différentes applications. Comme preuve de concept, nous avons développé une application Android mettant en œuvre les méthodes proposées et qui illustre la faisabilité de notre approche et son aptitude à renforcer la sécurité des applications Android sans l'apport de changements à la plateforme Android.

Toutefois, les méthodes proposées se limitent à protéger les applications légitimes installées sur des téléphones Android qui n'ont pas été rootés par leurs utilisateurs ou par des applications malveillantes, c.-à-d. sur lesquels il est impossible d'obtenir les privilèges de l'administrateur root sur le système (hypothèse 1). D'autre part, dû aux différentes limitations imposées par Google, il nous est impossible d'empêcher l'installation des applications malveillantes sur le téléphone (hypothèse 2). Cependant, nous nous proposons de fournir des

contre-mesures capables de détecter la présence de ces applications malveillantes et de limiter leurs impacts.

3) Organisation du mémoire

Ce travail est organisé comme suit : dans le chapitre1 et 2, nous présentons l'architecture du système Android et la structure des applications Android. Le troisième chapitre présente les détails du modèle de sécurité actuel utilisé dans Android. Notre nouvelle classification des applications malveillantes est présentée dans le chapitre 4. Le chapitre 5 présente les différentes motivations derrière les différentes applications malveillantes. Le chapitre 6 expose les solutions de sécurité proposées dans la littérature. Dans le chapitre 7, nous rappelons les meilleures pratiques qui devraient être suivies par les développeurs dans l'optique de développer des applications Android sécurisée. Par la suite, nous exposons les détails de l'architecture et du fonctionnement des mécanismes de sécurité que nous introduisons.

CHAPITRE 1

LE SYSTÈME ANDROID

Dans ce chapitre, nous commençons par la présentation du système d'exploitation Android pour les plateformes mobiles. Nous présentons aussi les différentes couches faisant partie de l'architecture de ce système source libre ainsi qu'un exemple illustratif des interactions entre les différentes couches du système.

1.1 Introduction à Android

Android est un système d'exploitation source libre initialement développé par Android inc., une firme achetée plus tard par Google en 2005. En janvier 2011, Google détenait 33,3 % du marché des téléphones intelligents dans le monde entier, démontrant une croissance incroyable pour Android qui détenait seulement 4,7 % un an plus tôt. D'autre part, Nokia, Apple, RIM et Microsoft détenaient 31 %, 16,2 %, 14,6 % et 3,1 % respectivement (Cheng, 2011).

La croissance d'Android sur le marché des téléphones intelligents peut être attribuée à divers facteurs. Contrairement au système d'exploitation mobile iOS d'Apple, Android est totalement gratuit et ouvert. Ce fait a contribué à tourner l'attention des constructeurs de téléphones mobiles vers Android, qui voyaient en lui une opportunité pour s'introduire sur le marché des téléphones intelligents. Par conséquent, plusieurs modèles de téléphone intelligent utilisant Android ont vu le jour.

De plus, Google met à la disposition des développeurs des applications Android un moyen facile pour développer leurs applications en utilisant le kit de développement de logiciels (SDK) d'Android. Contrairement à Apple, la publication des applications sur le marché est beaucoup plus simple. Google offre la possibilité aux développeurs de publier leurs applications directement sur le marché d'applications de Google appelé *Android Market*, mais aussi sur les marchés d'applications parallèles qui ne sont pas gérés par Google. Les

utilisateurs d'Android ont donc le choix de télécharger les applications à partir d'*Android Market* ou à partir des sites indépendants sur Internet. Ceci est impossible avec l'iOS d'Apple, qui force les utilisateurs de cette plateforme à télécharger les applications seulement à partir du marché d'applications d'Apple appelé *Apple Store*. Par ailleurs, Apple applique une politique stricte de publication des applications en vérifiant le code source de chaque application avant de la rendre disponible sur Apple Store. Ainsi, la simplicité du processus de publication des applications d'Android peut être considérée comme un des facteurs primordiaux contribuant à la popularité croissante de ce système d'exploitation. Toutefois, nous pourrions espérer que la vérification du code par Apple élimine de nombreuses applications malveillantes

1.2 Architecture du système d'exploitation mobile Android

Android est une suite logicielle destinée aux appareils mobiles, et comprend un système d'exploitation, un intergiciel et des applications essentielles, telles qu'un client de messagerie électronique, une application de messagerie SMS, un navigateur Internet, une application gérant les contacts, etc. (AOSP, 2008b). Ces applications fonctionnent au-dessus d'un intergiciel écrit en Java qui roule dans un noyau Linux embarqué optimisé pour les téléphones mobiles. Dans ce qui suit, nous exposons les détails de chaque couche d'Android. (voir la figure 1.1)

1.2.1 Le noyau Linux

Le noyau Linux version 2.6 est la couche la plus basse d'Android. Cette couche est une couche d'abstraction matérielle qui permet l'interaction des couches supérieures avec la couche matérielle à travers les pilotes des périphériques. Le noyau Linux fournit également les services les plus fondamentaux tels que la sécurité, la gestion de la mémoire, la gestion des processus ainsi que les fonctionnalités de communication réseau (Shabtai et al., 2009).

Figure 1.1 Architecture d'Android
Tirée de AOSP (2008h)

Le choix du noyau Linux pour Android peut être justifié par un certain nombre de facteurs. Linux est réputé pour sa gestion stable et performante de la mémoire et des processus, ainsi que par son modèle de sécurité robuste basé sur un contrôle d'accès discrétionnaire (DAC) qui n'a pas changé depuis les années soixante-dix.

1.2.2 Les bibliothèques natives

Les bibliothèques natives sont écrites à l'aide du langage de programmation C/C++ et elles se localisent dans la couche située au-dessus du noyau Linux. Ces bibliothèques sont utilisées par les différents composants du système se trouvant dans la couche supérieure. Les développeurs d'applications Android peuvent aussi intégrer ces fonctionnalités directement dans leurs applications. Ceci est possible via l'utilisation de l'interface Java native JNI. Cette interface permet aux applications écrites en Java d'interagir avec les bibliothèques rédigées en code natif C/C++.

Parmi l'ensemble des bibliothèques fournies, les plus importantes sont (AOSP, 2008b) :

- la bibliothèque standard du système C adaptée pour les appareils fonctionnant avec le système Linux embarqué;
- les bibliothèques média responsables de la lecture et l'enregistrement des données audio et vidéo sous plusieurs formats ;
- les polices de caractères;
- le moteur de base de données relationnelle allégé SQLite utilisé par les applications afin de gérer le stockage des données. Son architecture simple et sa taille peu volumineuse le rendent parfaitement adapté aux téléphones mobiles ayant des ressources limitées;
- les bibliothèques graphiques 3D basées sur OpenGL souvent utilisées pour développer des jeux 3D puissants.

1.2.3 L'environnement d'exécution Android

L'environnement d'exécution Android contient les bibliothèques internes de base ainsi que la machine virtuelle Dalvik. Les bibliothèques de base fournissent la plupart des fonctionnalités disponibles dans les bibliothèques de base de Java. La machine Dalvik (Bornstein, 2008) quant à elle a été conçue pour les appareils limités en ressource mémoire, en capacité de calcul et en autonomie d'énergie.

Contrairement aux machines virtuelles Java (JVM) classiques basées sur le concept de la machine à pile, Dalvik est une machine virtuelle basée sur le concept de machine à registres ce qui lui permet d'économiser les ressources des appareils. En effet, Dalvik nécessite 30% moins d'instructions qu'une JVM typique. Plus précisément, elle nécessite 30% moins d'instructions pour effectuer le même calcul qu'une machine à pile typique, entraînant ainsi la réduction du temps de calcul et des accès mémoire.

Bien que les applications Android soient écrites en langage Java, Dalvik utilise son propre format de bytecode prenant la forme de fichier exécutable **.dex*. Cela offre la possibilité

d'inclure plusieurs classes dans un seul fichier et d'effectuer des optimisations de code supplémentaires au moment de la génération du bytecode d'une application. En appliquant ces principes, il est possible d'avoir des fichiers *.dex moins volumineux comparativement aux fichiers **.jar* contenant du bytecode Java traditionnel.

1.2.4 Le cadre d'application

Entièrement écrit en Java, le cadre d'application fournit les diverses interfaces de programmation (API) utilisées par les applications s'exécutant dans la couche supérieure (p. ex., l'API pour effectuer des appels téléphoniques ou utiliser la caméra). Outre ces API, le cadre d'application comporte un ensemble de services permettant l'accès aux fonctionnalités de base d'Android telles que les composants graphiques, les gestionnaires de localisation et les gestionnaires de notification.

1.2.5 Couche des applications

La couche application comporte les applications distribuées par défaut avec Android, notamment l'application SMS, l'application de téléphonie, l'alarme, caméra, etc. Les applications installées par l'utilisateur sont également exécutées au niveau de cette couche.

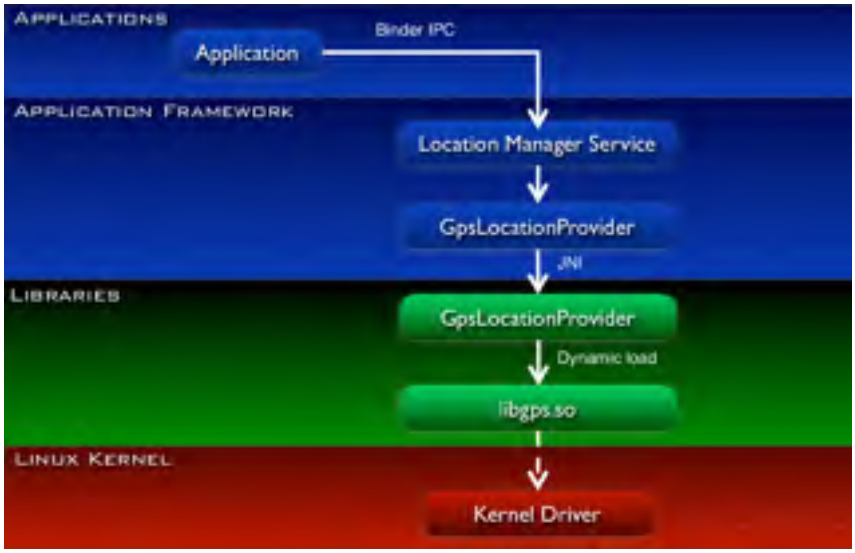

Figure 1.2 Interactions entre les couches du système Android
Tirée de Townsend (2010)

1.3 Interaction entre les différentes couches

Les différentes couches du système Android travaillent ensemble afin de répondre au besoin de l'utilisateur. Considérons l'exemple d'une application ayant besoin d'obtenir les données relatives à l'emplacement physique de l'appareil (voir figure 1.2). Pour se faire, l'application consulte le service *LocationManagerService* situé dans la couche du cadre d'application. Ce dernier interroge le *GpsLocationProvider* situé dans la même couche, qui lui appelle la bibliothèque C/C++ *GpsLocationProvider* à travers l'interface JNI. Ladite bibliothèque agit en chargeant la bibliothèque dynamique *libgps.so* du noyau Linux. Finalement, le noyau Linux interagit avec la couche matérielle du téléphone mobile à travers les pilotes logiciels et obtient l'emplacement physique de l'appareil.

CHAPITRE 2

STRUCTURE DES APPLICATIONS ANDROID

Après avoir présenté dans le chapitre 1 les caractéristiques et les détails de l'architecture d'Android, nous présenterons à travers ce chapitre la structure des applications Android s'exécutant dans la couche supérieure de l'architecture du système Android. Cette couche est la couche applications.

Les applications Android sont distribuées sous forme de fichier *.*apk* (package Android) et le processus d'installation de celles-ci consiste à déployer ces fichiers.apk (Shabtai et al., 2009). Le package d'une application Android contient le code de l'application (fichier *.*dex*), les fichiers de ressources (images, sons, etc.), ainsi que le fichier AndroidManifest.xml que nous appellerons simplement fichier Manifest par la suite.

Le cadre d'application Android force les développeurs d'applications à se conformer avec une structure bien définie qui est spécifique aux applications Android. Dans ce chapitre, nous exposons les détails de la structure des applications Android et les possibles interactions entre celles-ci.

2.1 Exemples d'applications Android

Afin de mieux comprendre l'architecture des applications Android, nous commençons par présenter deux exemples d'applications. Considérons une application dont l'utilité consiste à avertir l'utilisateur du téléphone mobile dès que celui-ci se rapproche de l'adresse du domicile de l'un de ses contacts. Nous séparons cette fonctionnalité dans deux applications (voir figure 2.1). Nous appellerons la première application *DomicileContactProximite*. Son rôle consiste à suivre de façon continue la position physique du téléphone de l'utilisateur et déterminer s'il se trouve près de l'adresse d'un de ses contacts. L'application utilise une base de données contenant la liste des contacts et leurs adresses. La seconde application qu'on

appelle *VoirDomicileContact* est ensuite utilisée pour afficher l'adresse de ce contact sur une carte et assister l'utilisateur pour se rendre à cette adresse en cas de besoin.

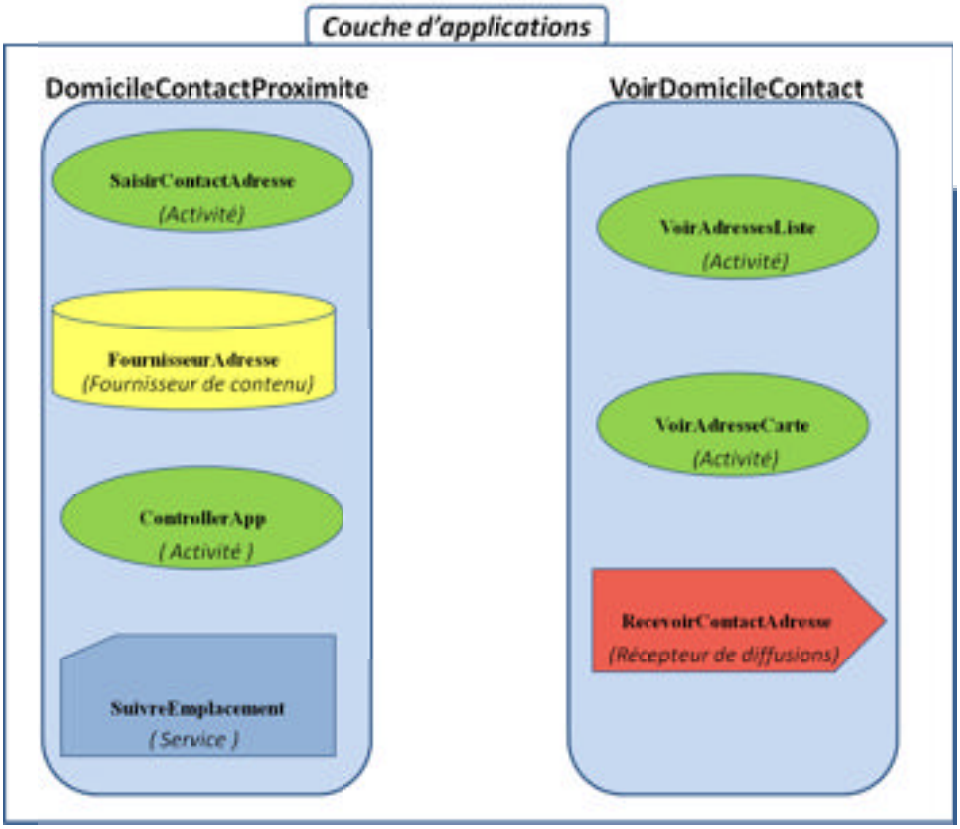

Figure 2.1 Exemple d'applications pour Android

2.2 Les composants des applications

Bien qu'elles soient écrite en langage de programmation Java, les applications Android ne possèdent pas de fonction m*ain* ou de point d'entrée unique à l'instar des programme Java. Les développeurs doivent concevoir les applications en terme de composants (Enck, Ongtang et McDaniel, 2009). Les quatre types possibles de composants sont : *activité*, *service*, *fournisseur de contenu* et *récepteur de diffusion*.

Dans la présente section, nous présentons les détails des quatre types de composants sur lesquelles repose l'architecture des applications Android.

2.2.1 Activité

Une activité est une interface visuelle à travers laquelle l'utilisateur interagit avec le système d'exploitation Android et les applications installées sur le téléphone. Une application peut être constituée d'une ou plusieurs activités dépendamment de son architecture. Généralement, une seule activité est spécifiée comme étant une activité principale, et elle est présentée en premier lieu à l'utilisateur lors du lancement de l'application. Cette activité est la porte d'entrée pour l'application, et pourra par la suite démarrer d'autres activités afin d'effectuer différentes actions.

Dans notre exemple de l'application *VoirDomicileContact*, l'activité *VoirAdresseListe* (voir figure 2.1) représente l'interface utilisateur de l'application. Cette activité est marquée comme la porte d'entrée de l'application. Sa tâche consiste à afficher la liste des noms et prénoms des contacts de l'utilisateur avec l'adresse de leur domicile. L'activité *VoirAdresseCarte* quant à elle est utilisée pour afficher l'adresse d'un contact donné sous forme d'un marqueur sur une carte géographique.

2.2.2 Service

Un service (*Service*) se réfère à la partie du code de l'application qui s'exécute en arrière-plan, et qui ne peut pas avoir d'interface visuelle. Cependant, un service peut être lié à une activité permettant ainsi à l'utilisateur d'une application de la contrôler (démarrer ou arrêter). Habituellement, un service s'exécute en arrière-plan d'une application ou d'une activité. Lorsqu'une activité a besoin d'exécuter une opération qui doit se poursuivre après la disparition de l'interface utilisateur (p. ex., le téléchargement d'un fichier ou jouer de la musique), elle fait appel à un service spécialement conçu pour cette action.

Dans notre exemple d'application, le service *SuivreEmplacement* (voir figure 2.1) de l'application *DomicileContactProximite* se charge de vérifier l'emplacement physique du téléphone à chaque intervalle de temps défini. Le service en question compare l'emplacement de l'utilisateur avec l'adresse de chaque contact se trouvant dans la base de données de l'application. Si l'utilisateur se trouve à la proximité de l'adresse de l'un de ses contacts figurant dans la base de données, le service émet un message afin de signaler l'événement à l'application *VoirDomicileContact*. Le démarrage et l'arrêt du service sont gérés par l'utilisateur à travers l'activité *ControllerApp* (p. ex., boutons Démarrer et Arrêter).

2.2.3 Récepteur de diffusion

Les récepteurs de diffusion (*Braodcast Receiver*) agissent comme des boîtes aux lettres pour les messages provenant d'autres applications ou du système (Enck, Ongtang et McDaniel, 2009). Leur rôle consiste à recevoir les diffusions et de réagir en fonction du contenu de chaque diffusion reçue.

Typiquement, les diffusions sont émises soit par le système, soit par les applications. Le système Android envoie souvent des diffusions pour alerter les applications des changements d'état, tels que la réception d'un SMS, le changement des préférences de langue ou encore l'annonce d'un faible niveau de la batterie.

Les applications peuvent également envoyer des diffusions destinées aux récepteurs de diffusion appartenant à la même application ou à d'autres applications installées. Ces derniers peuvent réagir à la réception des diffusions si elles sont configurées ainsi. L'application peut avoir plusieurs récepteurs de diffusion, ce qui lui permet de recevoir les diffusions émises simultanément.

Les récepteurs de diffusion étendent la classe de base *BroadcastReceiver* et ne peuvent pas avoir d'interface visuelle. Cependant, ils ont la possibilité de lancer une activité suite à la réception d'une diffusion. Un récepteur de diffusion peut également notifier l'utilisateur au

sujet de l'évènement qui a eu lieu en utilisant le gestionnaire de notification. Les notifications peuvent prendre la forme d'un signal sonore, une vibration ou encore une icône sur la barre d'état du système.

L'application *VoirDomicileContact* définit un récepteur de diffusion appelée *RecevoirContactAdresse*. Le rôle de ce dernier consiste à attendre les diffusions indiquant que le téléphone se trouve à proximité de l'adresse d'un contact particulier, en provenance de l'application *DomicileContactProximite*. À la réception de la diffusion, l'application affiche une notification visuelle pour l'utilisateur lui demandant s'il désire lancer l'activité *VoirContactCarte*. Si c'est le cas, l'activité *VoirAdresseCarte* est démarrée et l'adresse du contact en question est affichée sur une carte géographique.

2.2.4 Fournisseur de contenu

Les fournisseurs de contenu (Content Provider) est le composant chargé du partage des données entre les applications, en utilisant l'interface de base de données relationnelle SQLite (Enck, Ongtang et McDaniel, 2009a). Chaque fournisseur de contenu est conçu pour traiter un type spécifique de données telles que les données audio, vidéo ou image. Ce type de composant est l'unique moyen de stocker, récupérer et partager des données entre les applications, et ce, dû à l'absence de base de données centralisée dans le système.

Android est livré avec un certain nombre de fournisseurs de contenu (situés dans le cadre d'application) qui gèrent certains types de données (p. ex., les informations des contacts), et qui peuvent être utilisés par les applications. Deux méthodes existent pour rendre les données d'une application publique, soit en définissant son propre fournisseur de contenu ou soit en insérant ces données dans un fournisseur de contenu déjà existant capable de supporter le même type de données (p. ex., l'ajout d'un contact par une application autre que celle par défaut).

Le fournisseur de contenu *FournisseurAdresse* défini dans l'application *DomicileContactProximite* détient la liste des noms et prénoms des contacts de l'utilisateur et leurs adresses. Ces informations sont regroupées sous forme d'un fichier de base de données géré par le fournisseur de contenu *FournisseurAdresse*. Dans notre exemple, ces informations sont entrées par l'utilisateur qui spécifie l'adresse de chaque contact à travers l'activité *SaisirAdresseContact*.

2.3 Le fichier de déclaration AndroidManifest.xml

Chaque package d'une application Android comporte son propre fichier de déclaration structuré XML, portant toujours le nom de AndroidManifest.xml. Ce fichier comporte la déclaration des composants de l'application afin que ceux-ci puissent être reconnus et initiés par le système, lorsque nécessaire. Outre la déclaration des composants de l'application, le fichier Manifest remplit les fonctionnalités suivantes (AOSP, 2008e) :

- il identifie le nom du package de l'application qui est utilisé comme identifiant unique pour l'application.
- il présente les classes qui implémentent chacun des composants et publie leurs capacités (p. ex., les tâches qu'elles peuvent accomplir au profit des applications externes). Ces déclarations permettent au système Android de savoir ce que les composants peuvent faire et sous quelles conditions ils peuvent les faires.
- il détermine quel processus sera l'hôte des éléments de l'application.
- il déclare les permissions requises pour accéder aux parties protégées de l'API d'application et d'interagir avec d'autres applications (plus de détails au Chapitre 3).
- il déclare aussi les permissions que les autres applications sont tenues d'avoir en vue d'interagir avec les composants de l'application (plus de détails Chapitre 3).
- il déclare le niveau minimum de l'API Android que l'application exige. En effet, il existe un niveau d'API qui correspond à une ou plusieurs version d'Android (p, ex. API 10 correspond aux versions Android 2.3.3 et Android 2.3.4)
- il énumère les bibliothèques auxquelles l'application doit être liée.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.android.exampleApplication">
   <uses-sdk android:minSdkVersion="10" />

<application   android:label="Example application" >
      <!—Exemple de déclaration d'une activité -->
      <activity android:name=".ControllerApp" >   </activity>

      <!—Exemple de déclaration d'un fournisseur de contenu -->
      <provider android:name="Fournisseur Adresse"
                  android:authorities="com.example.android.exampleApplicationProvider">
      </provider>

      <!—Exemple de déclaration d'un service -->
       <service android:name="ExampleService" android:exported = "false">
       </service>

       <!—Exemple de déclaration d'un recepteur de diffusion -->
         <receiver android :name= "ExampleReceiver">
                  <intent-filter android:name="ExampleFilter">
                  <action android:name="exampleAction"/>
                  </intent-filter>
</receiver>
   </application>
</manifest>
```

Figure 2.2 Exemple de fichier de déclaration d'une application Android

Une activité est déclarée dans le fichier Manifest en utilisant la balise <*ativity*> (voir figure 2.2). De la même manière, les services, les récepteurs de diffusion et les fournisseurs de contenu sont définis à travers les balises <*receiver*>, <*service*> et <*provider*>, respectivement. Alors que les composants de type activité, service et fournisseurs de contenu doivent impérativement être déclarées dans *AndroidManifest.xml*, les récepteurs de diffusion peuvent être déclarés dans le fichier Manifest ou créés dynamiquement au cours de l'exécution de l'application puis enregistrés dans le système via la méthode *Context.registerReceiver()*.

2.4 Interaction entre les composants

Dans Android, les différents composants appartenant à la même application ou à des applications différentes ont la possibilité d'interagir les uns avec les autres. L'interaction entre deux composants repose essentiellement sur le mécanisme des intentions. Une intention est un message asynchrone qui contient le nom de l'action qu'une application désire accomplir, ainsi que l'URI des données qui seront traitées par le composant cible s'il y a lieu. Autrement dit, un objet de type intention (appelé simplement intention par la suite) définit une «intention» d'effectuer une «action» (p. ex., envoyer un SMS).

Le mécanisme d'adressage des intentions, quant à lui, est considéré comme l'une des fonctions les plus puissantes d'Android grâce à sa flexibilité (Enck, Ongtang et McDaniel, 2009). En fait, Android présente deux catégories d'intention : implicite et explicite. Les développeurs des applications peuvent cibler de manière unique un composant en le désignant par son nom tel qu'il apparait dans le fichier Manifest. Dans ce cas, nous parlons d'intentions explicites. Cependant, pour des raisons de sécurité, les noms des composants des applications sont souvent mis à l'abri des développeurs, à moins que les deux applications (celle en train d'être développée et celle déjà installée) appartiennent au même développeur.

L'autre alternative réside dans l'utilisation des intentions implicites. Une intention implicite ne mentionne pas le nom du composant demandé, mais au lieu, elle spécifie un nom implicite appelé chaine d'action. Par exemple, un composant voulant effectuer un appel téléphonique envoie une intention avec l'action *ACTION_CALL* sans préciser le nom du composant destinataire de l'intention. Dans ce dernier cas, le système Android se charge de trouver le meilleur composant capable de traiter l'intention en se basant sur la liste des applications installées (en consultant les fichiers AndroidManifest.xml de chacune d'elles). Le système est amené à se référer au choix de l'utilisateur si plusieurs possibilités sont offertes.

Les filtres d'intention déclarés dans le fichier Manifest informe le système sur le type d'intentions implicites qu'un composant peut traiter. Il décrit les fonctionnalités offerte par chacun des composants au système et décrit une liste des actions qu'ils sont prêts à gérer. Par ailleurs, il est possible de définir la liste des intentions implicites qu'un composant ne désire pas recevoir. Si un composant ne dispose d'aucun filtre d'intention, alors il ne peut être activé que par une intention explicite.

2.4.1 Interaction avec une activité

Lors d'une interaction entre deux composants, plusieurs actions sont possibles dépendamment du type du composant cible (voir Figure 2.3). La communication avec une activité s'effectue en passant un objet de type intention à la méthode *Context.StartActivity(intention)*. Lorsqu'une activité démarre une deuxième activité tout en s'attendant à un résultat en retour, la méthode *StartActivityForResult(intention)* est utilisée, autrement on utilise la méthode *Context.StartActivity(intention).*

Supposons que l'utilisateur désire afficher l'adresse de l'un de ses contacts sur une carte géographique. Tout d'abord, il commence par démarrer l'activité *VoirAdresseListe* et choisit un contact. Ensuite, l'activité *VoirAdresseListe* crée une intention avec l'activité *VoirAdresseCarte* comme destinataire, et les données du contact comme données. Finalement, l'intention est acheminée vers l'activité destinatrice via la méthode *StartActivity(intention).*

2.4.2 Interaction avec un service

Durant une interaction avec un service, trois actions sont disponibles : démarrer, arrêter ou se lier. Un service est lancé en passant une intention à la méthode *Context.startService(intention)*. La méthode *Context.bindService(intention)* sert à établir une connexion entre le composant appelant et le service ciblé. Cette connexion permet au composant appelant d'utiliser les méthodes définies dans le service.

Dans notre exemple d'applications, l'activité *ControlerApp* peut démarrer le service *SuivreEmplacement* via *StartService (intention)* pour commencer à suivre l'emplacement physique du téléphone mobile. La même activité peut être utilisée pour mettre fin à l'exécution du service. Le service *SuivreEmplacement* utilise la méthode *bindservice(intention)* pour se lier au service de localisation situé dans la couche cadre d'application du système fournissant les informations liées à l'emplacement physique du téléphone mobile.

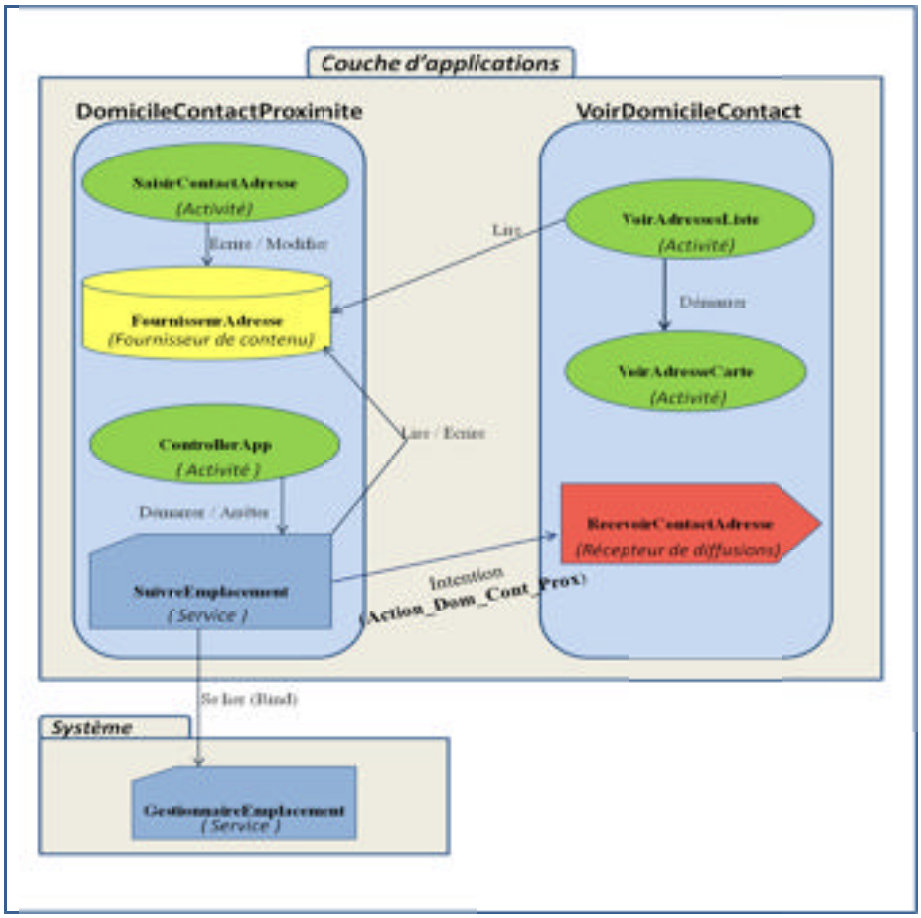

Figure 2.3 Les interactions inter-composants

2.4.3 Interaction avec un récepteur de diffusion

L'interaction avec les récepteurs de diffusion se fait par le biais de la méthode *sendBroadcast(intention)*. Une diffusion consiste en une intention envoyée par un composant via la méthode *sendBroadcast(intention)*. L'intention peut contenir le nom du récepteur cible et dans ce cas elle est dirigée directement vers la cible. La deuxième méthode d'envoi d'intention consiste à spécifier l'action pour laquelle le récepteur s'est enregistré via le filtre d'intention dans le fichier Manifest (voir figure 2.2). Par exemple, le composant *RecevoirAdresseContact* s'enregistre pour recevoir les intentions contenants l'action *ACTION_DOM_CONT_PROX*. Lorsque le service *SuivreEmplacement* s'aperçoit que l'utilisateur est à proximité d'une adresse d'un contact donné, il crée une intention contenant l'action *ACTION_DOM_CONT_PROX* et l'envoi au récepteur de diffusion *RecevoirAdresseContact*. Les données contenues dans l'intention sont le nom et prénom du contact concerné ainsi que l'adresse de son domicile. Suite à l'envoi de l'intention, le système consulte les fichiers Manifest des applications installées afin de reconnaître les composants qui se sont inscrits pour recevoir ce type d'actions. Dans le cas présent, le récepteur *RecevoirAdresseContact* est choisi par le système et un message est affiché à l'utilisateur pour lui annoncer cet événement.

2.4.4 Interaction avec un fournisseur de contenu

À la différence des composants vus précédemment (activité, service et récepteur de diffusion), les fournisseurs de contenu ne sont pas accessibles aux applications externes via les intentions.

Afin d'accéder aux informations détenues par un fournisseur de contenu, les applications doivent utiliser un composant appelé résolveur de contenu (*ContentResolver*). En fait, tous les fournisseurs de contenu ont essentiellement une interface commune qui permet aux applications d'interagir les unes avec les autres. Pour cela, chaque application doit instancier la classe de base *ContentResolver*, lui permettant l'accès à l'interface commune fournie par tous les fournisseurs de contenu (AOSP, 2008d).

CHAPITRE 3

MODÈLE DE SÉCURITÉ D'ANDROID

Comme vu précédemment, Android est construit sur un noyau Linux. Par conséquent, il hérite de l'architecture de sécurité de Linux ainsi que de son modèle de contrôle d'accès. Le noyau Linux fait d'Android un système multiprocessus exécutant chaque application dans son propre processus. La sécurité d'Android est implémentée par différents mécanismes et à différents niveaux. Celle-ci est renforcée, en grande partie, au niveau du noyau Linux à travers des mécanismes de Linux tels que le UID et les GID assignés à chaque application (Asaf et al., 2010). Au niveau des applications, Android propose un système de permissions en vue de restreindre l'accès à des composants spécifiques appartenant au système et aux applications installées sur le téléphone. En outre, un mécanisme basé sur les signatures numériques force les développeurs à signer leurs applications pour qu'elles puissent être installées. D'autre part, Android met en place des mécanismes visant principalement à protéger les données manipulées par l'application.

Dans le présent chapitre, nous présentons les détails de l'architecture de sécurité d'Android au niveau du noyau Linux et celui des applications.

3.1 Mécanisme de sécurité au niveau du noyau Linux

Android fait usage des mécanismes de sécurité de Linux comme moyen pour identifier les applications et isoler leurs ressources. Le noyau Linux implémente la sécurité entre les applications au niveau des processus en utilisant ses propres mécanismes que nous présentons dans cette section.

3.1.1 Isolation des processus des applications

Au moment de son installation, chaque application Android se voit attribuer un numéro unique d'identification UID et s'exécute dans son propre processus. L'UID accompagne l'application tout au long de sa durée de vie sur l'appareil (Shabtai et al., 2009). Cette architecture crée une sorte de bac de sable (*SandBox*), où chaque application s'exécute en s'isolant des autres applications, l'empêchant d'interférer avec celles-ci, et inversement, empêchant les autres applications de nuire à son fonctionnement.

Par défaut, Android ne permet pas l'exécution de deux applications dans le même processus. Toutefois, des exceptions existent. Dans le cas où deux applications désirent être exécutées dans le même processus, elles doivent le demander explicitement avec l'utilisation de la fonctionnalité *sharedUserID* offerte par Android. Ladite fonctionnalité leur donne la possibilité de partager le même UID. Par conséquent, les deux applications deviennent capables de partager le même processus, et les mêmes données, ainsi que le même ensemble de permissions. Par ailleurs, les deux applications doivent nécessairement être signées par le même certificat numérique, et donc appartenir au même développeur.

Dans certains cas d'utilisation, une application peut avoir recours aux services offerts par une application externe. Dans ce cas, l'application appelée se charge de remplir la tâche qui lui a été demandée par l'application appelante. Toutefois, chacune des applications s'exécute dans son propre processus et avec ses permissions initiales. Autrement dit, l'application appelée ne peut accéder aux fichiers de données de l'application appelante, ni bénéficier des privilèges accordés à celle-ci.

Une deuxième mesure de sécurité spécifique à la version Linux utilisée dans Android consiste à empêcher toute application installée par l'utilisateur de s'exécuter avec les privilèges de l'administrateur root. Dans Linux, les administrateurs root bénéficient d'un accès complet à toutes les applications et toutes les données des applications et ont la possibilité de lire et modifier les fichiers et données appartenant au système Linux lui-même.

Dans Android, seules les applications de base peuvent s'exécuter avec des privilèges root. Cependant, certains utilisateurs peuvent supprimer ce paramètre particulier. Les implications d'une telle démarche seront expliquées à la section 4.2.3.

3.1.2 Accès aux fichiers de l'application

Android gère le contrôle d'accès aux fichiers en utilisant le même mécanisme que Linux. Chaque fichier dans Android se voit attribuer l'UID de son propriétaire, l'ID du groupe auquel il appartient, avec trois tuples d'autorisation (en lecture, écriture et exécution). Le premier tuple définit les droits d'accès du propriétaire, tandis que le second et le troisième définissent l'ensemble des droits pour les utilisateurs du même groupe et le reste des utilisateurs, respectivement. Généralement, un fichier appartient soit à l'administrateur root, soit au système ou soit à un utilisateur non privilégié du système. Dans la version de Linux utilisée dans Android, chaque application est considérée comme un utilisateur (Shabtai et al., 2010). Ce mécanisme empêche chaque application d'accéder aux fichiers appartenant au système, à l'administrateur ou aux autres applications, sauf si elles partagent le même UID.

3.2 Mécanisme de sécurité au niveau de l'application

3.2.1 Signature numérique du package de l'application

Android exige que chaque application soit signée numériquement par une clé privée associée à un certificat numérique (Shabtai et al., 2009), faute de quoi elle ne peut être installée sur l'appareil. Cependant, aucune autorité de certification (CA) n'est requise, permettant ainsi aux développeurs d'utiliser des certificats auto-signés. Par conséquent, la signature ne sert qu'à associer l'application à son développeur, sans pour autant servir à vérifier sa vraie identité. Au moment de l'installation de l'application, Android utilise le certificat du développeur (fourni avec l'application) afin de vérifier l'intégrité du package de l'application. Si le contenu du package de l'application a été modifié, alors la signature devient invalide et l'installation de l'application est annulée.

Les signatures sont aussi utilisées afin de vérifier si deux applications ou plus appartiennent au même développeur – en fait, signées en utilisant le même certificat digital. Si tel est le cas, elles peuvent utiliser le même ID à travers la fonctionnalité *shareUserId*, utiliser le même ensemble de permissions et acquérir les signatures de type *Signature* et *SignatureOrSystem* accordées systématiquement au moment de l'installation. Plus de détails seront présentés dans la prochaine section.

3.2.2 Le modèle des permissions

Par défaut, Android ne permet l'accès qu'à des parties restreintes des ressources système, sauf pour les applications système. En effet, Android comporte un certain nombre d'API sensibles qui sont destinées à l'usage exclusif des applications système ainsi que les applications auxquelles l'utilisateur accorde sa confiance (AOSP, 2008f). Ces API renferment des fonctionnalités sensibles telles que les fonctions de téléphonie, SMS/MMS, GPS, appareil photo, etc., et sont protégées par le mécanisme des permissions.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.android.exampleApplication">
   <uses-sdk android:minSdkVersion="10" />

<application   android:label="Example application" >
       <activity android:name=".ControllerApp" >   </activity>

      <provider android:name="Fournisseur Adresse"
                  android:authorities="com.example.android.exampleApplicationProvider">
      </provider>

       <service android:name="ExampleService"
            android:exported = "false">
       </service>
   </application>
 <uses-permission android:name = "android.permission.ACCESS-FINE_LOCATION"
</manifest>
```

Figure 3.1. Déclaration de la permission d'accéder à l'emplacement GPS

3.2.2.1 Déclaration des permissions

Afin de pouvoir utiliser les fonctionnalités protégées du système, une application doit déclarer dans son fichier AndroidManifest.xml les permissions requises à cet effet (voir figure 3.1). Lors de l'installation de l'application, le système affiche une boîte de dialogue décrivant les permissions sollicitées et demande à l'utilisateur s'il souhaite continuer l'installation (voir figure 3.2).

Si l'utilisateur souhaite continuer, le système accorde toutes les permissions demandées à l'application. Par contre, Android n'offre pas la possibilité d'accorder ou de refuser des permissions individuelles, ce qui force l'utilisateur à accorder ou refuser toutes les permissions demandées en bloc. Une fois installée, l'application conserve ces permissions jusqu'à ce qu'elle soit désinstallée par l'utilisateur. De plus, il n'existe aucun moyen de révoquer ces permissions mise à part la désinstallation de l'application.

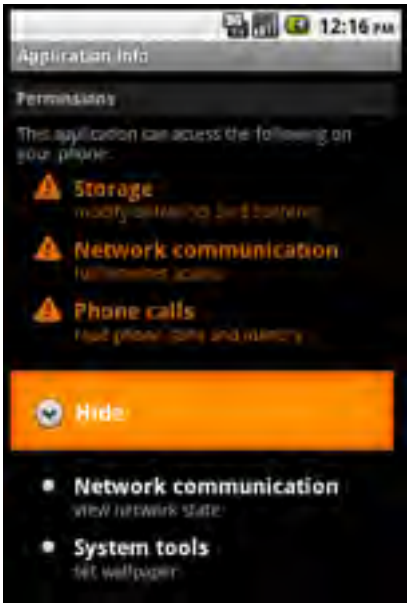

Figure 3.2 Demande de permissions lors de l'installation de l'application

Dans l'exemple d'application que nous avons présenté au chapitre 2, l'application *DomicileContactProximite* a besoin d'accéder aux données GPS du téléphone mobile. Par

conséquent, la permission *ACCESS_FINE_LOCATION* doit figurer dans le fichier de déclaration de l'application (voir figure 3.1).

3.2.2.2 Les permissions au niveau des composantes

Comme nous l'avons vu au chapitre 2, une application Android est un ensemble de composants de différents types pouvant interagir les uns avec les autres en utilisant le mécanisme d'intention. Les permissions sont également utilisées pour aider les développeurs à limiter l'accès aux composants de leurs applications en définissant leurs propres permissions dans le fichier Manifest. Les permissions définies par l'utilisateur servent à protéger les composants de l'application contre toute utilisation abusive. Les permissions sont définies en utilisant la balise *<permission>*, et doivent être mentionnées dans l'attribut *Android:permission* du composant devant être protégé. Une application désirant accéder à un composant protégé par une permission, doit explicitement demander la permission requise à cet effet dans son fichier Manifest

Chaque permission déclarée par le développeur possède un niveau de protection qui lui est associé (Shabtai et al., 2010) :

- *Normal* : Ce type de permissions est accordé de façon automatique sans demander l'approbation de l'utilisateur.
- *Dangerous* : Le système demande à l'utilisateur s'il souhaite accorder ce type de permission à l'application qui la demande.
- *Signature*: Ce type de permissions est accordé automatiquement par le système aux applications ayant été signées par la même clé privée et donc appartenant au même développeur.
- *SignatureOrSystem* : Ce type de permissions n'est accordé qu'aux applications signées par la même clé privée que l'image système (donc signées par Google) ou aux applications signées par la même clé privée qu'une application déjà installée.

Il appartient au développeur d'attribuer le niveau de permissions pour chaque permission qu'il déclare. Les permissions de type *Normal* impliquent un risque mineur et ne servent qu'à informer l'utilisateur que l'application qu'il est sur le point d'installer demande l'accès à une fonctionnalité donnée (p. ex., utiliser le vibreur). D'autre part, les permissions de type *Dangerous* sont utilisées pour protéger des composants offrant des fonctionnalités impliquant un risque plus important. Les permissions de type *Signature* sont utilisées pour garantir que seules les applications appartenant au même développeur peuvent interagir avec le composant protégé par la permission.

3.2.2.3 Usage des permissions au niveau des composants

Les permissions sont utilisées afin de protéger les composants du système ainsi que les composants appartenant aux applications. Les restrictions d'accès diffèrent selon le type de composant. Le tableau 3.1 montre comment les différents types de composants sont protégés, le moment de vérification et l'action prise par le système dans le cas où une application ne disposerait pas des permissions nécessaires.

3.2.2.4 Usage des permissions au niveau des intentions

Android offre la possibilité aux développeurs de spécifier quelle application peut recevoir et lire le contenu des intentions qu'elle envoie. Ce type de restriction s'applique aux intentions implicites décrites dans la section 2.4. En passant une permission comme paramètre à la méthode *sendBroadcast (intention, permission)*, seules les applications qui détiennent cette permission particulière sont autorisées à recevoir l'intention envoyée. Lorsqu'une application envoie une intention, le système vérifie d'abord si l'application appelée détient la permission de la recevoir. Par la suite, il vérifie si l'application appelante a le droit d'envoyer des intentions à l'application destinatrice.

Tableau 3.1 Vérification des permissions au niveau des composants

Composant Android	**Actions restreintes**	**Permissions vérifiées lors de**	**Action déclenchée par le manque de permissions**
Activité	Démarrer l'activité	*startActivity(), ActivityForResult()*	Une exception de sécurité est renvoyée à l'appelant
Service	Démarrer ou se lier à un service	*startService(), stopService(), bindService()*	Une exception de sécurité est renvoyée à l'appelant
Récepteur de diffusion	Envoi de diffusion au récepteur de diffusion	*sendBroadcast()*	L'intention n'est pas livrée au destinataire, sans exception de sécurité
Fournisseur de contenu	Accéder aux données détenues par le fournisseur de contenu	*ContentResolver.query(), ContentResolver.insert(), ContentResolver.update(), ContentResolver.delete()*	Une exception de sécurité est renvoyée à l'appelant

3.2.2.5 Vérification des permissions au niveau des processus

En plus des vérifications de permissions effectuées par le système lors des interactions entre les composantes des applications, Android offre la possibilité de vérifier les permissions des autres applications à n'importe quel moment durant l'exécution de l'application. La méthode *checkCallingPermissions()* peut être utilisée par une application afin de vérifier les permissions de l'application qui l'appelle. Également, la méthode *checkPermissions()* est invoquée lors qu'on désire vérifier les permissions accordées à une application donnée en utilisant le PID du processus dans lequel elle s'exécute. La dernière forme de vérification se base sur le nom d'un package (application) donnée en utilisant *PackageManager.checkPermission()*.

3.2.3 Les permissions sur les fournisseurs de contenu

L'utilisation des permissions pour les fournisseurs de contenu peut s'avérer insuffisante (AOSP, 2008f). Un développeur pourrait exprimer le besoin de définir son application comme étant la seule capable de mettre à jour le contenu de son fournisseur de contenu, et limiter l'accès des autres applications à la lecture seulement (Enck, Ongtang et McDaniel, 2009). Afin de combler ce besoin, Android permet de séparer les droits de lecture et d'écriture pour chaque fournisseur de contenu ce qui augmente le degré de contrôle sur les données de l'application.

3.2.4 Les permissions par URI

Android permet une forme de délégation momentanée de permissions qui, lorsqu'utilisée, permet à une application d'accéder à un URI appartenant à une autre application sans avoir demandé la ou les permissions au préalable (Enck, Ongtang et McDaniel, 2009). En effet, une application peut envoyer une intention qui contient l'URI d'une donnée quelconque, avec le flag *Intent.FLAG_GRANT_READ_URI_PERMISSION*. L'application destinatrice pourrait alors lire le contenu de l'URI passé dans l'intention, peu importe si elle dispose ou non des permissions pour accéder au fournisseur de contenu de l'application appelante.

Par exemple, une application de messagerie électronique peut utiliser le mécanisme décrit ci-dessus afin de permettre à une deuxième application de consulter le contenu d'un courriel en particulier sans pour autant lui permettre d'accéder au contenu de tous les courriels.

3.2.5 Composant publique et composant privé

Les applications Android comportent souvent des composants qui ne doivent jamais être accessibles par le reste des applications (Enck, Ongtang et McDaniel, 2009). En effet, les composants peuvent manipuler des informations sensibles et retourner des résultats (tels que

les mots de passe et les numéros de carte de crédit) à d'autres composants. Android permet de mettre ce type de composant à l'abri des autres applications, en définissant l'attribut *exported* dans la déclaration du composant dans le fichier AndroidManifest.xml. Si *exported* est *true*, le composant est accessible à toutes les applications. Si la valeur est *false*, alors le composant ne peut être accessible qu'aux composants de la même application ou des applications ayant le même ID.

Par ailleurs, la valeur par défaut de l'attribut *exported* est déterminée par l'existence ou l'absence des filtres d'intention dans la déclaration du composant. Si aucun filtre d'intention n'est attaché au composant, celui-ci est considéré comme privé (AOSP, 2008c). Cependant, il est toujours possible de l'activer s'il est désigné par son nom tel qu'il apparait dans le fichier Manifest. D'autre part, l'existence des filtres d'intention implique que le composant est public et accessible aux autres applications. L'accès est alors régi par le mécanisme de permissions.

CHAPITRE 4

ATTAQUES ET VULNÉRABILITÉS CONTRE LE SYSTÈME ANDROID

Les applications malveillantes Android exploitent de nombreux types de vulnérabilités. La forme la plus simple des applications malveillantes ciblant Android tire parti de l'inadvertance des utilisateurs des appareils Android, pour atteindre leurs objectifs. Un type plus sophistiqué d'applications malveillantes exploite des vulnérabilités de la plateforme mobile. Ces vulnérabilités peuvent être des vulnérabilités de la plateforme Android ou des vulnérabilités spécifiques au noyau Linux sur lequel est construit Android. Dans ce chapitre, nous proposons une nouvelle classification qui partage les différents types des applications Android malveillantes en se basant sur la nature des vulnérabilités exploitées. Nous illustrons cette classification en présentant des exemples d'applications malveillantes pour chaque catégorie.

Cette nouvelle classification est présentée schématiquement dans la figure 4.1.

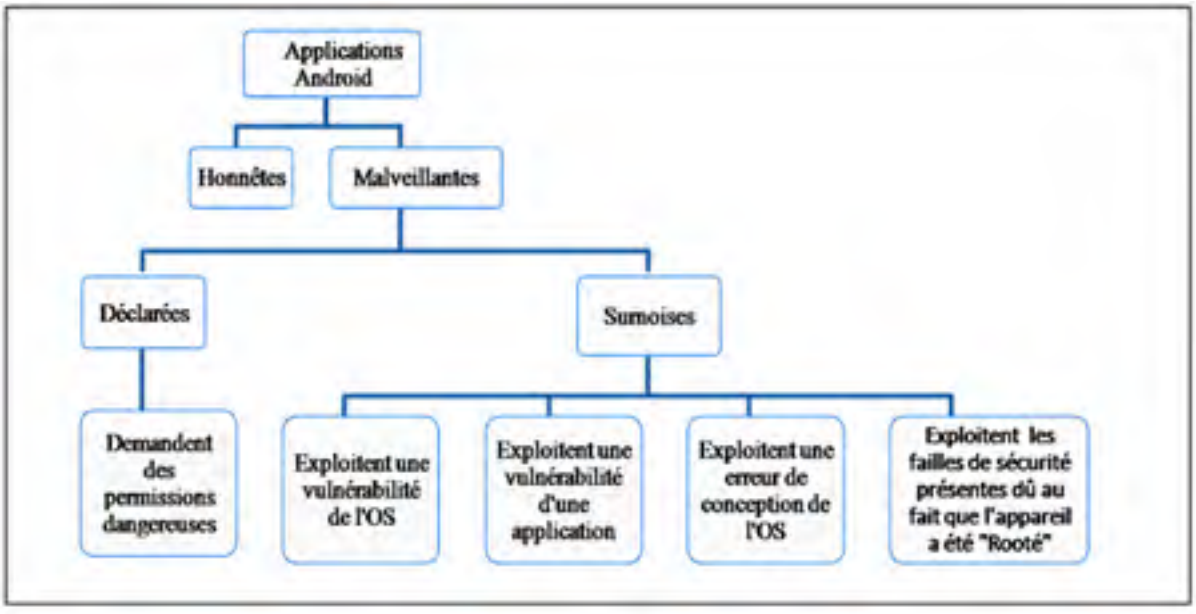

Figure 4.1 Arbre de classification des applications Android

4.1 Les applications malveillantes déclarées

Comme nous l'avons vu précédemment, la sécurité d'Android est essentiellement basée sur le modèle de permissions, qui compte sur l'utilisateur pour donner les permissions nécessaires à l'application qui les demande. Cependant, l'efficacité du système de permissions est fortement dépendante des connaissances de l'utilisateur, en plus de sa compréhension du modèle de permissions. Alors qu'une minorité d'utilisateurs d'Android peut avoir une très bonne connaissance liée aux systèmes d'exploitation, la grande majorité de cette communauté n'a pas les connaissances nécessaires liées à la plate-forme mobile, et encore moins sur la sécurité d'Android. Par ailleurs, la plupart des utilisateurs sont insensibles aux risques et aux impacts que la décision d'octroi de permissions peut avoir sur la confidentialité de leurs données personnelles ainsi que sur la fiabilité de leur téléphone mobile Android.

Les applications malveillantes déclarées tirent avantage de l'inadvertance de la plupart des utilisateurs d'Android quand il s'agit de l'octroi de permissions aux applications. Nous classons ce type d'applications malveillantes comme déclarées puisqu'elles demandent des permissions qui sont inutiles pour leur fonctionnement et donc elle déclare implicitement ses intentions malveillantes.

Manifestement, la majorité des utilisateurs prêtent rarement attention à la liste des permissions requises par une application, soit parce qu'ils ne comprennent pas l'intérêt de le faire ou soit parce que la tentation de tester l'application dépasse leurs soucis de sécurité. Dans les deux cas, l'application malveillante finit par obtenir les permissions nécessaires pour effectuer ses tâches malveillantes, sans avoir à chercher des vulnérabilités supplémentaires liées au système.

L'application malveillante classée sous cette catégorie se présente comme une application légitime, qui prend souvent la forme d'un simple jeu. Au moment de l'installation, elle requiert une ou plusieurs permissions dangereuses. Des exemples de permissions dangereuses seraient :

- effectuer des appels téléphoniques
- envoyer des SMS
- obtenir la localisation GPS
- obtenir la liste des contacts de l'utilisateur

Toutefois, rien dans la description de l'application n'indique que l'application aurait besoin de ces privilèges pour accomplir les tâches qu'elle prétend réaliser. Ces applications sont plus susceptibles d'être des applications malveillantes que des applications légitimes. Une application malveillante de ce genre contient un code caché qui effectue des tâches en silence à l'insu des utilisateurs, et en l'absence de moyen d'avertissement (p. ex., notifications d'envoi de message), l'utilisateur ne pourrait éviter l'impact d'une telle attaque à un stade précoce.

L'exemple le plus illustratif appartenant à cette catégorie a été l'application *TapSnake* qui n'était rien d'autre qu'un espiogiciel (Perez, 2010). Cette application se présentait comme un jeu de serpent simple pour Android. La simplicité et la popularité du jeu ont fait en sorte que l'application fut téléchargée par un grand nombre d'utilisateurs à partir d'*AndroidMarket* et les marchés d'applications parallèles. Lors de son installation, *TapSnake* demandait les permissions pour accéder à la localisation GPS du téléphone et à Internet sans aucune contrainte. Malgré le danger potentiel de cette combinaison de permissions et son inadéquation avec la description de l'application, la plupart des utilisateurs ont ignoré ce fait en raison de la forte motivation d'avoir le jeu.

La version gratuite du logiciel malveillant était conçue pour envoyer les données GPS de l'utilisateur à un serveur externe toutes les 15 minutes, tandis que la version payante, appelée GPSSpy, permettait à son utilisateur de retracer l'emplacement d'un utilisateur spécifique. Supposons que nous voulons suivre quelqu'un. Nous n'avons qu'à inciter cette personne à installer l'application sur son téléphone. À la première utilisation, l'utilisateur sera invité à saisir son adresse de courriel. En utilisant la version payante, nous utilisons la même adresse

courriel avec l'application GPSSpy payante pour suivre l'historique des emplacements géographiques visités par l'utilisateur espionné.

Suite à la découverte de l'application malveillante, Google l'a retiré d'*AndroidMarket*, et a procédé à sa désinstallation à distance des appareils infectés. Cependant, quelques versions de l'application malveillante continuent à exister sur les marchés d'applications parallèles et sont disponibles pour le téléchargement.

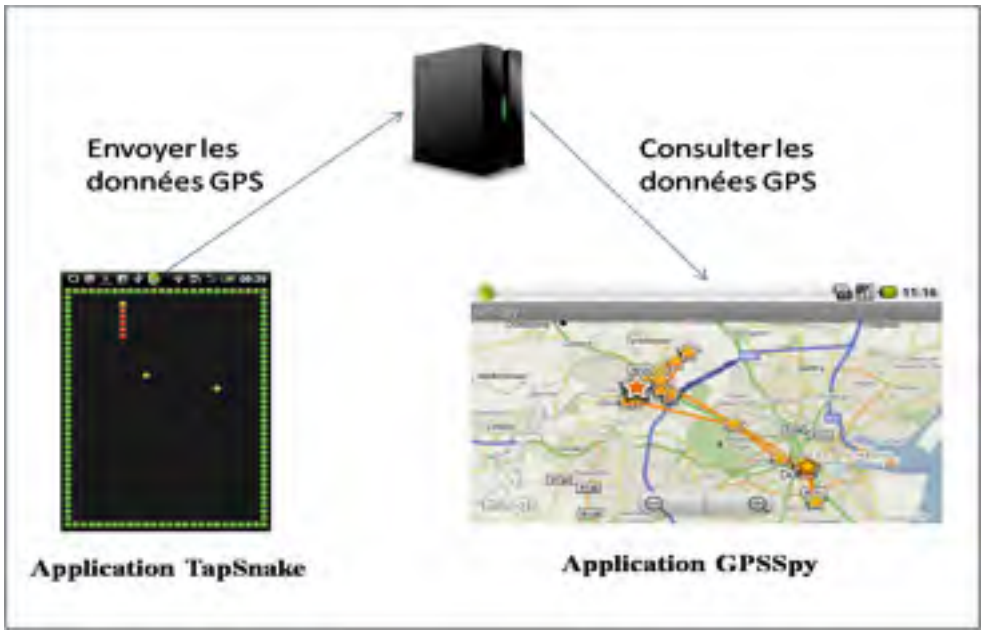

Figure 4.2 Le fonctionnement des applications TapSnake et GPSSpy

4.2 Les applications sournoises

La principale différence entre les applications malveillantes déclarées et sournoises, c'est que ces dernières ne donnent pas d'indice sur leurs intentions malveillantes, et ne demandent pas forcément de permissions spéciales pouvant paraître suspectes. Ce type d'applications malveillantes est plus sophistiqué, et peut exploiter des vulnérabilités liées soit au noyau Linux ou soit à la plateforme Android. Il peut également exploiter les vulnérabilités des applications déjà installées. Dans ce qui suit, nous exposons les différents types d'applications malveillantes sournoises sur la base des failles qu'elles exploitent.

4.2.1 Applications sournoises exploitant les vulnérabilités du noyau Linux

Android est un système multicouche qui repose essentiellement sur un noyau Linux. Comme tout autre système d'exploitation, Linux a souffert de plusieurs vulnérabilités. Ainsi, les téléphones mobiles Android sont potentiellement vulnérables à des attaques ciblant le noyau Linux.

Les applications malveillantes conçues pour exploiter les vulnérabilités du noyau ont un objectif commun, qui consiste essentiellement à effectuer une attaque d'élévation de privilèges et obtenir les privilèges de l'administrateur root sur téléphone mobile. Le but principal est de contourner le système de bac de sable d'Android dans l'optique d'effectuer des tâches non autorisées sur le système. De plus, en obtenant les privilèges de l'administrateur root, l'application malveillante gagne un accès à certaines parties de l'API censées être protégées par le système de permissions.

Suite à l'obtention les privilèges de l'administrateur root une application malveillante peu :

- lire ou modifier le système de fichiers dans son intégralité;
- lire ou modifier des fichiers d'autres applications pouvant contenir des informations sensibles de l'utilisateur;
- lire la liste des contacts, ainsi que le contenu des SMS et MMS;
- effectuer des appels téléphoniques, envoyer des SMS, etc.;
- télécharger et installer d'autres applications sans le consentement de l'utilisateur.

Les vulnérabilités du noyau Linux les plus exploitées sur Android sont *RageAgainstTheCage* et *Exploid* (US-cert/NIST, 2009). Ces deux vulnérabilités particulières permettent à l'attaquant d'obtenir un accès privilégié de l'administrateur root sur Android.

DroidDream (Strazzere, 2011) serait la première application malveillante exploitant une telle vulnérabilité afin d'obtenir un accès d'administrateur root, et gagner un contrôle illimité sur

les appareils infectés. *DroidDream* s'appuie sur les deux vulnérabilités mentionnés précédemment comme moyen pour contourner le système de bac de sable implémenté au niveau du noyau Linux. Les deux vulnérabilités ciblées ont été corrigées dans la version Android 2.3. Cependant, les statistiques présentées par Google (Smith, 2011) montrent que 60% des appareils Android n'ont pas encore installé la dernière version d'Android et restent donc vulnérables face à *DroidDream*.

Une caractéristique intéressante de *DroidDream* est sa capacité de télécharger et d'installer d'autres logiciels sur les téléphones mobiles. Suite à son obtention d'un accès privilégié de l'administrateur root, *DroidDream* effectue une recherche d'un package spécifique nommé *com.android.providers.downloadsmanager*. Si le package n'est pas trouvé, *DroidDream* installe discrètement une deuxième application malicieuse appelée *ProviderDownloadManager.apk*, qui agit à la réception de commandes en provenance d'un serveur de contrôle externe. Ainsi, ce logiciel malveillant simule le fonctionnement d'un réseau de bots (Feily, Shahrestani et Ramadass, 2009). *DroidDream* pourrait être considéré comme un agent zombie puissant qui peut installer des applications silencieusement et exécuter du code avec les privilèges de l'administrateur root à volonté (Strazzere, 2011).

4.2.2 Applications sournoises exploitant les vulnérabilités de la plateforme Android

Ce type d'applications malveillantes exploite les vulnérabilités de la plateforme Android située au-dessus du noyau Linux. Les vulnérabilités sont principalement dues à des erreurs conceptuelles dans la plateforme au niveau de l'architecture de sécurité. Contrairement aux vulnérabilités du noyau, les erreurs conceptuelles sont difficiles à corriger et peuvent exiger un changement majeur à la plateforme.

Le site web d'*AndroidMarket* inclut une fonctionnalité qui permet aux utilisateurs des ordinateurs conventionnelles d'installer automatiquement des applications sur leurs appareils à partir d'*AndroidMarket* simplement en cliquant sur un lien (Oberheide, 2010). En procédant ainsi, l'OS Android ne demande pas aux utilisateurs de confirmer l'installation de

l'application et lui accorder ou non les permissions qu'elle demande, ce qui rend le scénario d'attaque plus simple. En exploitant cette vulnérabilité, un attaquant peut déclencher en silence l'installation d'applications malveillantes en incitant une victime à cliquer sur un lien tout en étant connectée à son compte Google sur leur ordinateur de bureau ou sur leur téléphone. L'application malveillante transmise au téléphone de la victime peut utiliser toutes les permissions demandées dans son fichier AndroidManifest.xml comme si elles étaient explicitement accordées par l'utilisateur.

4.2.3 Application malveillante exploitant le Rooting de l'appareil

À l'intérieur de Linux, l'administrateur root est l'utilisateur ayant le plus de privilèges du système. Une fois qu'il a obtenu les privilèges de l'administrateur root dans l'environnement Linux, un utilisateur a la possibilité d'effectuer toutes les actions désirées sur le système sans être confronté à aucune restriction. Pour l'environnement Android, le «Rooting » du téléphone mobile est équivalent à avoir les privilèges root sur Linux. Suivant cette analogie, un utilisateur « root » sur Android à la possibilité d'effectuer toutes les actions désirées, et de contourner les restrictions de sécurité imposées par Google et les fabricants des appareils. Actuellement, les utilisateurs légitimes et les amateurs d'Android utilisent la technique présentée dans la section 4.2.1 afin d'obtenir le contrôle total de leurs téléphones mobiles et d'acquérir la capacité d'augmenter les fonctionnalités qu'ils obtiennent via un accès privilégié d'administrateur root.

Les utilisateurs qui passent outre le système de protection de leurs téléphones mobiles sont motivés par au moins l'une des limites suivantes imposées par Google, les fabricants des téléphones mobiles ou les fournisseurs de téléphonie mobile (Felt et al., 2011):

- Les utilisateurs ne peuvent pas effectuer des sauvegardes complètes du système.
- L'impossibilité d'utiliser l'appareil comme un point d'accès wifi, afin de partager la connexion internet mobile avec d'autres appareils mobiles ou avec des ordinateurs conventionnelles.

- L’impossibilité de désinstaller les applications installées par les fournisseurs de téléphonie mobile.
- L’impossibilité d’installer des versions personnalisées du système d'exploitation qui contiennent des fonctionnalités supplémentaires ou des améliorations.
- L’impossibilité de migrer les applications vers la carte de stockage amovible SD afin d’économiser l'espace disque.

Quoique le rooting d’Android permette de contourner ces restrictions, cette pratique n’est pas sans conséquence sur la sécurité d’Android. La liste suivante contient un exemple des actions qu’une application malveillante peut effectuer sur un appareil « rooté » :

- Remplacer le clavier Android avec une version contenant un enregistreur de frappes;
- Lire/modifier/supprimer les fichiers des autres applications ou du système;
- Télécharger et installer des applications à l’insu de l’utilisateur;
- Accéder au compte *GMail* de la victime et effectuer des achats d’applications à son insu;
- Effectuer des appels téléphoniques discrètement sans demander l’accord de l’utilisateur ni déclarer les permissions nécessaires.

Les attaquants peuvent développer des applications malveillantes qui ciblent les appareils rootés en particulier, à l’opposé de *DroidDream* qui lui, « roote » le téléphone par lui-même. Dans le cas des téléphones Android « rooté », les développeurs malicieux ne possèdent pas nécessairement de connaissances approfondies du noyau Linux. Les applications malveillantes n’ont qu’à demander un accès privilégié d’administrateur root sur le système à l’aide de la commande shell *su*. Puisque l’utilisateur à passer outre le système de protection de son téléphone Android, l’accès root est attribué à toute application qui le demande. Toute application installée sur le téléphone pourrait utiliser la commande shell *su* de Linux et bénéficier des privilèges de l’administrateur root lui permettant d’effectuer toute sorte d’opérations sur le système sans la moindre restriction.

4.2.4 Applications sournoises exploitant les vulnérabilités des applications installées

Comme tout logiciel, une application Android peut être assujettie à des erreurs conceptuelles et de développement, résultant en brèches de sécurité. L'API d'Android offre plusieurs moyens aux développeurs pour protéger leurs applications. Cependant, de nombreux développeurs développent des applications vulnérables par inadvertance ou en raison de la mauvaise compréhension de l'architecture de sécurité Android.

4.2.4.1 Vulnérabilités au niveau des composants

Une attaque d'élévation de privilèges s'effectue lorsqu'une application non privilégiée réussit à interagir avec une application ayant plus de privilèges. Considérons une application A ayant la permission d'effectuer des appels téléphoniques. Cette application comporte un composant (p. ex., un service) capable de passer des appels téléphoniques. Cependant, ce service n'est pas protégé par une permission qui restreint l'accès à celui-ci (voir section 3.2.2.2). Par conséquent, toute application non priviligé pourrait utiliser le service en question pour passer des appels téléphonique sans avoir la permission explicite de l'utilisateur (Davi et al., 2011).

Un développeur malicieux ayant connaissance de la vulnérabilité de l'application A, développe une application B qui vise à exploiter cette faille de sécurité. L'application B a pour but de passer des appels téléphoniques vers des numéros surtaxés sans que l'utilisateur du téléphone mobile ne se rende compte.

Ce modèle d'attaque a été détaillé dans (Davi et al., 2011). Au moment de son installation, l'application B ne demande pas la permission d'effectuer des appels. Une fois installée, elle accède au composant non protégé de l'application A et lui demande via une intention de passer un appel téléphonique vers un numéro de téléphone surtaxé.

4.2.4.2 Vulnérabilités au niveau des données de l'application

L'application *Skype* (Case, 2011) est un exemple illustrant bien le cas des applications vulnérables. Dans sa version vulnérable, le répertoire de données *Skype* contenait un dossier ayant comme nom le pseudonyme *Skype* du propriétaire du téléphone (p. ex., */data/data/com.Skype.amine_hadhiri/files/jcaseap*), comportant la liste de contacts *Skype* de l'utilisateur, son profil ainsi que l'historique de messageries instantanées stocké sous forme de fichiers de bases de données. Par erreur, les développeurs de *Skype* ont attribué à ces fichiers des permissions d'accès incorrects, permettant à n'importe quelle application de les lire. Non seulement ils étaient accessibles, mais totalement en clair. La vulnérabilité a été rapportée par Androidpolice (Case, 2011) avant qu'elle soit exploitée par une application malveillante.

4.2.4.3 Vulnérabilités au niveau des intentions

Les intentions implicites, pouvant contenir des informations sensibles, peuvent être interceptées par des applications malveillantes si elles ne sont pas protégées adéquatement avec des permissions. Une application malveillante peut déclarer des filtres d'intention avec toutes les actions et les catégories possibles, tout en s'associant à une priorité élevée. Ceci offrirait à l'application un accès au contenu sensible de l'intention. L'interception des intentions peut mener à des scénarios différents incluant les activités, les récepteurs de diffusion et les services. Ces scénarios sont présentés dans ce qui suit (Chin et al., 2011):

a) Vol de diffusions

Les intentions qui ne sont pas correctement protégées sont potentiellement vulnérables aux écoutes passives et aux dénis de service. Une application malicieuse peut écouter silencieusement les intentions publiques de toutes les applications en déclarant un filtre d'intention qui répertorie toutes les actions possibles. On désigne par intention publique, toute intention implicite qui n'est pas protégée par une permission. Les intentions

persistantes (envoyées utilisant la méthode *sendStickyBraodcast()*) sont particulièrement vulnérables aux écoutes. Contrairement aux intentions standards, ce type d'intentions n'est pas détruit suite à sa diffusion, et est diffusé à chaque nouveau récepteur dès son installation, ce qui offre plus de temps aux attaquants pour les lire.

De plus, les intentions ordonnées (envoyées utilisant la méthode *sendOrderedBroadcast ()*) peuvent être l'objet d'une attaque de dénis de service ou d'injection de données. Ce type d'intentions est livré aux applications en suivant un ordre déterminé qui se base sur les priorités accordées à chaque récepteur. Une application ayant la priorité la plus élevée reçoit l'intention en premier, et ainsi de suite. Chacune de ces applications peut arrêter la diffusion d'une intention. Une application malveillante pourrait déclarer un récepteur de diffusion avec une priorité élevée, et empêcher toute propagation supplémentaire par la suite.

Les intentions ordonnées peuvent aussi être victime d'injection de données malicieuses. Lors du traitement de l'intention, un récepteur peut passer un résultat pour celui qui le succède. À la fin, le résultat est renvoyé à la composante qui a initié la diffusion. Un récepteur malveillant pourrait modifier ces résultats, causant un comportement inattendu de la part des applications.

Par ailleurs, les intentions non ordonnées ne peuvent être victimes d'une attaque de dénis de service, puisqu'elles sont livrées simultanément à toutes les applications, mais restent tout de même vulnérables à l'injection malicieuse de données.

b) Détournement d'activité

Une attaque de détournement d'activité consiste à lancer une activité malicieuse au lieu de l'activité envisagée par l'intention. L'application malveillante peut accéder aux données sensibles contenues dans l'intention, puis relayer l'intention vers l'activité légitime.

Dans un scénario plus sophistiqué, l'application malicieuse peut usurper l'activité légitime, afin de forcer l'utilisateur à saisir des données sensibles telles que mot de passe et ses numéros de carte de crédit.

c) Détournement de service

Le détournement de service se produit quand un service malveillant intercepte une intention destinée à un service légitime. L'application victime de cette attaque établit alors une connexion avec un service malveillant au lieu de celui désiré. Le service malveillant peut voler des données et retourner des informations fausses au sujet de l'accomplissement des actions demandées. Contrairement au détournement d'activité, le détournement de service est plus difficile à détecter par l'utilisateur, car aucune interface utilisateur n'est impliquée. Lorsque plusieurs services peuvent gérer une intention, Android choisit un service au hasard sans que l'utilisateur soit invité à sélectionner un service.

Comme avec le détournement d'activité, le détournement de service permet à l'attaquant d'usurper les réponses aux requêtes provenant des autres applications. Une fois que le service malveillant est lié à l'application appelante, l'attaquant peut retourner des données arbitraires malveillantes ou simplement retourner un résultat, sans avoir exécuté les tâches demandées.

d) Risques liés aux intentions portant des privilèges spéciaux

Comme vu dans la section 3.2.4, Android offre la possibilité de définir les permissions par URI. Si un composant malicieux intercepte l'intention contenant un URI (utilisant les manières mentionnées précédemment), il peut accéder aux données contenues dans celui-ci.

De même, les intentions de type *pending intents* délèguent des privilèges. Ce type d'intentions délègue les permissions de l'application appelante à l'application appelée.

L'intention conserve la totalité des privilèges de l'application qui l'a créé. Le bénéficiaire d'un *pending intent* peut envoyer l'intention d'une application tierce, et l'intention garde encore les permissions de son créateur. Si une application malveillante obtient une intention de type *pending intent*, les permissions de l'application qui a créé l'intention peuvent être abusée.

CHAPITRE 5

OBJECTIFS ET MOTIVATIONS DES ATTAQUES CONTRE ANDROID

Dans la section précédente, nous avons classé les applications malveillantes Android selon les vecteurs d'attaque qu'elles exploitent. À travers ce chapitre, nous présentons une classification des applications Android malveillantes en nous basant sur les motivations et objectifs de ceux qui l'ont développée. Cette classification est tirée de (Felt et al., 2011).

5.1 Nouveauté et divertissement

Ce type d'applications malveillantes est le moins nocif parmi les applications malveillantes ciblant Android. Ce type d'applications malveillantes est développé dans l'unique but de divertir son auteur et lui apporter une satisfaction personnelle, sans chercher à causer des dommages à l'utilisateur ou à son téléphone mobile.

Par exemple, *Smspacem* (Hayashi, 2011) envoyait des messages texte dénigrant la religion à partir des téléphones Android. Cependant, ce type d'application malveillante est de moins en moins observé pour céder la place à des applications malveillantes à but lucratif.

5.2 Vol et vente des informations de l'utilisateur

Un nombre croissant d'applications Android présente une menace pour la confidentialité des informations personnelles des utilisateurs. En fait, l'API Android permet aux applications l'accès à un certain nombre d'informations sensibles de l'utilisateur. La liste de contacts téléphoniques, l'historique de navigation internet et de téléchargements, la liste des applications installées, l'emplacement de l'utilisateur ainsi que l'IMEI (numéro d'identification unique du téléphone mobile) sont des exemples des informations accessibles sur demande à partir des applications installées.

Le numéro IMEI est en soi une information précieuse pour la vente de téléphones sur le marché noir. Les téléphones mobiles volés sont souvent vendus à des prix très attractifs. Cependant, un appareil volé peut être désactivé à l'aide de son numéro IMEI si son propriétaire avertit les autorités (Felt et al., 2011). Ainsi, en changeant l'IMEI du dispositif volé avec un IMEI valide rend le téléphone exploitable, d'où l'importance accrue des IMEI.

L'emplacement de l'utilisateur est aussi une information précieuse pour les attaquants. Depuis quelques années, les annonceurs et les sociétés de marketing s'intéressent davantage à la géo localisation. Ces informations sont collectées par les applications malveillantes et probablement vendues aux annonceurs. La liste des applications installées et l'historique du navigateur permettent de mieux cibler les clients potentiels et d'optimiser le profil comportemental de chaque utilisateur et client potentiel (Felt et al., 2011).

5.3 Abus des services payants

Un nombre élevé d'applications malveillantes pour Android cherche à abuser des services payants (téléphonie et SMS) afin de réaliser des gains financiers. Ces applications malveillantes sont conçues de telle sorte à pouvoir effectuer des appels téléphoniques et d'envoyer des SMS à des numéros de téléphone surtaxés à l'insu de l'utilisateur. En outre, ces appels peuvent passer inaperçus pour l'utilisateur jusqu'à ce qu'il reçoive sa facture.

Dans le présent scénario, l'attaquant commence par l'acquisition d'un numéro de téléphone dédié surtaxé. Lors de l'appel ou de l'envoi de SMS à ce numéro, l'opération est facturée à un prix supérieur au coût normal d'un SMS ou d'un appel téléphonique. Les recettes sont ensuite partagées entre l'attaquant et les fournisseurs de téléphonie mobile.

Android.fakePlayer (FortiGuard, 2010) est un exemple d'application malveillante développée pour abuser des services téléphoniques payants. Cette application malveillante est configurée pour envoyer de multiples SMS à deux numéros surtaxés. Le premier SMS envoyé est chargé

à 3.50 $ alors que le deuxième coûte 6 $, totalisant une charge totale de 9.50 $ sur la facture de l'utilisateur, et ce, à chaque fois que l'application est exécutée.

5.4 Pourriel via SMS

Le pourriel par SMS est utilisé pour la diffusion de publicité commerciale et de liens d'hameçonnage. Les polluposteurs commerciaux sont de plus en plus tentés d'utiliser les logiciels malveillants afin d'envoyer indirectement des pourriels par SMS, car l'envoi de pourriels par SMS est illégal dans la plupart des pays. Par conséquent, l'envoi des messages à partir d'un téléphone compromis réduit le risque pour le polluposteur puisque cette technique obscurcit la provenance du pourriel. *AdSMS* (ThreatSolutions, 2011) est un exemple de logiciels de pourriel SMS et se présentait à l'utilisateur comme étant une mise à jour du système.

5.5 Vol des informations d'authentification de l'utilisateur

Android stocke les informations d'identification de l'utilisateur directement sur le téléphone mobile. Ces informations d'identification sont une information précieuse pour les attaquants. Ils peuvent inclure des informations des comptes Google, ses mots de passe de réseaux sociaux et d'autres comptes de messagerie en ligne, sans oublier les numéros de carte de crédit et des informations bancaires. Les informations d'identification et authentification de l'utilisateur peuvent être obtenues par des attaquants via les applications malveillantes, puis utilisées de sorte à générer des profits financiers. Les enregistreurs de frappes sont un moyen simple pour voler ces informations de l'utilisateur. Des méthodes d'hameçonnage et d'ingénierie sociale sont aussi envisageables.

Une application malveillante récemment découverte nommée *Netflic* (SonicWall, 2011) agissait dans ce sens. *Neflic* se faisant passer pour l'application *Netflix* (site payant pour regarder des films et série en ligne) dans le but de collecter les informations du compte *Netflix* de l'utilisateur et les envoyer ensuite à un serveur distant. L'auteur de l'application malveillante, dans ce cas, a profité de la popularité de l'application mobile *Netflix* et du fait

qu'elle ne soit pas disponible sur toutes les versions d'Android. L'application malveillante ressemble beaucoup à l'application officielle *Netflix* Android et vole des informations du compte utilisateur avant de s'auto désinstaller du système.

Par ailleurs, un nouveau type d'applications malveillantes sont en train d'émerger. Ces applications visent le domaine bancaire mobile en particulier. Lors d'une transaction ou l'accès à un compte bancaire en ligne, certaines banques exigent des informations d'identification supplémentaires envoyées pour empêcher des attaques conduites par des personnes malveillantes s'insérant entre deux entités légitimes (Felt et al., 2011). En effet, la banque envoie par SMS un nombre aléatoire, appelé numéro d'authentification de transaction mobile (*mTAN*), à un numéro de téléphone mobile précédemment enregistré. Ce type d'applications malveillantes est conçu pour intercepter les messages SMS reçus dans le but de récupérer le *mTan* et ainsi contourner l'authentification à double facteur employée dans le domaine bancaire mobile.

5.6 Extorsion

La plupart des téléphones mobiles Android détiennent des informations sensibles. Si ces informations tombent entre de mauvaises mains, elles peuvent menacer la vie privée des utilisateurs de ces appareils. L'historique des appels téléphoniques, l'historique des sites Web visités et l'historique de la messagerie instantanée peuvent être volés et utilisés afin d'exercer du chantage sur les utilisateurs victimes. Un auteur d'application malveillante peut menacer l'utilisateur de rendre ces informations accessibles au public sur le web, si une rançon n'est pas versée.

Une application malveillante nommée *Golddream* (Venkatesan, 2011) avait la capacité d'enregistrer la conversation téléphonique. Bien qu'il existe déjà des logiciels malveillants pour Android qui ont la capacité de stocker des informations liées à l'historique des appels, cette application malveillante en particulier, stocke les enregistrements audio de l'appel téléphonique sous format de fichier *. *amr* sur la carte de stockage amovible SD du téléphone

mobile. De plus, il stocke un fichier de configuration dans la mémoire du téléphone, avec les détails d'un serveur externe. La présence d'un tel fichier suggère que les conversations enregistrées peuvent éventuellement être envoyées à un serveur distant, présentant une source d'intimidation à l'utilisateur si le contenu des conversations téléphoniques contient des informations sensibles.

Le code de l'application malveillante est caché dans ce qui semble être une application légitime et présente un écran d'installation semblable à celui des applications légitimes. Une fois installée, la charge utile est activée et l'enregistrement des appels peut débuter. À ce jour, on ne sait pas si l'auteur a tenté de contacter ses victimes pour réclamer une rançon, cependant la nature de l'attaque laisse croire que l'application malveillante a été conçue à des fins de chantage.

5.7 Empoisonnement des moteurs de recherche

Souvent, les moteurs de recherche recommandent des sites web ou modifient le classement des moteurs de recherche d'un site en surveillant les taux de visite des utilisateurs. Le classement d'un site web donné s'améliore lorsque le nombre des visiteurs augmente.

Les applications malveillantes peuvent initier des requêtes multiples pour un site en particulier, faisant en sorte que le taux de cliques surveillé par les moteurs de recherche soit faussé artificiellement. En augmentant le rang de recherche d'un site appartenant à un attaquant, le nombre des visites de clients potentiels augmente ce qui génère des revenus à partir des publicités figurant sur le site.

Android.Adrd (Symantec, 2011) est un exemple d'application malveillante qui empoisonnait les résultats du moteur de recherche chinois mobile nommé *Baidu*, en générant des visites artificielles pour un site de nouvelles conçu pour les mobiles, augmentant potentiellement son rang dans les résultats de recherche *Baidu*.

5.8 Publiciel

Un grand nombre de réseaux publicitaires paie les développeurs des applications mobiles pour chaque affichage et clique quand ils affichent leurs publicités, en moyenne autour de 2 $ par millier d'affichages. Le modèle d'attaque dans ce cas consiste à cloner des jeux populaires légitimes et inclure des bibliothèques de publicité mobiles enregistrées au nom de l'attaquant.

Chaque fois que l'application est utilisée, les annonces sont affichées. L'attaquant génère ainsi des revenus publicitaires. L'application modifiée fonctionne exactement comme l'application originale, et l'utilisateur ne se rend pas compte qu'ils utilisent une version illégitime.

5.9 Les motivations futures

Alors que les objectifs cités dans la section précédente ont été déjà observés dans les logiciels malveillants récents, d'autres possibilités existent. Dans ce qui suit, nous présentons quelques objectifs qui peuvent être observés dans les applications malveillantes Android dans l'avenir.

5.9.1 La communication en champ proche (NFC)

La communication en champs proche est de plus en plus incorporée dans les téléphones mobiles, permettant de simplifier les transactions entre les appareils compatibles NFC. Étant donné que de plus en plus de personnes utilisent les téléphones intelligents dans des tâches sensibles telles que les opérations financières, les logiciels malveillants mobiles peuvent s'avérer particulièrement préjudiciables. Un nombre croissant d'utilisateurs va utiliser les interfaces NFC de leur téléphone pour effectuer des transactions et pour le magasinage dans les magasins compatibles NFC.

Dans (Felt et al., 2011), les auteurs prédisent que l'interface NFC deviendra une cible populaire pour les logiciels malveillants en raison de la facilité avec laquelle les transactions financières peuvent se produire en utilisant NFC. Cette prédiction est également appuyée par (Robert, 2011), qui rapporte les vulnérabilités de l'implémentation actuelle de NFC dans Android. Une application malveillante qui est capable d'utiliser NFC a la capacité potentielle d'interagir avec les matériels validés NFC placés à proximité du téléphone (p. ex., dans une poche), tels que les cartes de crédit.

5.9.2 DDoS

Une attaque de dénis de service distribuée (DDos) se produit lorsqu'un attaquant ordonne à un grand nombre d'ordinateurs compromis d'envoyer simultanément un grand nombre de requêtes à un serveur particulier. Un moyen pour limiter l'impact de l'attaque est de bloquer les adresses IP des ordinateurs des visiteurs se comportant d'une manière suspicieuse. Cependant, les fournisseurs de téléphonies mobiles changent souvent l'adresse IP attribuée aux téléphones mobiles (Felt et al., 2011) , ce qui rend le blocage des adresses IP peu efficace pour contrer l'attaque. Ceci peut encourager les attaquants à utiliser les téléphones mobiles pour monter des attaques DDoS. Toutefois, les attaquants doivent prendre en considération les ressources limitées en batterie et en bande passante des téléphones mobiles, en réduisant le nombre de requêtes envoyées à quelques requêtes HTTP par minute (Felt et al., 2011).

CHAPITRE 6

REVUE DE LITTÉRATURE DES SOLUTIONS DE SÉCURITÉ PROPOSÉES POUR ANDROID

La sécurité mobile et en particulier la sécurité d'Android continue à captiver l'intérêt d'un nombre croissant de chercheurs dans le domaine de la sécurité informatique. Ceci pourrait être attribué au nombre croissant d'attaques malveillantes ciblant Android et l'expansion des téléphones Android à travers le monde. Dans ce chapitre, nous présentons quelques-unes des solutions de sécurité proposées pour le système Android. Ces solutions prennent plusieurs directions dans la sécurisation du système Android. Certaines solutions utilisent des techniques déjà existantes dans la sécurité des ordinateurs conventionnels. D'autres solutions sont présentées comme des systèmes de détection d'applications malveillantes sur Android tandis que le dernier type de solutions étend la plateforme Android afin d'enrichir le modèle de sécurité existant avec de nouvelles fonctionnalités de sécurité.

Dans le présent chapitre, nous allons explorer les différentes directions prises par les chercheurs afin d'améliorer la sécurité d'Android.

6.1 Utilisation des techniques existantes

Un certain nombre de travaux ont mis à profit les techniques déjà existantes pour les ordinateurs conventionnels, afin de sécuriser le système Android. Étant donné qu'Android est basé sur un noyau Linux, certaines recherches abordent le problème de la sécurité Android par des approches visant à améliorer la sécurité du noyau Linux. Une des approches proposées (Shabtai, Fledel et Elovici, 2010) consiste à incorporer le cadre du module de sécurité Linux (LSM) dans le système Android en utilisant SELinux.

Le cadre LSM a été conçu afin de permettre au noyau Linux de supporter une variété de modèles de contrôle d'accès. Il permet d'incorporer des modules de sécurité destinés au

noyau Linux permettant ainsi d'éviter le favoritisme envers un modèle de sécurité en particulier (Wright et al., 2002).

En effet, le modèle de contrôle d'accès existant dans la version standard du noyau Linux est le modèle traditionnel de contrôle d'accès discrétionnaire (DAC) (Gollmann, 2006) connu aussi sous le nom de *file permissions*. Dans le modèle DAC, chaque objet (p. ex., processus, fichier) doit appartenir à un propriétaire qui se charge de contrôler les permissions d'accès à cet objet. Ce modèle de contrôle d'accès est dit discrétionnaire dans le sens où un sujet détenant certaines permissions peut passer ces permissions indirectement à d'autres sujets dans le système. Par exemple, l'utilisateur ayant les privilèges de l'administrateur root sous Linux peut attribuer ces permissions d'accès à une ressource à d'autres utilisateurs ayant moins de privilèges.

Généralement, avec le modèle DAC, les utilisateurs spécifient des permissions d'accès permettant à tous les utilisateurs dans le système de lire, modifier ou supprimer les fichiers qui leur appartiennent. Cela peut avoir des conséquences fâcheuses. En outre, dans le cas où un processus réussit à élever ses privilèges, il est capable de modifier et supprimer les fichiers appartenant au système. Ces scénarios d'attaques sont décrits dans la section 4.2.1 et la section 4.2.3.

Contrairement au contrôle d'accès discrétionnaire, le contrôle d'accès mandataire (MAC) (Gollmann, 2006) gère l'accès des sujets (utilisateurs) aux objets (processus, fichiers) en se basant sur une politique de sécurité organisationnelle. Le contrôle d'accès MAC attribue des étiquettes aux sujets et aux données qui définissent les permissions d'accès pour les premiers et la sensibilité des informations pour les derniers. La politique de sécurité utilisée par le modèle MAC regroupe un certain nombre de règles définissant le type d'accès et les actions qu'un sujet peut effectuer sur un objet. Toute opération effectuée par n'importe quel sujet sur n'importe quel objet est testée contre l'ensemble des règles de la politique de sécurité afin de déterminer si l'opération est permise. Par exemple, les militaires doivent souvent manipuler et traiter différents niveaux hiérarchiques d'informations classifiées. Le niveau le plus haut

de cette classification contient les informations classées top secret. Ce niveau est suivi par les niveaux secret, confidentiel, et non classée. L'accès aux informations classifiées est basé sur les autorisations représentant le niveau de confiance attribué à un individu. Ces autorisations sont associées à des niveaux de sensibilité hiérarchiques mentionnés précédemment. Par exemple, une personne ayant une autorisation secret peut accéder aux informations classées secret, confidentiel et les des données non classifiées, mais ne peut pas accéder aux informations qui sont classées top secret.

En général, une politique mandataire peut être représentée par une matrice indiquant pour chaque niveau hiérarchique des sujets quelles sont les actions que ces derniers peuvent effectuer pour chaque niveau d'objets. Cette matrice est définie par un officier de sécurité et elle est utilisée par le noyau des systèmes d'exploitation pour appliquer la politique (p. ex., SELinux). La politique de sécurité de ce modèle est centralisée et ne peut être définie que par un administrateur de politique. Contrairement au modèle DAC, les utilisateurs n'ont pas la possibilité de modifier la politique en place ou de passer outre celle-ci.

Plusieurs modules de sécurité ont été proposés pour le noyau Linux au cours des dernières années dont nous citons AppArmor (Quette, 2011) , TOMOYO (Harada, Horie et Tanaka, 2004) et SELinux (Loscocco et Smalley, 2001). L'intégration de ces modules de sécurité dans le noyau Linux se fait à travers le cadre LSM.

SELinux (Loscocco et Smalley, 2001) est une implémentation du noyau Linux permettant de complémenter le modèle DAC avec un contrôle d'accès mandataire MAC . Ce module fournit plusieurs fonctionnalités de sécurité, notamment le renforcement de type (*type enforcement)*, le contrôle d'accès basé sur les rôles (RBAC). Dans SELinux, les décisions d'autorisation sont fondées sur une politique centralisée chargée depuis un fichier.

Shabtai, Fledel et Elovici (2010) proposent l'intégration de SELinux dans le noyau Linux d'Android, tout en focalisant sur la fonctionnalité *Type Enforcement*. Ladite fonctionnalité de SELinux attribue une étiquette d'identification de type à chaque sujet (processus) et objet

(fichier,socket) en se basant sur le principe du moindre privilège. Ce principe stipule qu'un sujet ne doit posséder que les privilèges nécessaires à son fonctionnement. Les étiquettes attribuées aux objets sont nommées domaines alors que celles attribuées aux objets sont appelées types. Afin d'accéder à un objet, un sujet doit être autorisé à accéder au type de l'objet en question dans la politique de sécurité. Par exemple, une application d'enregistrement de vidéos sur Android se verra attribuer l'étiquette *video_recorder_t* alors que le fichier du pilote de la caméra se verra attribuer l'étiquette *driver_camera_t*. Le fichier de la politique contiendra une règle qui précise que tout processus marqué *video_recorder_t* peut lire, écrire et obtenir les attributs de n'importe quel fichier du pilote portant l'étiquette *camera_driver_t*, mais ne peut pas le supprimer ou changer ses permissions d'accès.

Dans la version standard de Linux, l'application d'enregistrement vidéo ne sera pas en mesure d'accéder au pilote de l'appareil sauf si elle détient des privilèges de l'administrateur root. Cela implique qu'elle sera capable de non seulement accéder au pilote de l'appareil, mais aussi effectuer toute action souhaitée dans le système (p. ex., suppression de fichiers, le changement d'autorisation d'accès). Considérons le cas où l'application d'enregistrement de vidéos présente une vulnérabilité menant à une attaque d'escalade de privilèges comme décrit dans la section. 4.2.1 Avec SELinux, les logiciels malveillants qui réussissent à exploiter cette vulnérabilité ne seront pas en mesure d'avoir un contrôle complet sur le téléphone infecté, puisque la politique en place restreint les actions que le processus exploité pourra exécuter sur le système.

Similairement, Zhang et al., (2007) proposent une technique d'isolement pour le noyau Linux en intégrant SELinux. Cependant, le travail présenté va au-delà du *type enforcement* en intégrant le contrôle d'accès basé sur les rôles (RBAC) et en rajoutant des attributs de vérification d'intégrité liée aux objets.

Le contrôle d'accès basé sur les rôles (RBAC) est un modèle de contrôle d'accès dans lequel l'accès aux objets dépend du rôle assigné au sujet. Lorsqu'un sujet tente d'accéder à un objet, le système extrait la politique de sécurité et vérifie ensuite si le rôle assigné au sujet possède

la permission d'accéder au type de l'objet. Alors que Shabtai, Fledel et Elovici (2010) associent un type à chaque sujet, l'approche de Zhang et al. consiste à attribuer un rôle à chaque sujet, et un ensemble de types à chaque rôle. Les auteurs stipulent que l'utilisation des rôles simplifie la gestion des utilisateurs.

En résumé, l'intégration de SELinux dans Android permettrait de renforcer la sécurité d'Android en appliquant un contrôle d'accès robuste au plus bas niveau du système, plus précisément sur les processus critiques qui s'exécutent avec un haut niveau de privilèges sous Linux. Cependant, le surcoût d'une telle solution reste à évaluer en raison des ressources informatiques très limitées des téléphones mobiles par rapport aux capacités offertes par les ordinateurs conventionnels.

6.2 Détection d'applications malveillantes

Dans la présente section, nous présentons les différentes techniques utilisées pour la détection des logiciels malveillants. Ensuite, nous passons en revue les différentes solutions proposées pour la détection d'applications malveillantes pour Android.

6.2.1 Les techniques de détection de logiciels malveillants

Plusieurs techniques d'analyse pour la détection de logiciels malveillants ont été proposées pour les ordinateurs conventionnels. En général, il existe deux méthodes d'analyse des programmes malveillants : l'analyse dynamique et l'analyse statique.

L'analyse statique se réfère à l'analyse du code source et ses fichiers contenant le code binaire du programme en vue de détecter des intentions malveillantes. Cette technique a l'avantage d'être rapide et efficace. Toutefois, l'analyse statique a ses limites puisque les logiciels malveillants sont de plus en plus sophistiqués et utilisent plus souvent des techniques d'offuscation pour dissimuler leurs intentions malveillantes (Moser, Kruegel et Kirda, 2007). Autrement dit, la recherche de signatures malveillantes entraîne un faible

nombre de *faux positifs* – logiciel légitime signalé comme malveillant –, mais génère un nombre élevé de *faux négatifs* – logiciel malveillant signalé comme légitime – puisqu'il est facile de contourner la méthode de l'analyse statique des logiciels malveillants.

La détection basée sur les signatures est surtout utilisée par les antivirus et se base essentiellement sur les signatures uniques qui définissent les logiciels malveillants. Cette méthode a un degré élevé de précision pour les logiciels malveillants connus à priori, mais elle est inefficace contre les logiciels malveillants inconnus.

La méthode d'analyse dynamique, aussi connue sous le nom d'analyse comportementale, est basée sur les informations recueillies à partir du système d'exploitation pendant l'exécution du programme (Rieck et al., 2008). Le programme est souvent exécuté dans un environnement contrôlé en utilisant des machines virtuelles et des systèmes de bac à sable. Cette technique recueille les appels systèmes et surveille l'accès au réseau et l'utilisation de mémoire ainsi que d'autres informations et compare le tout aux comportements des familles connues de logiciels malveillants.

Toutefois, il est souvent difficile de simuler les conditions réelles susceptibles de déclencher le comportement malveillant du programme. En outre, la période de temps nécessaire pour observer un comportement malveillant n'est pas standard et peut être différente d'un programme à un autre (Shabtai et al., 2011).

L'analyse dynamique est généralement basée sur les heuristiques et se déroule généralement en deux phases : une phase d'apprentissage et une phase de détection. Pendant la première phase, le système de détection tente d'apprendre le comportement normal du système d'exploitation. Durant la phase de détection, toute déviation du comportement normal du système est signalée comme une attaque (Jacob, Debar et Filiol, 2008). Cependant, l'absence de dérivation n'implique pas nécessairement l'absence de logiciel malveillant. L'avantage clé de la présente méthode réside dans sa capacité à détecter les attaques inconnues (zero day) et génère donc un nombre réduit de faux négatif. Cependant, le nombre de faux positifs généré

est plus élevé dû au fait que le comportement de certains logiciels légitimes peut être similaire à un logiciel malveillant sans qu'ils aient des intentions malveillantes.

6.2.2 La détection des applications malveillantes pour Android

Jusqu'à présent, plusieurs systèmes de détection d'intrusions et d'applications malveillantes ont été proposés pour le système Android en utilisant soit l'analyse statique ou dynamique, ou les deux en conjonction. Ces solutions peuvent être catégorisées comme suit : système autonome, systèmes client-serveur et systèmes collaboratifs.

6.2.2.1 Détection autonome

Dans cette approche, les tâches liées à l'analyse et la détection d'applications malveillantes sont entièrement exécutées sur le téléphone de l'utilisateur.

Fsecure (F-Secure) est l'une des multiples compagnies qui présentent des solutions d'antivirus pour Android. Toutefois, les antivirus utilisent souvent la technique l'analyse statique des applications malveillantes qui demande des ressources de calcul élevées et se base généralement sur les signatures pour identifier les applications malveillantes. En raison des ressources limitées des téléphones mobiles, la majorité des travaux présentés en matière de détection d'applications malveillantes pour Android se sont concentrés sur l'analyse dynamique (Burguera, Zurutuza et Nadjm-Tehrani, 2011).

Toujours dans la même catégorie des IDS autonomes, Bläsing et al., (2010) proposent un système de bac à sable pour les applications Android. L'approche proposée utilise les techniques d'ingénierie inverse afin d'obtenir le code source de l'application à analyser à partir de son fichier **.apk*. Ensuite, ils effectuent l'analyse statique du code source obtenu afin de détecter toute intention malveillante. Finalement, l'analyse dynamique est réalisée par l'exécution et le suivi des applications Android dans un environnement totalement contrôlé et sécurisé. Lors de l'analyse dynamique, l'ensemble des événements survenant dans l'appareil

(fichiers ouverts, fichiers accédés, la consommation de batterie, etc.) sont surveillés puis analysés pour détecter tout comportement malveillant.

Enck et al., (2010) proposent le système TaintDroid utilisant des techniques d'analyse dynamiques des données souillées (*taint*) qui permettent de suivre le flux des données sensibles entre les applications tierces.

TaintDroid suppose que les applications tierces téléchargées ne sont pas fiables, et surveille en temps réel la façon dont les applications installées accèdent et manipulent les informations sensibles telles que les informations de localisation GPS ou le carnet d'adresses. L'objectif principal est de détecter le moment pendant lequel les données sensibles quittent le système via des applications non fiables et de faciliter l'analyse des applications par les utilisateurs de téléphone ou des services de sécurité externes.

6.2.2.2 Architecture client-serveur

Contrairement au système autonome, les systèmes appartenant à cette catégorie délèguent la tâche d'analyse et de détection d'applications malveillantes à un ou plusieurs serveurs externes.

Burguera, Zurutuza et Nadjm-Tehrani (2010) stipulent que les techniques d'analyse et de détection d'applications malveillantes utilisées dans les ordinateurs ne peuvent être aussi efficaces avec les téléphones mobiles possédant des ressources limitées. Afin d'y remédier, ils proposent le système *Crowdroid* qui permet d'exécuter le processus d'analyse sur un serveur externe dédié. La collecte des informations à partir des téléphones Android se fait par le biais de l'application *Crowdroid* chargée de surveiller les appels systèmes dans le noyau Linux pour les envoyer ensuite vers le serveur centralisé. L'idée est que chaque utilisateur ayant installé l'application *Crowdroid* contribue à forger une meilleure idée sur le comportement de chacune des applications installées sur son téléphone. Le serveur est responsable d'analyser les données relatives au comportement de chaque application en se

basant sur les données envoyées par un ou plusieurs utilisateurs. Dès qu'un comportement malveillant est détecté, le système alerte les utilisateurs de l'application malveillante.

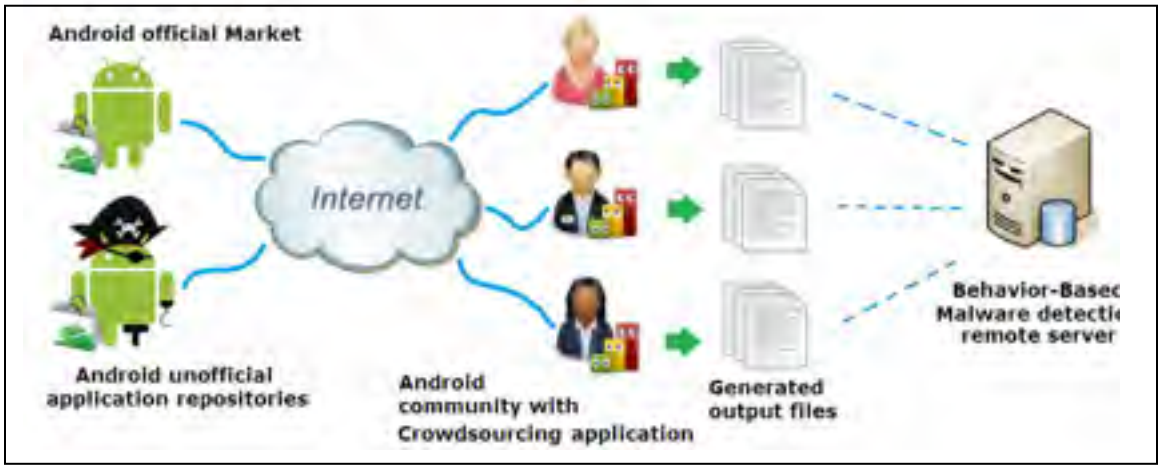

Figure 6.1 Fonctionnement de Crowdroid
Tirée de Burguera, Zurutuza et Nadjm-Tehrani,(2011, p. 4)

6.2.2.3 Système collaboratif

Schmidt et al., (2009) introduisent un système de détection d'applications malveillantes pour Android basé sur le concept de collaboration. La collaboration est utilisée afin de permettre à un téléphone Android d'interagir avec d'autres téléphones dans son entourage en échangeant des données liées à la détection ainsi que les données liées à l'état du système. L'approche proposée met en place un serveur externe à l'instar de *Crowdroid* (voir figure 6.2).

Le système proposé offre essentiellement trois fonctionnalités principales : L'analyse sur l'appareil (autonome), l'analyse à distance (client-serveur) et la collaboration. Le client recueille des données sur l'appareil pour la collaboration ou l'analyse à distance. Afin d'améliorer la détection, les données peuvent être échangées entre deux ou plusieurs téléphones mobiles d'où le concept de collaboration. Ces données peuvent être des résultats de détection ou des événements liés au comportement d'une application donnée. Un téléphone A peut signaler une application malveillante à un téléphone B se situant à son entourage.

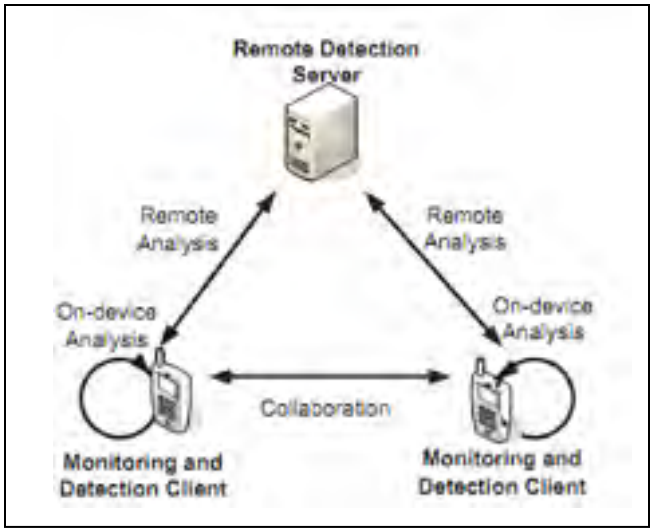

Figure 6.2 Architecture du système collaboratif
Tirée de Schmidt et al., (2009, p. 2)

Chaque fois que la détection autonome ne peut être effectuée (manque de ressources de calcul sur le téléphone), le téléphone mobile peut envoyer des données vers le serveur distant ou à un téléphone dans son entourage. En retour, le serveur peut envoyer les résultats de détection vers le client.

6.3 Solutions étendant le modèle de permissions

Dans cette section, nous présentons les solutions de sécurité enrichissant le modèle de permissions d'Android existant. Les solutions que nous présentons ont modifié et étendu le mécanisme des permissions existant en utilisant des approches différentes.

6.3.1 APEX

Le but principal d'APEX (Nauman, Khan et Zhang, 2010) est de remédier au manque de flexibilité d'Android lorsqu'il s'agit de l'octroi des permissions aux applications. Comme nous avons vu à la section 3.2.2.1, l'utilisateur qui souhaite utiliser une application n'a pas d'autre choix que d'accorder toutes les permissions sollicitées par celle-ci.

APEX est une extension au mécanisme de permissions d'Android permettant aux utilisateurs d'octroyer un sous-ensemble de permissions aux applications au moment de l'installation. Il est également possible de placer des contraintes d'utilisation sur chaque permission accordée à travers une interface utilisateur simple à comprendre et à utiliser.

Ces contraintes sont prises en considération lors de l'exécution de l'application et permettent au système d'octroyer et révoquer les permissions dynamiquement. Par exemple, l'utilisateur peut limiter le nombre de SMS envoyés à cinq SMS par jour pour chaque application ayant la permission *SEND_SMS*. Une fois qu'une application atteint cette limite, le système lui révoque la permission d'envoyer des SMS jusqu'au jour suivant. La figure 6.3 illustre l'interface qui permet à l'utilisateur de placer les contraintes sur une permission et à travers laquelle il peut définir un comportement à suivre pour chacune des permissions (toujours accorder, toujours refuser ou Octroi conditionnel*)* L'approche d'APEX est susceptible de réduire considérablement la menace que représentent les applications malveillantes déclarées décrites dans la section 4.1. Cependant, l'intégration d'APEX dans Android nécessite des modifications dans la plateforme Android touchant l'installateur d'application.

Figure 6.3 Interface d'installation d'APEX
Tirée de Nauman, Khan et Zhang, (2010,p.332)

6.3.2 CREPE

CRePE (Conti, Nguyen et Crispo, 2011) présente un concept de sécurité pour Android sensible au contexte et qui agit au moment de l'exécution des applications. L'idée clé est de prendre des décisions d'utilisation basées sur l'état actuel du téléphone, y compris son emplacement, l'heure et la température et ainsi de suite. Lorsque réunies, ces informations constituent un contexte. Conti, Nguyen et Crispo (2011) soutiennent que le contexte doit être considéré comme un des attributs importants lors de la définition des politiques de sécurité.

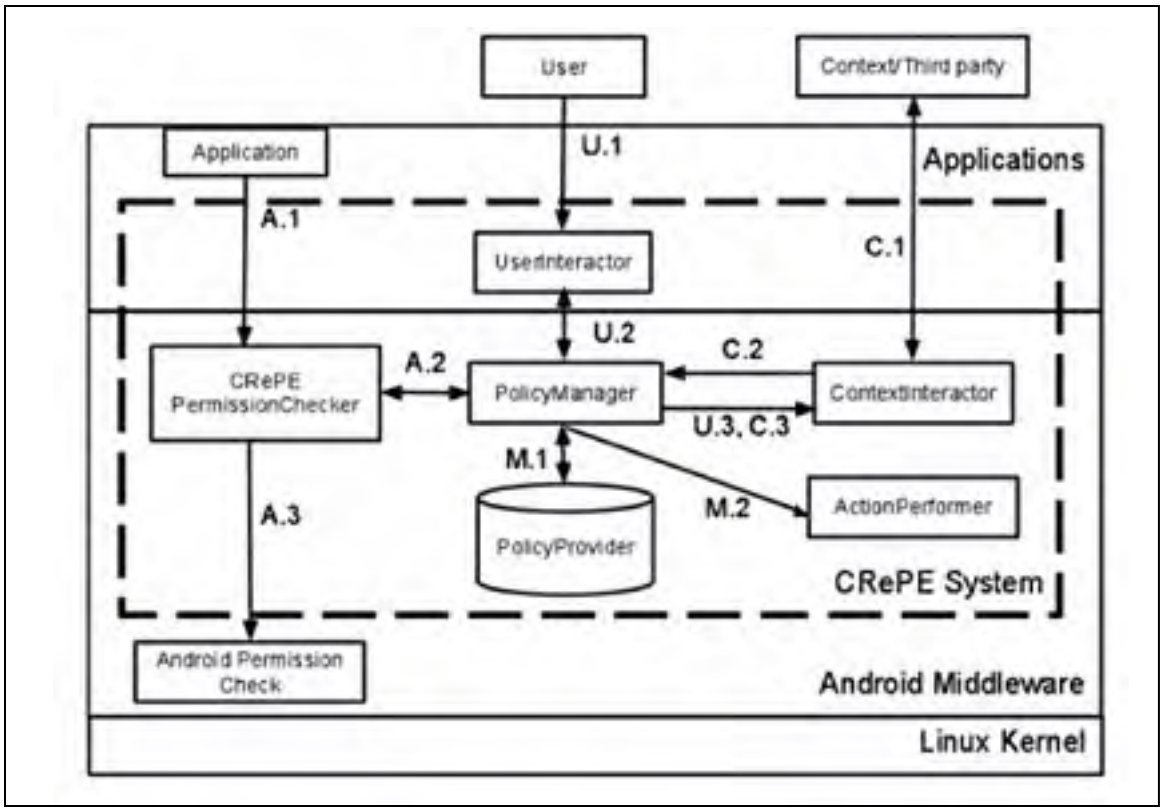

Figure 6.4 Architecture de CRePE
Tirée de Conti, Nguyen et Crispo, (2011, p.337)

L'approche de CRePE est centrée sur l'utilisateur et il appartient aux utilisateurs de définir quel type de politiques de sécurité doit être appliqué à leurs téléphones quand le contexte change.

Outre la définition de la politique de sécurité, l'utilisateur est capable d'activer et de désactiver les politiques de sécurité. Ces politiques sont appliquées au moment de l'exécution des applications. L'état du téléphone est découvert lors de l'exécution afin que le système puisse activer une politique spécifique qui sera la base de toutes les décisions liées à la sécurité. L'architecture de CREPE est présentée dans la figure 6.4.

CRePE intervient lorsqu'une application commence l'interaction avec une deuxième application ou demande l'accès à une ressource donnée. La requête est interceptée par le *CRePEpermissionsChecker* avant d'atteindre la vérification des permissions standard d'Android. Celui-ci interagit avec le P*olicyManager* afin de déterminer si l'utilisation de la permission par l'application est permise par la politique de sécurité active. Le *PolicyProvider* détient une liste des contextes avec leurs politiques correspondantes, ainsi que la liste des contextes actifs. Le *ContextIteractor* est chargé de détecter les contextes et d'avertir CRePE lorsqu'un contexte devient actif ou inactif. Lors d'un changement de contexte, *le ActionPerformer* prend les actions nécessaires pour s'assurer que la politique liée au nouveau contexte est respectée par toutes les applications sur le système (p. ex., arrêter l'exécution d'une application qui ne doit plus accéder à une ressource).

Les contextes peuvent être ajoutés et modifiés par l'utilisateur. Il est également possible pour une tierce partie externe authentifiée d'activer et désactiver un contexte.

6.3.3 Kirin

Kirin (Enck, Ongtang et McDaniel, 2009b) est présenté comme un outil pour gérer l'octroi des permissions au moment de l'installation en se basant sur l'analyse du fichier Manifest de l'application à installer. Kirin peut être utilisé pour interdire l'installation des applications qui demandent une combinaison de permissions dangereuses, en se basant sur une politique définie par l'utilisateur du téléphone mobile. Par exemple, l'utilisateur peut interdire l'installation des applications demandant l'accès au microphone du téléphone et un accès Internet. Kirin peut également analyser les combinaisons de permissions déjà accordées à

toutes les applications installées sur le système. Cette dernière approche permet la détection des applications malveillantes déclarées demandant des permissions dangereuses qui ne sont pas nécessaires pour le fonctionnement prévu de l'application (voir la section 4.1 pour plus de détails).

6.3.4 Secure Application INTeraction (Saint)

Saint (Ongtang et al., 2009) répond aux capacités limitées des applications à contrôler qui peut accéder à leurs composants, tout en améliorant le contrôle des applications sur la façon dont leurs composants sont utilisés par d'autres applications. Également, Saint apporte une amélioration permettant aux applications de mieux sélectionner les composants externes qu'elles utilisent (Ongtang et al., 2009).

L'installateur et le médiateur sont des éléments clés de l'architecture de Saint (voir figure 6.3). L'installateur applique les politiques supplémentaires d'octroi des permissions alors que le médiateur contrôle la communication entre les composants des différentes applications afin de s'assurer que les politiques d'interaction spécifiées par l'appelant et l'appelé sont appliquées.

6.3.4.1 Les politiques de Saint

L'infrastructure Saint permet aux applications de définir des politiques d'installation qui servent à réglementer l'attribution des permissions qui protègent leurs composants, ainsi que des politiques d'interaction qui gèrent les interactions entre les applications.

6.3.4.2 Politique d'installation

Android permet aux applications de définir leurs propres permissions qui sont ensuite utilisées afin de protéger les composants de celle-ci. Saint va au-delà du modèle traditionnel d'Android, en permettant à ces applications de définir les conditions sous lesquelles ces

permissions sont octroyées aux applications qui les demandent. Avec Saint, les applications peuvent exercer un contrôle sur l'attribution des permissions déclarées à travers une politique explicite (Ongtang et al., 2009).

Outre l'octroi des permissions traditionnelles d'Android, Saint définit deux types d'octroi de permissions supplémentaires. Le premier est régi par une politique d'installation basée sur les signatures. Dans le système d'Android actuel, les permissions de type signature ne peuvent être octroyées qu'aux applications ayant été signées par la même clé privée que l'application qui a défini la permission. Cependant, avec Saint, une application peut se voir accorder une permission de type Signature, même si celle-ci porte une signature différente, et ce, en se basant sur la politique des signatures. Soit A une application qui protège un de ces composants avec une permission P_A de type Signature qu'elle définit dans son fichier Manifest (voir section 3.2.2.2). Avec Saint, le développeur de A peut spécifier la condition suivante : Toute application demandant la permission P_A doit porter la signature « xxx…xxx » ou la signature « yyy…yyy ». Cette approche permet d'établir une relation de confiance entre les applications provenant des développeurs différents.

La deuxième politique d'installation proposée se base sur la configuration de l'application à installer telle que la combinaison des permissions qu'elle demande ainsi que sa version. Par exemple, le développeur de A peut refuser l'octroi de la permission P_A à toute application ayant la permission d'accéder à internet et lire les SMS reçus. Cette politique aide à contourner les applications malveillantes déclarées visant à exploiter une application installée sur les téléphones Android.

6.3.4.3 La politique des interactions

La politique d'interaction telle que son nom l'indique, gère les interactions entre les composants des différentes applications. Dans le modèle de sécurité actuel d'Android, lors d'une interaction entre deux composants appartenant à deux applications différentes, le

système vérifie seulement si l'application appelante dispose des permissions nécessaires pour communiquer avec l'application appelée.

Avec Saint, il devient possible à l'application de définir une politique d'interaction basée sur les signatures afin de restreindre les applications qui peuvent interagir avec elle. Par exemple, une application peut refuser d'interagir avec une application B si la signature de celle-ci ne figure pas dans la liste des signatures autorisées dans la politique d'interaction.

D'autre part, Saint permet aux applications de définir des politiques basées sur les configurations des applications qui l'appellent (p. ex., la version et l'ensemble des permissions demandées).

Saint applique des politiques d'exécution de deux types : les politiques d'accès pour identifier les exigences de sécurité de l'application appelante et les politiques d'exposition pour identifier les exigences de sécurité de celle appelée. Le dernier type de politiques supportées par Saint est basé sur les contextes, utilisant le même principe employé par CRePE (Conti, Nguyen et Crispo, 2011).

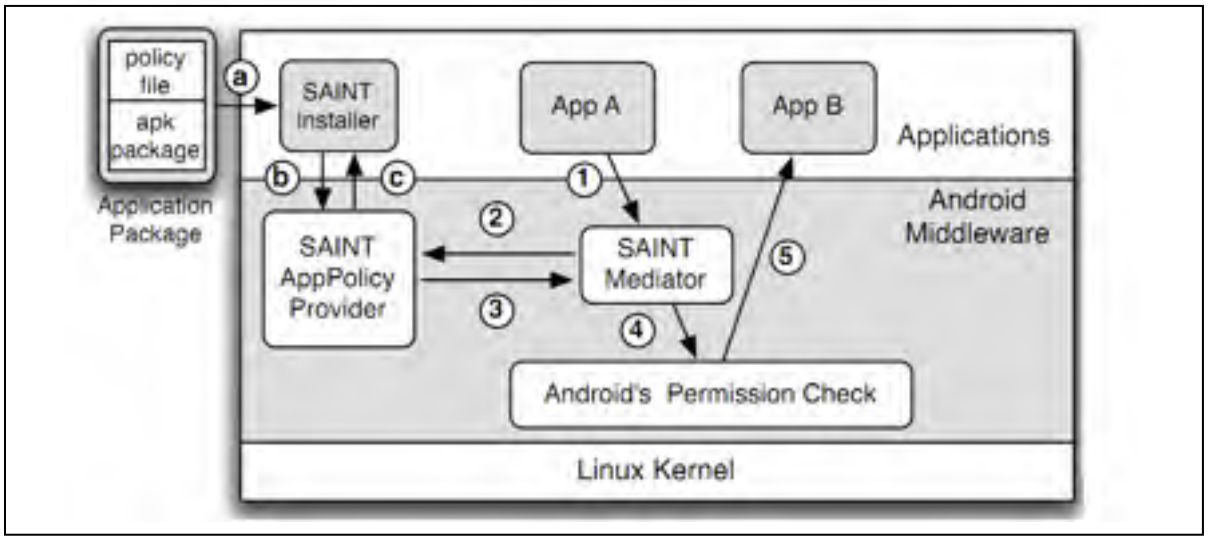

Figure 6.5 Architecture de Saint
Tirée de Ongtang et al. (2009, p.344)

6.3.4.4 Application de la politique d'installation

La politique d'installation est appliquée par l'installateur de Saint qui se réfère au fournisseur de politique *AppPolicyProvider*. Ce dernier contient une base de données renfermant l'ensemble des politiques d'installation et d'interaction relatives à toutes les applications installées sur le système. Au moment de l'installation d'une application, l'installateur Saint *SaintInstaller* extrait les permissions demandées dans le fichier Manifest et pour chaque permission demandée, il consulte le fournisseur de politique. Celui-ci retourne une décision relative à la permission demandée, en se basant sur les politiques existantes. Dans le cas où l'application échoue à remplir une seule condition de la politique, l'installation est annulée.

6.3.4.5 Application de la politique d'interaction

La politique d'interaction est appliquée par le médiateur *Saint Mediator* introduit par Saint. Lors d'une communication entre deux composants de différentes applications, le médiateur Saint intercepte l'appel. Ensuite, il a recours au fournisseur de politique *AppPolicy*, qui vérifie les conditions existantes dans la politique d'interaction et spécifiées par l'application appelante et l'application appelée. La vérification effectuée concerne les signatures et les configurations de l'appelé et l'appelant ainsi que le contexte dans lequel se déroule la communication. Si les conditions sont remplies, la communication se redirige vers la vérification de permission standard d'Android, sinon elle est bloquée par le médiateur.

CHAPITRE 7

MÉTHODES DE DÉVOPPEMENT PROPOSÉES POUR SÉCURISER LES APPLICATIONS ANDROID

Dans la première partie du présent chapitre, nous rappelons les mécanismes de sécurité offerts aux développeurs Android afin de sécuriser leurs applications. Dans la seconde partie, nous proposons des nouvelles méthodes de programmation axées sur la sécurité des applications Android. Notre travail consiste à permettre aux développeurs Android de définir des politiques de sécurité qui sont appliquées afin de protéger leurs applications contre les applications malveillantes visant à exploiter les applications légitimes.

7.1 Motivation et objectifs

Notre travail consiste à proposer de nouvelles méthodes de programmation qui permettent aux développeurs de définir des politiques de sécurité pour leurs applications. Ce travail s'inspire des différentes solutions de sécurité présentées dans le chapitre 6 et particulièrement des solutions APEX (voir la section 6.3.1) et SAINT (voir la section 6.3.4). Alors que ces solutions sont présentées dans l'optique de protéger le téléphone Android contre toute exploitation malveillante, notre travail est orienté vers la protection des applications Android de toute utilisation abusive. À l'instar de SAINT, nos méthodes permettent aux développeurs d'applications Android de définir des politiques de sécurité régissant les interactions de leurs applications avec celles qui sont déjà installées. En outre, nous introduisons des contraintes d'utilisation des applications telles que proposées par la solution APEX. Les méthodes que nous présentons ici ne nécessitent aucune modification de la plateforme Android, contrairement aux solutions dans le chapitre 6, et sont applicables au système Android actuel.

Cependant, les solutions que nous introduisons se limitent à protéger les applications installées sur les téléphones Android non rootés volontairement (voir la section 4.2.3), ou par une application malveillante (voir la section 4.2.1). En effet, dans le cas où une application réussit à obtenir un accèes root, il nous est impossible de lui restreindre l'accès aux fichiers

des applications dû à la conception du noyau Linux qui offre toute les privilèges à l'utilisateur root. De plus, notre champ d'action se limite au moment de l'exécution des applications. En effet, les restrictions imposées par le système Android nous empêchent d'intervenir aux moments de l'installation afin d'empêcher l'installation d'application malveillante. Cependant, en dépit de ce fait, nous estimons qu'il est toujours possible de contrer les applications malveillantes visant à exploiter les applications légitimes au moment de leur exécution, ce qui représente l'objectif de notre travail.

7.2 Méthodes de protections existantes

Comme nous l'avons vu dans le chapitre 3, le modèle de sécurité d'Android est essentiellement centré sur le modèle de permissions. La présente section présente un rappel des mécanismes de sécurité offerts pour les développeurs Android, ainsi que les façons adéquates de les utiliser.

7.2.1 Utilisation des permissions

Une application Android peut avoir un ou plusieurs composants sensibles devant être protégés par les développeurs. Un composant pouvant effectuer des appels téléphoniques ou envoyer des SMS est considéré comme sensible, d'où l'importance de le protéger avec les permissions définies par les développeurs. Un composant sensible non protégé peut conduire à une attaque d'escalade de privilèges où une application non privilégiée utilise un composant d'une application privilégiée afin d'effectuer des actions non autorisées. Les développeurs d'applications ont la responsabilité de protéger leurs applications en définissant les droits d'accès à leurs applications via la déclaration des permissions décrite dans la section 3.2.2.1. Lors de la déclaration d'une permission, les développeurs doivent prendre en considération le fait que les utilisateurs des téléphones mobiles ont peu de connaissances sur le fonctionnement de ceux-ci. Par conséquent, la description accompagnant la permission doit être claire et compréhensible par tous. Si l'application cible des utilisateurs de nationalités différentes, il est utile alors de rédiger la description en plusieurs langues.

Par exemple, pour sécuriser un composant qui envoie l'emplacement du téléphone mobile par SMS, le développeur définit une permission portant le nom de *ENVOI_EMPLACEMENT_SMS* et la décrit comme suit traduite en plusieurs langues : « Cette application a besoin de connaître votre emplacement et l'envoyer par SMS ».

7.2.2 Utilisation des composants privés

Une application peut contenir des composants qui ne sont destinés qu'à l'usage privé de l'application qui les contient (composant privé). D'autre part, certains composants doivent être accessibles pour les applications externes. La visibilité des composants vis-à-vis des autres applications est gérée par l'attribut *exported* décrit dans la section 3.2.5.

Lors du développement, il est recommandé de séparer les parties du code devant demeurer publiques des parties privées. Idéalement, un composant public doit contenir uniquement du code destiné à l'usage des applications externes. Ainsi, le développeur évite que son application soit utilisée de manière malveillante.

7.2.3 Sécuriser les données des applications

Les données des applications sont protégées à l'aide du mécanisme *File access* du noyau Linux. À chaque fichier, le noyau Linux associe un tuple de permissions définissant les droits d'accès au fichier.

La SDK d'Android offre des moyens de programmation qui permettent aux développeurs de spécifier les permissions en lecture et écriture sur les fichiers appartenant à leurs applications. Il s'agit des trois constantes suivantes (AOSP, 2008e) :

- *MODE_WORLD_READABLE* : indique que toutes les applications peuvent lire le fichier;
- *MODE_WORLD_WRITEABLE* : indique que toutes les applications peuvent modifier le fichier;

- *MODE_PRIVATE* : indique que le fichier n'est accessible que par l'application à laquelle il appartient.

Ces constantes prennent effet lorsqu'elles sont passées à l'une des méthodes suivantes :

- *GetSharedPreferences(String, int mode) :* Cette fonction récupère et modifie le contenu des préférences, qui sont stockées avec un fichier donné.
- *OpenFileOutput(String, int mode)* : Cette fonction est utilisée pour ouvrir et lire le contenu d'un fichier qui est associé avec le paquet d'application. Le fichier sera créé s'il n'existe pas.
- *openOrCreateDatabase(String, int mode,SQLiteDatabase.CursorFactory)* : Cette fonction ouvre une nouvelle base de données SQLite qui est associée aux paquets d'application. Le fichier de base de données sera créé s'il n'existe pas.

Le noyau Linux a la responsabilité d'appliquer ces paramètres de sécurité en utilisant le mécanisme décrit dans la section 3.1.2.

```
File = openFileOutput("userInfo", Context.MODE_WORLD_READABLE);
```

Figure 7.1 Configuration des droits d'accès sur un fichier

La figure 7.1 présente un exemple de création de fichier accessible à toutes les applications. La création d'un tel fichier peut présenter un risque de divulgation d'information sensible, comme dans le cas de l'application de *Skype* (section 4.2.4.2). Les développeurs d'applications pour Android doivent vérifier les permissions sur chaque fichier appartenant à l'application. Une attention particulière doit être portée sur les fichiers contenant des informations sensibles.

7.2.4 Sécuriser les intentions

Les intentions sont utilisées comme moyen de communication entre les composants des applications. Elles peuvent également servir à échanger les données entre ceux-ci. Les intentions peuvent être utilisées de différentes manières. Elles peuvent être utilisées pour démarrer une activité, démarrer un service, ou encore diffuser vers les récepteurs de diffusions.

Lors de l'envoi d'intentions, les développeurs doivent penser à protéger l'intention avec une permission afin de préserver la confidentialité ainsi que l'intégrité des données qu'elle contient. Seules les applications qui détiennent cette permission seront en mesure de recevoir l'intention. Cependant, il est toujours possible pour les applications malveillantes d'obtenir ces permissions et de lire les données contenues dans l'intention en les demandant aux utilisateurs souvent naïfs et imprudents. Pour cette raison la meilleure pratique consiste à éviter autant que possible d'insérer des informations sensibles dans une intention (p. ex., numéro de carte de crédit, mot de passe).

Aussi, lors de la réception d'intentions provenant d'applications externes, les développeurs doivent veiller à ce que les données contenues dans l'intention reçue soient conformes aux données attendues par l'application. En effet, même si un composant est défini comme composant privé (*exported = false*), il est toujours possible pour les applications externes de lui adresser des intentions en spécifiant son nom exact dans l'intention. Dans ce cas, le composant externe se fait passer pour un composant appartenant à la même application que le composant cible, et il est donc digne de confiance. Cette attaque est similaire aux attaques d'usurpation d'adresse IP. Parmi les différents scénarios de cette attaque, un attaquant peut envoyer des paquets IP en utilisant l'adresse IP d'une machine appartenant à un réseau local, dans le but de s'infiltrer dans ce réseau. Cette attaque est efficace surtout lorsqu'une relation de confiance est établie entre les différentes machines appartenant au réseau local et lorsque l'authentification des machines est effectuée en se basant sur leur adresse IP. Dans ce cas,

tous les paquets envoyés par l'attaquant sont considérés légitimes et seront acceptés par les machines locales appartenant aux réseaux.

Dans Android, un composant appartenant à une application malveillante a la possibilité de se faire passer pour un composant appartenant à l'application légitime ciblée par l'attaque. Malgré qu'un composant peut être désigné comme privé, il est toujours possible de lui envoyer des intentions en précisant son nom exact, ce qui laisse à penser que le composant appelant appartient à la même application.

Par exemple, une application bancaire dispose d'un service qui envoie un SMS à l'institution bancaire de l'utilisateur pour ordonner une transaction. Le contenu du message SMS à envoyer est obtenu à partir d'une activité appartenant à la même application. L'activité en question récolte les informations liées à la transaction à effectuer (saisies par l'utilisateur). Ces informations sont ensuite acheminées vers le service chargé d'envoyer les SMS en utilisant une intention contenant le message à envoyer et le numéro de l'institution bancaire qui effectuera la transaction. Dans ce cas, une application malveillante ayant le nom du service (tel qu'il apparait dans le fichier Manifest de l'application) pourrait forger une intention contenant un faux message et un numéro de téléphone. À la réception de l'intention, le service de l'application bancaire extrait le contenu du message à envoyer et le numéro du destinataire comme si le message parvenait de l'activité appartenant à la même application que lui.

L'exemple ci-dessus souligne l'importance de filtrer les données contenues, dans les intentions reçues peu importe si elle était envoyée par un composant interne ou externe à l'application.

Dans le cas des applications impliquant l'envoi de SMS, il est fortement recommandé de limiter les numéros vers lesquels une application peut envoyer des SMS et éviter de recevoir ce numéro à partir des composants appelants. De même, il est recommandé d'utiliser des formats de messages prédéfinis qui ne doivent contenir que quelques champs libres pouvant

être fournis par les applications appelantes. Les entrées contenues dans ces champs doivent être filtrées par l'application réceptrice afin d'éviter les attaques similaires aux attaques d'injection SQL (OWASP, 2004a). Une attaque de ce type consiste à insérer ou injecter une requête SQL via un formulaire de saisie de données du client vers une application ou un site web. Dans une attaque SQL réussie, l'attaquant peut lire, modifier, supprimer ou rendre public le contenu des données sensitives qui résident dans la base de données. Afin d'éviter cette attaque, il est recommandé d'appliquer un principe essentiel de la sécurité qui consiste à filtrer les données saisies par l'utilisateur (*Filter the input*). Ce principe demeure valide pour les applications qui reçoivent des intentions contenant des données envoyées par des applications externes. Dans l'exemple de l'application envoyant des SMS, il est primordial de filtrer le champ de données contenant le numéro de téléphone du destinataire du SMS et de n'accepter que des numéros ayant un format spécifique (p. ex., les numéros qui commencent par 1-888). Ce mécanisme permettrait de contrer les applications malveillantes visant à envoyer des SMS à des numéros surtaxés entraînant des pertes financières pour l'utilisateur et des gains financiers pour le développeur de l'application malveillante.

7.3 Définition et application des politiques de sécurité pour les applications Android

Android propose plusieurs méthodes pour les développeurs afin de protéger leurs applications. Cependant, ces méthodes (voir le chapitre 3) sont présentées comme des moyens de sécurité séparés et sont souvent utilisées de façon arbitraire et appliquées par composant. Autrement dit, chaque développeur d'applications Android est responsable de définir sa propre stratégie de sécurité et de veiller à ce que son application ne contienne pas de vulnérabilité. Cependant, la grande majorité des développeurs ne sont pas conscients du danger présenté par les applications malveillantes et ignorent souvent l'impact qu'une application vulnérable peut avoir sur la confidentialité des données de l'utilisateur et la sécurité du téléphone mobile. De ce fait, un module de sécurité centralisé où les développeurs ont la possibilité de définir plusieurs types de politique de sécurité est plus que nécessaire. Un tel service viserait à rendre plus facile pour les développeurs de définir les besoins de sécurité de leurs applications pour ensuite les appliquer.

Dans notre travail, nous combinons les différentes mesures de sécurité fournies par l'API Android, afin de les présenter dans un module de sécurité centralisé présenté comme un tout et qui peut être relativement facile à intégrer dans les applications Android. Ce module englobe plusieurs fonctionnalités telles que la définition de politique de sécurité et un nouveau modèle d'authentification.

Finalement, nous avons développé une application Android qui utilise les méthodes présentées pour montrer les capacités de notre mise en œuvre de la sécurité.

7.3.1 Définition des politiques de sécurité

Notre objectif consiste à permettre au développeur ainsi qu'aux utilisateurs de définir des politiques de sécurité qui seront appliquées par les applications au cours de leur exécution. Ces politiques sont définies lors du développement de l'application par son développeur. Par ailleurs, le développeur pourrait offrir aux utilisateurs la possibilité de définir par eux même ces politiques de sécurité. Dans cette section, nous définissons quatre types de politiques de sécurité :

- La politique basée sur l'identité des applications;
- La politique basée sur l'identité des développeurs;
- La politique basée sur les permissions;
- La politique basée sur les actions.

7.3.1.1 La politique basée sur l'identité des applications

La politique fondée sur l'identité d'une application contient une combinaison de plusieurs règles. Par identité, nous entendons le nom du package d'une application qui est un identifiant unique de l'application installée sur un téléphone Android (p. ex., *com.android.skype*). Chaque règle définit la liste à laquelle appartient un package donnée. Dans la présente politique, nous définissons deux listes :

- Liste noire : cette liste contient les noms des packages des applications qui ne sont pas autorisées à interagir avec notre application. Ces applications ne sont pas autorisées à demander des données, envoyer des diffusions ou démarrer une activité ou un service appartenant à notre application. En revanche, notre application s'abstient d'envoyer toute demande d'interaction à ces applications comme l'envoi d'intentions et la demande de données à toutes les applications figurant dans cette liste.
- Liste blanche : cette liste contient le nom des packages des applications que notre application considère comme des applications légitimes.

Le développeur peut définir les règles de la politique en se basant sur une liste des applications malveillantes déjà connues, ainsi qu'une liste contenant les applications légitimes. Chaque fois qu'un développeur pense qu'une application particulière peut compromettre la sécurité de sa propre application, il ajoute une règle classant cette application dans la liste noire. Par ailleurs, l'application utilisant les méthodes proposées est capable de définir de nouvelles règles dans la politique en se basant sur une suite de contrôles de sécurité que nous exposons plus tard dans cette section.

7.3.1.2 La politique basée sur l'identité des développeurs d'applications

La politique basée sur l'identité des développeurs permet à l'application de prendre des décisions liées à la sécurité en fonction de la réputation du développeur d'une application donnée. Comme nous l'avons vu dans la section 3.2.1, toutes les applications Android doivent être signées avec un certificat numérique. Les signatures numériques des applications appartenant à un même développeur peuvent toutes être vérifiées grâce à la clé publique de celui-ci.

La politique basée sur l'identité des développeurs contient une liste noire des clés publiques appartenant à des développeurs malveillants. L'idée est de prévenir toute application fournie par un développeur considéré malveillant d'interagir avec notre application. Les clés

publiques présentes dans cette politique sont définies par le développeur qui définit la liste noire.

7.3.1.3 La politique basée sur les permissions

La politique basée sur les permissions vise à définir un certain nombre de critères liés aux permissions à partir desquelles notre application peut prendre une décision quant à la légitimité d'une application qui lui est externe. La politique définit deux types de règles :

- la règle « Doit avoir » : Une règle de ce type exige qu'une application ayant la permission P_a, doit aussi avoir l'ensemble de permissions $\{P_i, P_{j,...}\}$.
- La règle « Ne doit pas avoir » : Une règle de ce type stipule qu'une application ayant la permission P_a ne doit pas être en possession d'aucunes permission se trouvant dans l'ensemble de permissions $\{P_i, P_{j,...}\}$.

Il appartient au développeur de définir les règles de la politique basée sur les permissions selon les fonctionnalités que son application offre. Prenons l'exemple où le développeur développe une application contenant une activité appelée *CallPhoneActivity* capable d'effectuer des appels téléphoniques. Le développeur souhaite que les applications installées sur le téléphone puissent démarrer l'activité *CallPhoneActivity* et lui transmettre un numéro de téléphone à appeler. Comme nous l'avons décrit dans la section 3.2.2.2, le développeur a la possibilité de protéger l'activité *CallPhoneActivity* en définissant une nouvelle permission (p. ex. MY*_APPLICATION_CALL_PHONE*) dans son fichier Manifest et de l'utiliser ensuite pour protéger son activité. Toute application qui désire passer un appel téléphonique à travers l'activité *CallPhoneActivity* doit demander la permission *MY_APPLICATION_CALL_PHONE* à l'utilisateur.Toutefois, le développeur veut faire en sorte que toute application externe ayant la permission *MY_APPLICATION_CALL_PHONE* ait également la permission standard *android.permission.CALL_PHONE* d'Android qui permet à une application de lancer des appels téléphoniques. La figure 7.2 montre l'exemple de règle devant être définie dans ce cas.

Si une application détient *MY_APPLICATION_CALL_PHONE*
alors elle doit aussi avoir la permission
android.permission.CALL_PHONE.

Figure 7.2 Exemple de règle appartenant à la politique basée sur les permissions

De ce fait, l'application ne donne aucun nouveau droit aux autres applications. Ces dernières peuvent accéder à la nouvelle application que si elles avaient déjà le droit d'effectuer un appel téléphonique.

Dans un autre cas de figure, le développeur dispose d'un service qui fournit le contenu des SMS reçus vers des applications externes sur demande. Le service est protégé par la permission *GET_SMS_CONTENT* définie par le développeur. Ce dernier ne veut pas que les applications externes entrainent la fuite de telles informations confidentielles en les envoyant à un serveur externe (voir figure 7.3). La politique basée sur la permission offre une telle protection en permettant aux développeurs de définir des règles qui répondent à leurs besoins en matière de sécurité.

Si l'application possède la permission *GET_SMS_CONTENT*,
alors elle **ne doit pas avoir** la permission *INTERNET*

Figure 7.3 Exemple 2 de règle appartenant à la politique basée sur les permissions

7.3.1.4 La politique basée sur les actions

La politique basée sur les actions impose des contraintes d'utilisation sur les fonctionnalités fournies par l'application (p. ex. Appel téléphonique, envoi de SMS). Le développeur a la possibilité de définir un ensemble de règles contrôlant à quels moments les fonctionnalités offertes par son application peuvent être utilisées par d'autres applications. Le développeur

peut définir une règle dans la politique pour chaque action. Une action fait référence à la chaine d'actions spécifiée dans l'intention reçue pour démarrer un composant. Pour chaque action, le développeur peut spécifier un certain nombre de conditions qui forme un contexte. Le contexte peut être défini par des contraintes temporelles ou spatiales (liée à l'emplacement GPS du téléphone Android). Par exemple, le développeur souhaite limiter le nombre de fois que l'activité *CallPhoneActivity* peut être utilisée par jour. Il peut également avoir besoin de définir une période de temps de la journée pendant laquelle aucune application externe ne peut l'utiliser. Un exemple de règle appartenant à la politique basée sur les actions est illustré dans la figure 7.4.

```
Effectuer Appel téléphonique seulement si heure entre 8 h et
20 h et nombre d'appels effectués aujourd'hui < 20
```

Figure 7.4 Exemple 1 de règle appartenant à la politique basée sur les actions

Le développer peut également utiliser cette politique afin de limiter le nombre de SMS que son application peut envoyer. En effet, plusieurs applications malveillantes ont pour but d'envoyer des SMS pour générer des gains financiers au profit de leurs développeurs ou encore pour envoyer du pourriel. La politique des actions peut limiter l'impact de ce type d'applications malveillantes. L'impact de ces applications malveillantes peut être atténué considérablement grâce à la politique basée sur les actions. Un développeur pourrait ajouter une règle dans la politique afin de limiter le nombre de SMS que son application peut envoyer par jours tel qu'illustré dans la figure 7.5.

```
Envoyer SMS seulement si nombre de SMS envoyé
aujourd'hui < 10
```

Figure 7.5 Exemple 2 de règle appartenant à la politique basée sur les actions

7.3.2 Prototype de mise en œuvre des politiques

Dans notre preuve de concept, nous avons choisi de stocker les différentes politiques de sécurité sous la forme de base de données stockée sous la forme de fichier SQLite dont l'accès n'est accordé qu'à l'application qui définit les politiques. La base de données *Policies* regroupe quatre tables de données qui représentent les quatre politiques de sécurité : *AppIdentityPolicy*, *DevIdentityPolicy*, *PermissionsPolicy* et *ActionsPolicy*. Chaque enregistrement dans une table de donnée représente une règle dans la politique de sécurité correspondante à la table. Le tableau 7.1 montre un exemple d'une table de base de données qui correspond à la politique basée sur les permissions

Tableau 7.1 Implémentation de la politique basée sur les permissions

Id	***permission***	***type***	***Permission1***	***Permission 2***
1	*MY_APPLICATION_CALL_PHONE*	must_have	*android.CALL_PHONE*	none
2	*GET_SMS_CONTENT*	must_not_have	*INTERNET*	SEND_SMS

Dans notre implémentation, nous avons opté pour deux méthodes différentes pour le remplissage et la modification du contenu des tables de base de données correspondantes aux différentes politiques. Dans la première méthode, le développeur se charge d'insérer les règles dans les tables des politiques de façon entièrement programmatique et transparente à l'utilisateur. Le développeur est alors responsable de rechercher les informations nécessaires à la définition des quatre politiques.de sécurité que nous avons décrite dans la section 7.3.1. Ces informations sont :

- les noms des applications malveillantes connus à priori et qui peuvent présenter une menace pour son application;
- les clés publiques des développeurs de ces applications malveillantes;
- Définir les combinaisons de permissions qu'il juge dangereuses;
- Définir les contraintes qu'il souhaite imposer sur l'usage des fonctionnalités de son application.

La modification des politiques de sécurité par le développeur est dans ce cas effectuée à travers la mise à jour de l'application, et ce, à chaque fois qu'une modification d'une ou plusieurs politiques s'avère nécessaire. Durant la mise à jour de l'application, l'ensemble de la politique déjà existante est remplacé par les nouvelles politiques contenues dans la mise à jour de l'application. Cette approche met l'entière responsabilité de la sécurité sur les épaules du développeur qui est dans ce cas responsable de toute éventuelle brèche de sécurité.

La deuxième méthode permet à l'utilisateur de l'application de définir les règles des politiques de sécurité à travers un système de gestion des politiques que nous introduisons au cœur de l'application. Ledit système offre une multitude d'interfaces utilisateurs implémentées en utilisant les composants activité permettant de consulter, modifier ou supprimer les règles appartenant aux différentes politiques de sécurité.

Toutefois, la question de savoir si nous devrions permettre aux utilisateurs des systèmes mobiles de définir leurs propres politiques de sécurité est en prise avec les débats au sein de l la communauté des experts en sécurité. Certains pensent que la gestion politique de sécurité ne doit guère être relayée aux utilisateurs, puisque les développeurs d'applications sont les plus qualifiés à analyser les exigences de sécurité pour leurs applications et devraient donc être ceux qui définissent les politiques de sécurité de celles-ci. Donner ce privilège aux utilisateurs, souvent peu connaisseurs en sécurité, pourrait mettre la sécurité globale de l'application et du système mobile en péril. Toutefois, ne pas offrir cette possibilité aux utilisateurs pourrait les empêcher occasionnellement d'effectuer des opérations légitimes en raison des politiques de sécurité qu'il ne contrôle pas.

En revanche, certains stipulent que les utilisateurs d'appareils mobiles devraient être impliqués dans le processus de définition des politiques de sécurité puisqu'ils sont les ultimes propriétaires de leur appareil mobile et devraient donc avoir un contrôle total de celui-ci, tout en assumant les conséquences qui peuvent en résulter.

Le choix entre les deux approches est laissé au jugement du développeur de l'application. Dans notre preuve de concept, nous avons opté pour une approche mixte qui implique le développeur et l'utilisateur dans le processus de définition des politiques de sécurité pour l'application. Les politiques sont définies par le développeur de façon programmatique et peuvent être enrichies par des règles supplémentaires pouvant être ajoutées par l'utilisateur à travers les différentes interfaces visuelles (activités) de gestion des politiques de sécurité que nous introduisons dans les applications Android (voir figure 7.6)

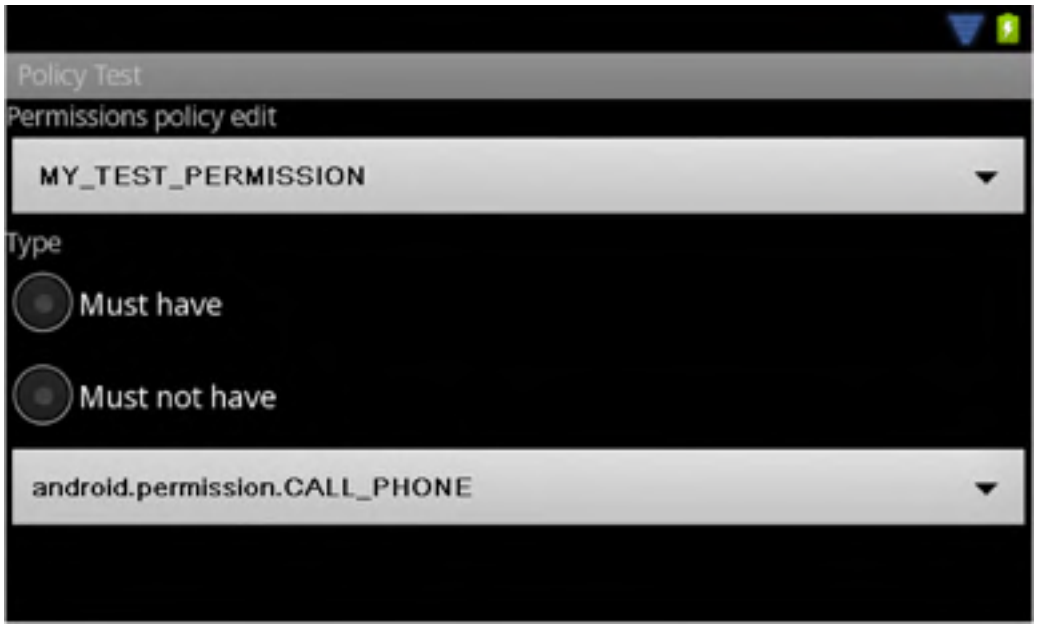

Figure 7.6 Activité d'insertion de nouvelle règle dans la politique

7.3.3 Certification et authentification des applications Android

Une application Android peut être composée de quatre types de composants : activité, service, fournisseur de contenu et récepteur de diffusion. Les trois premiers types de composants peuvent être accédés par des applications externes en utilisant le mécanisme d'intentions. Toutefois, selon l'API d'Android, seul un composant de type service nous permet d'obtenir des informations relatives à l'identité de l'application qui l'a appelé. Pour les autres types de composants, Android ne permet pas d'avoir des informations relatives à

l'application qui a envoyé une intention à notre application autre que si elle dispose des permissions nécessaires pour accéder à notre application (voir section 3.2.2.3). Dans l'exemple de l'activité *CalPhoneActivity*, cette activité ne peut obtenir d'information sur l'identité de l'application qui lui a demandé d'effectuer un appel téléphonique.

Un mécanisme d'authentification (Gollmann, 2006) permet la vérification de l'identité revendiquée par une entité (qui est l'application appelante) par une autre entité (notre application). Le modèle d'authentification que nous présentons a pour but de permettre aux composants de type activité, récepteur de diffusion et fournisseur de contenu appartenant à notre application d'identifier et authentifier l'application appelante durant une interaction et être en mesure de prendre une décision liée à la sécurité en se basant sur le résultat de l'authentification.

Notre mise en œuvre comporte trois composants essentiels : un service que nous appelons, Service de certification (*CertificationServi*ce), un authentificateur (*Authenticator*) et un vérificateur de contraintes d'usages.

7.3.3.1 Le service de certification

En raison des limitations du modèle actuel d'Android, nous avons choisi d'implémenter la logique de la sécurité servant à évaluer la légitimité d'une application appelante comme un composant de type service. Ce choix émerge du fait que seuls les services sont capables d'obtenir des informations suffisantes sur l'application appelante en plus de leur capacité à s'exécuter de façon permanente en arrière-plan.

Avant de démarrer une activité ou un service, ou d'envoyer une diffusion à notre application, une application externe doit se lier au service de certification et demander un jeton de session qui sera utilisée par la suite pour l'authentifier face au composant de notre application qu'elle compte appeler. Ce jeton est stocké dans une base de données comportant les noms des packages reconnus légitimes et les jetons qui leur sont assignés (un jeton par application).

Dans notre implémentation, nous utilisons un générateur de session capable de générer des jetons de session uniques pour chaque session. Le renouvellement du jeton de session est effectué à chaque fois qu'une application désire interagir avec notre application. Aussi, nous définissons trois jetons par défaut considérés non valides : 7700, 7600 et 7500. La signification de chacune de ces valeurs est présentée plus tard dans cette section.

```
public int generateSessionToken(){

    SecureRandom random = new SecureRandom();
    BigInteger tokenRand = new BigInteger(130,random);
    String keyString = tokenRand.toString();
    int token = Integer.parseInt(keyString);
    return token;
}
```

Figure 7.7 Fonction de génération de jeton de session

Chaque fois qu'une application se lie à notre service de certification demandant un jeton de session, ce dernier commence par demander l'UID assigné à cette application particulière sur le système. Nous tenons à souligner que seuls les services sont en mesure de demander une telle information au système. L'UID est ensuite utilisé pour obtenir le nom du package qui sera utilisé comme entrée pour les contrôles de sécurité. Ces contrôles sont effectués selon l'ordre suivant :

1) Vérification de l'identité de l'application

Durant cette vérification, le service a recours à la politique basée sur l'identité des applications stockée sous forme de fichiers de base de données. Le service vérifie si le nom du package d'application est reconnu par la politique et si oui, à quelle liste l'application appartient. Trois cas sont possibles :

- Le nom du package de l'application figure dans la liste blanche : le service vérifie la dernière date de mise à jour de l'application en question. Si la date stockée dans la base de données et la date obtenue du système concordent, alors un jeton de session valide est accordé, l'application et les contrôles de sécurité sont interrompus. Dans le cas contraire, le service passe au contrôle de sécurité suivant.
- Le nom du paquet appartient à la liste noire de l'application : L'application obtient le jeton 7700, ce qui indique que l'application est reconnue comme application malveillante par le service de certification, et donc par notre application.
- Aucune entrée ne correspond au nom du package: cela signifie que l'application n'est pas encore reconnue par notre application. Le service passe au deuxième contrôle

```
/*
 *
 * Méthode qui retourne le nom du package de l'application appelante
 *
 */
public void initializeCheck(){
    callerUid = Binder.getCallingUid();
    pm = getPackageManager();
    packageName= pm.getNameForUid(callerUid);

}
```

Figure 7.8 Obtention du nom du package de l'application appelante

2) Vérification basée sur l'identité du développeur

Cette vérification met en œuvre la politique basée sur l'identité des développeurs d'applications. Le service commence par interroger Android sur la clé publique du développeur de l'application appelante en se basant sur son nom de package obtenu lors de l'étape précédente.

```
try {
    info = pm.getPackageInfo(packageName, PackageManager.GET_SIGNATURES);
} catch (NameNotFoundException e) {
    // TODO Auto-generated catch block
```

Figure 7.9 Obtention de la clé publique du développeur d'une application

Ensuite, le service de certification compare la clé publique obtenue avec celles qui figurent dans la liste noire. Ainsi, deux cas se présentent :

- La clé publique figure dans la liste noire ce qui signifie que le développeur de l'application est reconnu comme un développeur malveillant par la politique basée sur l'identité des développeurs. L'application appelante obtient donc le jeton de session 7600.
- La clé publique ne figure pas dans la liste noire ce qui indique que le développeur de l'application est inconnu pour notre application. Par conséquent, nous procédons à la troisième vérification.

3) Vérification basée sur les permissions

Si le service de certification se rend à cette étape de la vérification, cela signifie que soit l'application appelante n'est pas reconnue par la politique basée sur l'identité des applications et celle basée sur l'identité des développeurs, soit elle a été mise à jour depuis sa dernière interaction avec notre application. Pour le dernier cas, nous considérons qu'il est plus sûr de revérifier les permissions détenues par une application appelante suite à sa mise-à-jour. En effet, un développeur malveillant peut publier une première version de son application qui serait considérée comme légitime par notre application pour ensuite la mettre à jour avec plus de permissions que notre application le permet afin de la tromper.

```
try {
    info = pm.getPackageInfo(packageName, PackageManager.GET_PERMISSIONS);
} catch (NameNotFoundException e) {
    // TODO Auto-generated catch block

}
```

Figure 7.10 Obtention de la liste de permissions de l'application appelante

Durant la vérification des permissions, le service de certification analyse l'ensemble des permissions détenues par une application donnée en se référant à la politique basée sur les permissions. Le service demande la liste des permissions obtenues par l'application appelante en se basant sur son nom de package. Pour chaque permission obtenue, le service consulte la base de données contenant la politique et vérifie l'existence d'une règle liée à cette permission (règles « Doit avoir » et « Ne Doit Pas Avoir »).

À la fin de la vérification, deux cas se présentent :

- La combinaison de permissions enfreint une règle dans la politique : le nom du package de l'application est alors ajouté à la liste noire dans la politique basée sur les identités des applications et le service attribue le jeton 7500 à l'application. À l'avenir, toutes les demandes de jeton de session provenant de cette application seront rejetées par la première vérification, qui est celle de l'identité de l'application, puisque désormais son nom figure sur la liste noire. Il est aussi possible d'ajouter la clé publique du développeur de l'application en question à la liste noire dans la politique basée sur l'identité des développeurs, afin d'empêcher toute application provenant de même développeur d'interagir avec notre application.

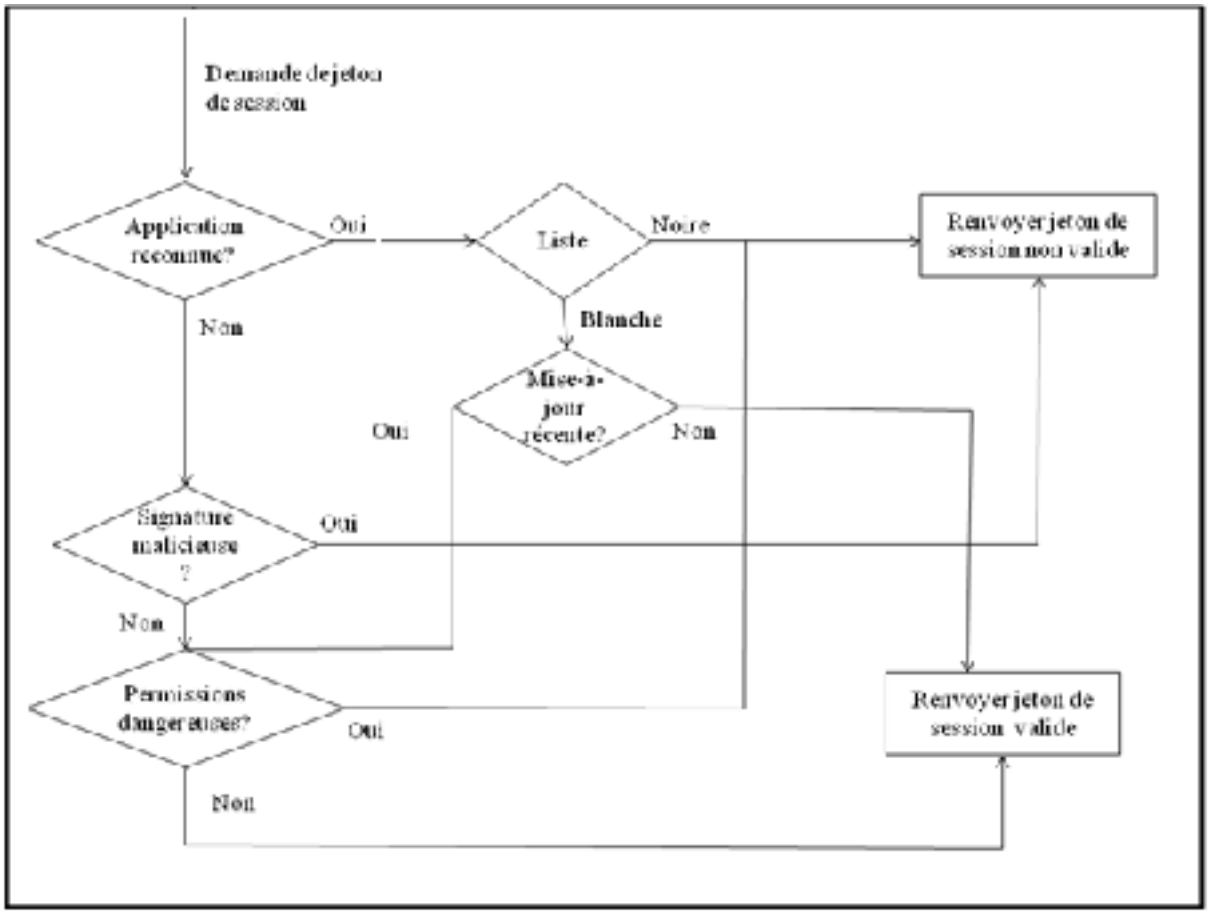

Figure 7.11 Arbre de décision utilisé par le service de certification

- La combinaison de permission respecte les règles de politique : le nom du package de l'application est alors ajouté à liste blanche des applications. Un jeton de session valide est envoyé à l'application appelante. Le jeton de session ainsi que la date de la dernière mise à jour de l'application sont aussi stockés dans la politique.

La logique utilisée par le service de certification et les différentes vérifications de sécurité est illustrée dans la figure 7.11.

7.3.3.2 L’authentificateur

Une application externe doit chaque fois se lier au service de certification que nous introduisant afin d'obtenir un jeton de session, et ce, avant de commencer à interagir avec n'importe quel composant protégé de notre application. Un composant protégé est celui qui a besoin de vérifier l'identité de l’application qui l’appelle.

En fait, un développeur peut vouloir laisser certains composants accessibles à toutes les applications installées sur le téléphone Android sans utiliser le mécanisme de jeton de session. Toutefois, seuls les composants qui gèrent des informations non sensibles et effectuant des tâches non critiques devraient être laissés sans protection de sécurité.

Les applications qui réussissent à passer les contrôles de sécurité effectués par notre service de certification reçoivent un jeton de session valide, autrement elles reçoivent un jeton non valide (7700,7600 ou 7500). À chaque fois qu’une application envoie une intention à un composant protégée appartenant à notre application (activité, service ou récepteur de diffusion), le jeton de session est ajouté au champ de données de l'intention comme un supplément via la méthode *putExtra(Session_token , valeur_jeton_de_session)* de la classe *Intent*.

À la réception d’une intention, le composant protégé de notre application appelle le module d’authentification et lui passe l'intention comme paramètre. Premièrement, l'authentificateur vérifie si l'intention possède un champ de donnée supplémentaire appelé *session_token*. À l’absence du champ *session_token* dans l'intention, une exception de sécurité est levée pour informer l’application appelante de la nécessité d’avoir un jeton de session pouvant être obtenu de notre service de certification. En revanche, si un jeton de session est trouvé dans l'intention, plusieurs cas sont possibles :

- La valeur du jeton de session est égale à 7700 : Une exception de sécurité est levée informant l’application appelante qu’elle est reconnue malveillante par la notre. Les

messages contenus dans l'exception peuvent aider le développeur de l'application appelante à mieux comprendre la cause de l'exception de sécurité et modifier son application en fonction du message obtenu.

- La valeur du jeton de session est égale à 7600 : Une exception de sécurité est levée informant l'application appelante que la clé publique avec laquelle elle a été signée figure dans la liste noire, et elle est donc rejetée.
- La valeur du jeton de session est égale à 7500 : Une exception de sécurité est levée informant l'application appelante que sa combinaison de permissions obtenue est jugée non adéquate par notre application.
- La valeur du jeton de session est différente de 7700, 7600 et 7500 : l'authentificateur consulte la politique basée sur l'identité et détermine si une règle liée à la valeur du jeton existe dans la politique. Nous rappelons que le jeton de session est stocké avec le nom du package si l'application appartient à la liste blanche.

Suite à cette vérification, l'authentificateur renvoie le résultat de l'authentification au composant qui l'a utilisé. Si la valeur du jeton de session présentée par l'application appelante figure dans la politique basée sur les identités, alors l'application est authentifiée et le jeton qui lui a été associé est remis à zéro afin d'empêcher les attaques de rejeu du jeton de session. Dans le cas échéant, l'interaction avec l'application est rejetée. Ce dernier cas signifie qu'une application malveillante a forgé un jeton de session dans le but de tromper notre application.

7.3.3.3 Vérification des contraintes d'utilisation

Le dernier contrôle de sécurité que nous présentons vise à éviter que notre application soit utilisée d'une manière malveillante. En fait, même si une application malveillante parvient à contourner les contrôles de sécurité, elle ne sera pas en mesure d'abuser des fonctionnalités offertes par notre application indéfiniment.

Le vérificateur de contraintes d'utilisation extrait l'action contenue dans l'intention et consulte la politique basée sur les actions. Si une règle pour l'action donnée existe, le vérificateur vérifie si l'opération liée à l'action peut être utilisée par l'application appelante. Dans notre implémentation actuelle, nous nous limitons à la vérification des contraintes temporelles. La vérification se déroule sur deux étapes.

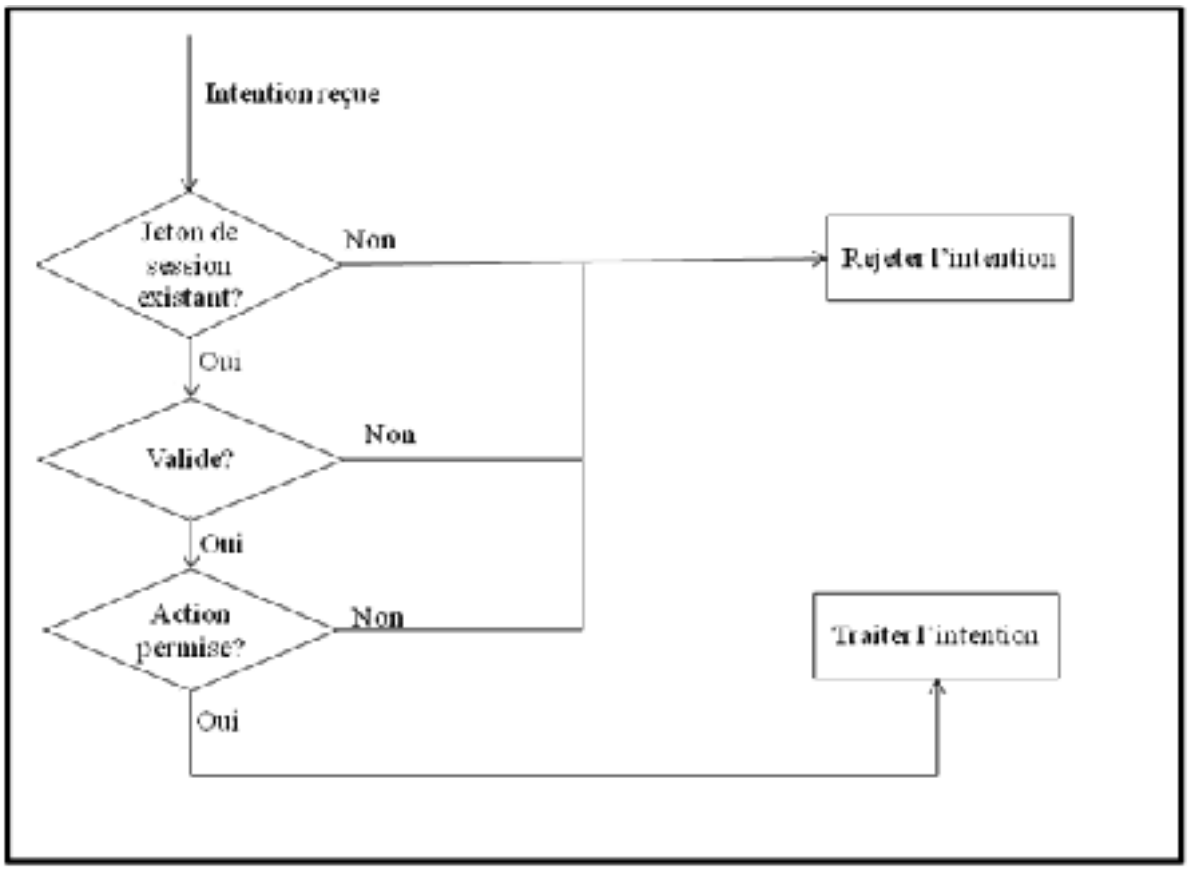

Figure 7.12 Arbre de décision utilisé au moment de la réception d'une intention

Durant la première étape, le vérificateur vérifie si le nombre d'usage maximum pour l'action contenue dans l'action a été atteint. Si c'est le cas, le vérificateur rejette l'intention. Dans le cas où la politique permet encore l'usage de l'action spécifiée, on passe à la deuxième étape qui consiste à vérifier si l'heure actuelle (obtenue à partir du système) permet l'usage de cette action. Le fonctionnement de l'authentificateur est montré dans la figure 7.12.

Dans la figure 7.13, nous illustrons un exemple le fonctionnement du service de certification et de l'utilisation de l'authentificateur.

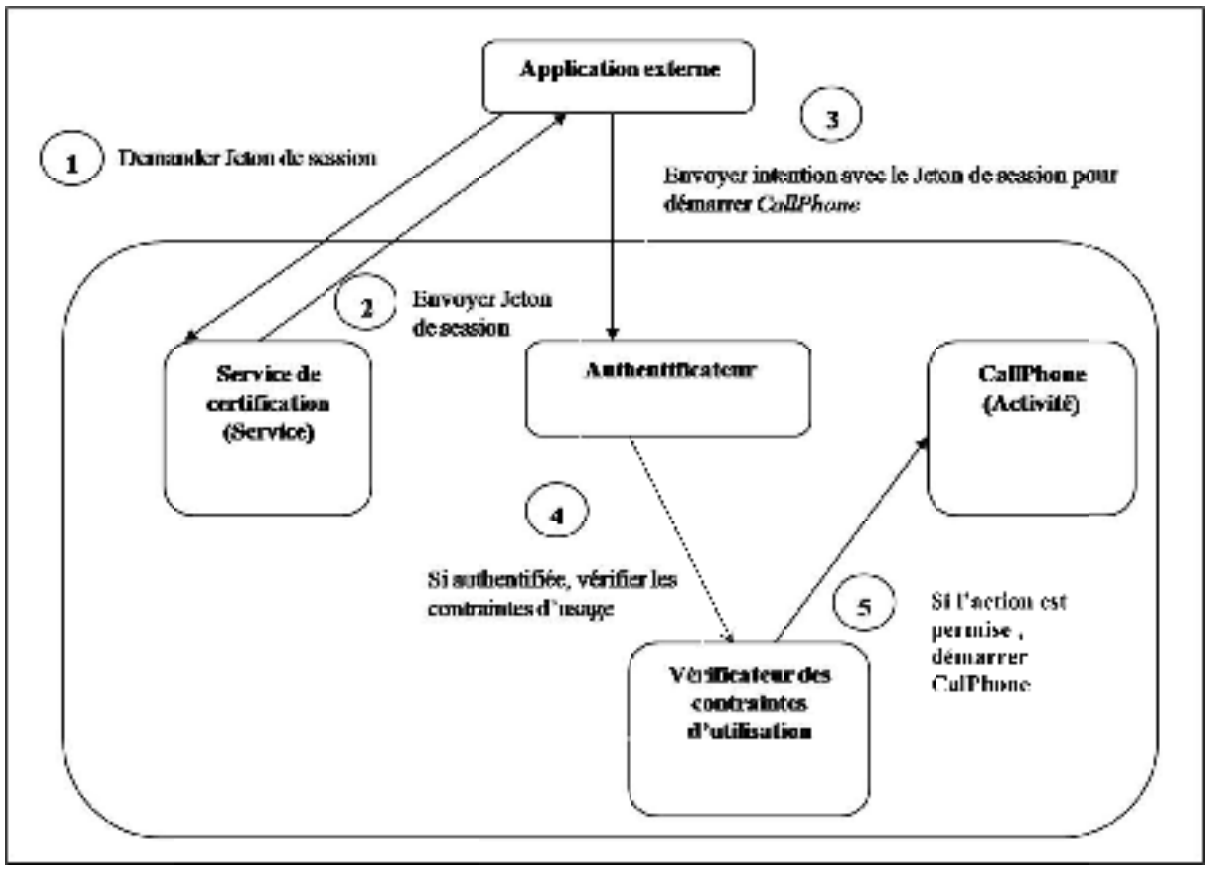

Figure 7.13 Exemple de déroulement des contrôles de sécurité

7.3.3.4 Application des politiques de sécurité lors de l'envoi d'une intention

Jusqu'ici, nous avons discuté l'application des politiques de sécurité au moment de la réception d'intentions provenant d'applications internes. Dans cette section, nous expliquons notre méthode pour sécuriser l'envoi des données contenues dans les intentions envoyées par l'application devant être protégées contre les attaques visant les intentions.

Comme nous avons vu dans la section 4.2.4.1, les intentions implicites (qui ne spécifient pas de destinataire) peuvent être détournées par les applications malveillantes cherchant à

exploiter les informations sensibles contenues dans celles-ci. Cependant, les développeurs se trouvent souvent contraints à envoyer des intentions implicites surtout dans le cas où ils ne connaissent pas les noms des applications capables de gérer leurs intentions.

Par exemple, si une application désire effectuer un payement en ligne, le développeur ne peut cibler une application précise, mais au lieu, il laisse le choix à Android ou l'utilisateur pour choisir l'application la mieux adaptée à ce genre d'action. Cependant, rien ne peut garantir que l'application choisie par le système ou l'utilisateur soit fiable.

L'objectif de la méthode que nous proposons ici consiste à éviter l'envoi d'intentions implicites. Au moment de l'envoi d'une intention, notre application demande au système Android de fournir une liste des applications capables de gérer l'intention au point d'être envoyées. Pour ce faire, trois méthodes différentes sont offertes aux développeurs des applications dépendamment du cas d'utilisations :

- *queryIntentActivities(intention_à_envoyer)* : si l'intention à envoyer sera utilisée pour démarrer une activité à l'aide de la méthode *startActivitiy(intention_à_envoyer).*
- *queryIntentServices(intention_à_envoyer)* : si l'intention à envoyer sera utilisée pour démarrer un service à l'aide de la méthode *startService(intention_à_envoyer).*
- *queryIntentBroadcastReceivers(intention_à_envoyer)* : si l'intention sera diffusée avec la méthode *sendBroadcast(intention_à_envoyer).*

Les trois méthodes retournent une liste des composants des applications installées capables de gérer l'intention. Pour chaque composant candidat, nous demandons à Android de nous fournir le nom du package auquel le composant appartient. Pour chaque application candidate, nous procédons au contrôle de sécurité tel que nous l'avons décrit dans la section 7.4.2.1.

Suite à la vérification de l'identité de l'application, l'identité du développeur et les permissions, trois cas se présentent :

- Aucune application n’est jugée légitime. Dans ce cas, l’application affiche un message à l’utilisateur pour l’informer que l’action demandée ne peut être traitée, et ce, pour des raisons de sécurité.

- Une seule application est légitime. Dans ce cas, le nom de cette application est spécifié comme destinataire de l’intention à envoyer. De cette façon, on évite que l’intention soit redirigée vers d’autres destinataires que celui spécifié dans l’intention.
- Plusieurs applications sont légitimes. Dans ce cas, un message s’affiche à l’utilisateur lui demandant de choisir parmi les applications candidates. Le nom de l’application choisie est alors spécifié comme étant le destinataire de l’intention.

7.3.4 Analyse des méthodes proposées

Nous estimons que le service de certification lorsqu’il est intégré dans une application peut limiter considérablement les impacts des applications malveillantes déclarées que nous avons décrits dans la section 4.1. Une application malveillante déclarée demandant une combinaison dangereuse de permissions ne sera pas en mesure d'exploiter notre application puisque la combinaison est examinée par le service de certification et sera rejetée. Même si d'autres applications vulnérables existent sur le téléphone Android, le service de certification couplé au modèle d'authentification évite à notre application d'être un vecteur d'attaque supplémentaire pour les applications malveillantes.

Lors de l'envoi d'intentions, Android offre la possibilité de protéger les intentions implicites avec des permissions d'accès. Toutefois, une application malveillante peut facilement obtenir ces permissions lors de l'installation et être en mesure de voler les intentions et accéder aux informations sensibles qu'elles peuvent contenir. La méthode d’envoi d’intentions présentées dans la section 7.4.2.4 permet d’éviter d'envoyer des intentions implicites pouvant être détournées (Attaque décrite dans la section 4.2.4.3).

Cependant, nous constatons que le modèle d'authentification présenté pourrait être vulnérable à des attaques de l'homme au milieu (OWASP, 2004b). En effet, au moment de l'envoi d'une intention vers tout composant protégé de notre application, une application externe annexe le jeton de session à l'intention. Une application malveillante pourrait s'inscrire dans le système pour recevoir la même intention et être capable de voler le jeton de session. Ensuite, elle peut réutiliser le même jeton afin de tromper notre composant qui va la considérer comme une application légitime.

Toutefois, l'impact de cette attaque est limité dû au fait que le jeton de session est renouvelé avant chaque interaction. De plus, notre application impose des contraintes d'utilisation sur les fonctionnalités sensibles qu'elle expose. Par conséquent, même si l'attaquant réussit à tromper notre application en envoyant un jeton volé, il ne sera pas capable de l'exploiter de façon illimitée.

Afin de contrer cette attaque, nous présentons une deuxième implémentation de nos méthodes de sécurité utilisant le concept de signature digitale (Rivest, Shamir et Adleman, 1978). Ce genre de mécanisme ajoute une nouvelle couche de sécurité permettant de vérifier l'authenticité de l'application ayant envoyé une intention à notre application, mais aussi de vérifier l'intégrité des données contenues dans l'intention. Avec l'utilisation du système cryptographique RSA, les vérifications de sécurité se déroulent comme suit :

- Une application externe désirant interagir avec un composant protégé de notre application commence par demander un jeton de session au service de certification tout en présentant sa clé publique.
- Le service de certification soumet l'application aux contrôles de sécurité que nous introduisons. Si l'application est considérée légitime, sa clé publique est stockée dans une base de données contenant les clés publiques de toutes les applications considérées légitimes par le service de certification.

- L'application appelante est sur le point d'envoyer une intention à notre application. Elle utilise sa clé privée pour signer le jeton de session qu'elle vient d'obtenir. La signature est ajoutée comme une donnée dans l'intention ainsi que la valeur du jeton en clair. Cependant, il est recommandé de conserver la clé privée de l'application dans un fichier protégé par les droits d'accès adéquats tel que décrit dans la section 7.2.3. Dans le cas où le téléphone est rooté, il est toujours possible pour un attaquant de voler la clé privée de l'application et l'utiliser pour usurper l'identité d'une application légitime. Nous rappelons que les méthodes présentées dans le cadre de ce travail se limitent à la protection des applications installées sur les téléphones Android non rootés.
- De même, toute autre donnée contenue dans l'application doit être signée par la clé privée de l'application appelante. Les signatures correspondant à chacune des données signées sont annexées à l'intention.
- Une fois l'intention envoyée vers notre application, le composant destinataire reçoit l'intention et fait appel à l'authentificateur.
- L'authentificateur extrait la valeur du jeton de session contenu dans l'intention et l'utilise afin d'obtenir la clé publique de l'application appelante.
- La clé publique obtenue est utilisée afin de vérifier la signature du jeton de session. Si la signature est vérifiée avec succès, cela signifie que le jeton de session envoyé par notre service de certification a été bel et bien reçu par l'application légitime qui l'a demandé et qu'il n'a pas été détourné par une application malicieuse.
- L'étape suivante consiste à vérifier la signature de chacune des données contenues dans l'intention (p. ex., numéro de téléphone à appeler). L'échec de vérification d'une signature signifie que l'intention a été détournée par une application malveillante qui a changé les valeurs contenues dans l'intention puis l'a renvoyé vers notre application (Attaque détaillée dans la section 4.2.4.3). L'intention est alors rejetée.

Dans la suite nous présentons un exemple d'utilisation du système cryptographique RSA en utilisant la méthode décrite ci-dessus. Considérons le cas où notre application expose une activité appelée *Send_SMS* qui peut être utilisée par les applications externes afin d'envoyer des SMS. Une application « A » voulant utiliser ce composant commence par demander un

jeton de session à notre service de certification et présente sa clé publique. Suite aux contrôles de sécurité, l'application « A » est considérée comme une application légitime et obtient un jeton de session valide. Le nom du package de l'application A est stocké dans une base de données avec sa clé publique ainsi que le jeton de session qu'elle vient d'obtenir.

Lorsque l'application « *A* » reçoit le jeton de session, elle le signe avec sa clé privée et annexe la signature obtenue dans le champ de données de l'intention. De même, l'application A signe le contenu du SMS qu'elle désire envoyer ainsi que le numéro du destinataire du SMS et les ajoute à l'intention. Un développeur malicieux cherchant à exploiter l'activité *SendSMS* de notre application afin d'envoyer des SMS vers des numéros surtaxés, développe une application B malveillante. L'application B est conçue pour détourner les intentions envoyées par l'application A (et par toute autre application légitime) vers notre application. Dans notre première implémentation, l'application B serait capable d'obtenir le jeton de session et l'utiliser pour pousser notre application à envoyer un SMS avec le contenu et vers le numéro que l'application malveillante choisit puisque le jeton de session utilisé par B est considéré valide. Cependant, avec l'utilisation de RSA, même si B réussit à obtenir l'intention, toute modification touchant le contenu du SMS à envoyer ou le numéro du destinataire serait détectée et entraînerait le rejet de l'intention.

Toutefois, l'utilisation du système cryptographique RSA s'est avérée coûteuse engendrant un délai d'attente supplémentaire lors de la réception d'une intention en raison des ressources de calcul limitées des téléphones Android. En effet, les opérations de signature et de vérification des signatures augmentent la charge de calcul considérablement. De plus, l'utilisation de RSA pourrait compliquer la tâche aux développeurs souhaitant utiliser les services offerts par notre application dans les leurs. Cependant, nous recommandons fortement l'utilisation du mécanisme des signatures RSA pour les applications offrant des services critiques pouvant être utilisés par les applications installées sur le téléphone Android tel que les applications de payement en ligne.

CONCLUSION

Android est le système d'exploitation mobile le plus répandu dans le monde et continue de gagner en popularité parmi les utilisateurs et les développeurs des applications mobiles. Cependant, la question de sécurité d'Android reste jusque-là un problème qui a besoin de solutions.

Dans le cadre de ce travail, nous nous sommes intéressés à proposer des mécanismes de sécurité capables d'améliorer la sécurité des applications Android. Ces mécanismes de sécurité ont l'avantage de pouvoir être intégrés au cœur des applications Android, et n'exigent aucune modification de la plateforme actuelle d'Android. Notre classification nous a montré que les applications Android malveillantes varient en complexité, allant des applications profitant des vulnérabilités difficilement exploitables. Nous estimons que le nombre contre le système Android va continuer à s'accroître dans le futur et ils auront tendance à devenir de plus en plus sophistiqués.

Notre travail a débuté par l'analyse des vulnérabilités du système Android ce qui nous a menés à établir une nouvelle classification des applications Android malveillantes. Cette classification prend en considération les vulnérabilités exploitées et nous a permis de comprendre les différents modes opératoires des applications malveillantes. La classification révèle également qu'un grand nombre d'utilisateurs des téléphones Android contribue de façon involontaire à la réussite des attaques visant Android par la négligence des mesures de sécurité offertes par la plateforme (lors de l'installation des applications) ou encore par la volonté de certains d'entre eux à augmenter le degré de contrôle sur leurs téléphones Android au détriment de la sécurité (le rooting).

Ensuite nous avons fait une revue critique des diverses solutions de sécurité d'Anroid proposées dans la littérature scientifique. En effet, plusieurs solutions proposées pourraient grandement améliorer le modèle de sécurité utilisé par Android et limiter les impacts des applications malveillantes. Cependant, Google s'abstient à ce jour d'intégrer aucune de ces

solutions au cœur de la plateforme Android. Nous pensons que ce fait pourrait être expliqué par les divers changements que ces solutions apportent à la plateforme Android. En effet, tout changement effectué sur le système Android impliquerait nécessairement des changements du Kit de développement SDK et d'éventuelles formations pour les développeurs d'applications, d'autant plus que ces changements doivent être adoptés par les divers fournisseurs de service et les constructeurs des téléphones mobiles.

Motivés par ce fait, nous nous sommes intéressés à concevoir des méthodes sécurisées pour le développement d'applications Android qui seraient adaptées au système Android actuel sans exiger des modifications à celui-ci. Notre objectif consistait principalement à améliorer le niveau de protection des applications légitimes contre celles qui sont malveillantes. Cependant, notre travail s'est limité à la protection des applications légitimes installées sur des téléphones Android standards (qui ne sont pas rootés).

Afin de réaliser notre objectif, nous avons conçu une nouvelle méthode de certifications des applications Android ainsi qu'une méthode d'authentification avec la possibilité de définir quatre types de politique de sécurité qui sont appliquées lors de l'exécution de l'application. Ces politiques de sécurité régissent les interactions entre les différentes applications offrant à celles-ci un contrôle d'accès de fine granularité. Ces politiques peuvent être définies par les développeurs d'applications ainsi que par les utilisateurs désirant définir leurs propres politiques de sécurité. Une interface de gestion de politique facile à utiliser a été mise en place afin de faciliter la gestion de ces politiques par les utilisateurs des applications, lorsqu'elles sont utilisées par les développeurs des applications, ces politiques augmentent le degré de contrôle sur la façon avec laquelle elles sont utilisées par les applications externes.

Notre méthode de certification permet à une application de juger le degré de fiabilité des applications qui l'appellent en se basant sur l'ensemble de politiques en place. Cette méthode utilise le principe de jeton de session qui une fois émit, il sert à authentifier une application externe face à l'application protégée par notre méthode aussi, nous avons introduit des méthodes afin de restreindre l'accès aux fonctionnalités offertes par une application légitime

afin de limiter l'impact des applications malveillantes exploitant les applications légitimes. Avec cette méthode, une application envoyant des SMS à la demande des applications externe peut limiter le nombre de SMS qu'elle peut envoyer par jour, restreindre la liste des numéros des destinataires et imposer une contrainte liée à une plage horaire déterminée durant laquelle elle peut effectuer l'envoi de SMS.

Dans l'implémentation actuelle de notre solution, nous avons utilisé un générateur de jetons de session simple capable de générer des valeurs aléatoires allant de 0 à 10000000. Nous estimons qu'il est possible d'améliorer notre méthode en utilisant un générateur de nombres pseudo-aléatoires sécuritaire connu tel que YARRO (Schneier, Kelsey et Ferguson, 2000).

D'autre part, nous comptons regrouper les méthodes proposées sous forme d'un plug-in intégré dans l'environnement de développement intégré (IDE) Eclipse (The Eclipse Foundation, 2001) utilisé par la grande majorité des développeurs Android. Actuellement, Andoid fournit le plug-in ADT (*Android Developpement Tool*) pour l'IDE Eclipse. Le plug-in ADT regroupe plusieurs outils facilitant plusieurs tâches telles que la conception des interfaces graphiques des applications, la définition du fichier Manifest des applications (AOSP, 2008a). Nous estimons que notre travail peut être intégré dans le plug-in ADT offrant ainsi aux développeurs des applications Android la facilité de définir les différentes politiques de sécurité pour leurs applications via une interface graphique simple et ergonomique. Le plug-in serait en charge de générer le code java nécessaire pour la définition et l'application des politiques de sécurité. L'objectif du plug-in est de favoriser le déploiement des méthodes proposées dans le cadre de ce travail de recherche et leur utilisation à grande échelle par les développeurs d'applications Android.

Le domaine de la sécurité des téléphones mobiles ne cesse de s'élargir. Ce mémoire ne présente qu'une fine partie d'un écosystème plus large. Avec la croissance actuelle des systèmes d'exploitation mobiles et l'émergence de nouveaux produits, la sécurité demeurerait l'un des défis les plus importants à soulever et le plus déterminants pour l'avenir de la technologie des téléphones intelligents.

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

AOSP. 2008a. « ADT Plugin for Eclipse ». < http://developer.android.com/sdk/eclipse-adt.html >.

AOSP. 2008b. « Dev guide ». In *Android Developers*. En ligne. < http://developer.android.com/guide/basics/what-is-android.html >. Consulté le 12 mai 2011.

AOSP. 2008c. « Dev Guide - <activity> ». In *Android developpers*. En ligne. < http://developer.android.com/guide/topics/manifest/activity-element.html#exported >. Consulté le 21 juin 2011.

AOSP. 2008d. « Dev guide - Content Providers ». In *Android Developers*. En ligne. < http://developer.android.com/guide/topics/providers/content-providers.html >. Consulté le 12 mai 2011.

AOSP. 2008e. « Dev Guide - The AndroidManifest.xml File ». In *Android developpers*. En ligne. < http://developer.android.com/guide/topics/manifest/activity-element.html#exported >. Consulté le 21 juin 2011.

AOSP. 2008f. « Reference ». In *Android developpers*. En ligne. < http://developer.android.com/reference/android/content/Context.html#MODE_WORLD_READABLE >. Consulté le 23 Septembre 2011.

AOSP. 2008g. « Security and Permissions ». In *Android developpers*. En ligne. < http://developer.android.com/guide/topics/security/security.html#uri >. Consulté le 13 octobre 2011.

Asaf, Shabtai, Fledel Yuval, Kanonov Uri, Elovici Yuval, Dolev Shlomi et Glezer Chanan. 2010. « Google Android: A Comprehensive Security Assessment ». *Security & Privacy, IEEE,* vol. 8, p. 35-44.

Bläsing, T., L. Batyuk, A. D. Schmidt, S. A. Camtepe et S. Albayrak. 2010. « An Android Application Sandbox system for suspicious software detection ». In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*. (19-20 Oct. 2010), p. 55-62.

Bornstein, Dan. 2008. *Dalvik virtual machine internals*. En ligne. Video youtube, 60 min. < https://sites.google.com/site/io/dalvik-vm-internals >.

Burguera, Iker, Urko Zurutuza et Simin Nadjm-Tehrani. 2011. « Crowdroid: behavior-based malware detection system for Android ». In *Proceedings of the 1st ACM workshop on*

Security and privacy in smartphones and mobile devices. (Chicago, Illinois, USA), p. 15-26.

Case, Justin. 2011. « News ». In *AndroidPolice*. En ligne. < http://www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/ >. Consulté le 13 Mai 2011.

Cheng, Jacqui 2011. « Gadgets - News ». In *http://arstechnica.com*. En ligne. < http://arstechnica.com/gadgets/news/2011/01/android-beats-nokia-apple-rim-in-2010-but-firm-warns-about-2011.ars >. Consulté le 6 mai 2011.

Chin, Erika, Adrienne Felt, Kate Greenwood et David Wagner. 2011. « Analyzing inter-application communication in {Android} ». In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. (Bethesda, MD, USA), p. 239-252. ACM.

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. Springer-Verlag.

Davi, Lucas, Alexandra Dmitrienko, Ahmad-Reza Sadeghi et Marcel Winandy. 2011. « Privilege Escalation Attacks on Android
Information Security ». In, sous la dir. de Burmester, Mike, Gene Tsudik, Spyros Magliveras et Ivana Ilic. Vol. 6531, p. 346-360. Coll. « Lecture Notes in Computer Science »: Springer Berlin / Heidelberg.

Enck, W., M. Ongtang et P. McDaniel. 2009a. « Understanding Android Security ». *Security & Privacy, IEEE,* vol. 7, p. 50-57.

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.

Enck, William, Machigar Ongtang et Patrick McDaniel. 2009b. « On lightweight mobile phone application certification ». In *Proceedings of the 16th ACM conference on Computer and communications security*. (Chicago, Illinois, USA), p. 235-245.

F-Secure. « Mobile Security ». In *F-Secure*. < http://www.f-secure.com/en/web/home_us/protection/mobile-security/overview >. Consulté le 23 novembre 2011.

Feily, M., A. Shahrestani et S. Ramadass. 2009. « A Survey of Botnet and Botnet Detection ». In *Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09. Third International Conference on*. (18-23 June 2009), p. 268-273.

Felt, Adrienne, Matthew Finifter, Erika Chin, Steve Hanna et David Wagner. 2011. « A Survey of Mobile Malware In The Wild ». In *2011 ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. (Chicago 17-21 Oct. 2011).

FortiGuard. 2010. In *Fortinet.com*. En ligne. < http://www.fortiguard.com/av/VID2100835 >. Consulté le 5 juin 2011.

Gollmann, D. 2006. *Computer security*. Wiley.

Harada, T. , T. Horie et K. Tanaka. 2004. « Task Oriented Management Obviates Your Onus on Linux ». In *Linux Conference 2004*. (Tokyo, Japan).

Hayashi, Kaoru 2011. « Threats and Risks ». In *symantec.com*. En ligne. < http://www.symantec.com/security_response/writeup.jsp?docid=2011-052310-1322-99&tabid=2 >. Consulté le 05 Septembre 2011.

Jacob, Grégoire, Hervé Debar et Eric Filiol. 2008. « Behavioral detection of malware: from a survey towards an established taxonomy ». *Journal in Computer Virology,* vol. 4, p. 251-266.

Loscocco, Peter, et Stephen Smalley. 2001. « Integrating Flexible Support for Security Policies into the Linux Operating System ». In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. p. 29-42.

Moser, A., C. Kruegel et E. Kirda. 2007. « Limits of Static Analysis for Malware Detection ». In *Proceedings of the Twenty-Third Annual Computer Security Applications Conference*. (10-14 Décembre. 2007), p. 421-430.

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.

Oberheide, jon. 2010. « Remote Kill and Install on Google Android ». En ligne. < http://jon.oberheide.org/blog/2010/06/25/remote-kill-and-install-on-google-android/ >.

Ongtang, Machigar, Stephen McLaughlin, William Enck et Patrick McDaniel. 2009. « Semantically rich application-centric security in android ». In *Proceedings of the 25th Annual Computer Conference Security Applications, ACSAC 2009*. (Honolulu, HI, United states, 7-11 Décembre 2009), p. 340-349.

OWASP. 2004a. <https://www.owasp.org/index.php/SQL_Injection >.

OWASP. 2004b. < https://www.owasp.org/index.php/Man-in-the-middle_attack >.

Perez, Sarah. 2010. « Archives ». In *ReadWriteWeb*. En ligne. < http://www.readwriteweb.com/archives/tap_snake_game_in_android_market_is_actually_spy_app.php >. Consulté le 17 Avril 2011.

Quette, Arnaud 2011. « Community Documentation ». In *help.Ubuntu.com*. En ligne. < https://help.ubuntu.com/community/AppArmor >. Consulté le 4 Novembre 2011.

Rieck, Konrad, Thorsten Holz, Carsten Willems, Patrick Düssel et Pavel Laskov. 2008. « Learning and Classification of Malware Behavior ». In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. (Paris, France), p. 108-125.

Rivest, R. L., A. Shamir et L. Adleman. 1978. « A method for obtaining digital signatures and public-key cryptosystems ». *Commun. ACM,* vol. 21, p. 120-126.

Robert, Paul. 2011. « Business Security ». In *Threatpost.com*. En ligne. < http://threatpost.com/en_us/blogs/android-nfc-bug-could-be-first-many-062011 >. Consulté le 17 Septembre 2011.

Schneier, Bruce, John Kelsey et Niels Ferguson. 2000a. « Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator ». In *Proceedings of the 6th Annual International Workshop on Selected Areas in Cryptography*. p. 13-33.

Shabtai, Asaf, Yuval Fledel et Yuval Elovici. 2010. « Securing Android-Powered Mobile Devices Using SELinux ». *Security & Privacy, IEEE,* vol. 8, p. 36-44.

Shabtai, Asaf , Yuval Fledel, Uri Kanonov, Yuval Elovici et Shlomi Dolev. 2009. « Google Android: A State-of-the-Art Review of Security Mechanisms ». *CoRR*. In *DBLP, http://dblp.uni-trier.de*.

Shabtai, Asaf, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev et Chanan Glezer. 2010. « Google Android: A Comprehensive Security Assessment ». *IEEE Security and Privacy,* vol. 8, p. 35-44.

Shabtai, Asaf, Uri Kanonov, Yuval Elovici, Chanan Glezer et Yael Weiss. 2011. « "Andromaly": a behavioral malware detection framework for android devices ». *Journal of Intelligent Information Systems*. 5 janvier 2011, p. 1-30.

Smith, Mat. 2011. « News ». In *Engadget Mobile*. En ligne. < http://www.engadget.com/2011/10/05/android-gingerbread-has-growth-spurt-grabs-38-2-percent-device/ >. Consulté le 11 octobre 2011.

SonicWall. 2011. « SonicWall Security Center ». In *SonicWall*. En ligne. < https://www.mysonicwall.com/SonicAlert/searchresults.aspx?ev=article&id=381 >. Consulté le 14 Octobre 2011.

Strazzere, Tim. 2011. « Blog ». In *Lookout Mobile Security*. En ligne. < http://blog.mylookout.com/2011/03/do-androids-dream%E2%80%A6/ >. Consulté le 5 septembre 2011.

Symantec. 2011. En ligne. < http://www.symantec.com/security_response/writeup.jsp?docid=2011-021514-4954-99&tabid=2 >.

The Eclipse Foundation. 2001. < http://www.eclipse.org/ >.

ThreatSolutions. 2011. « Weblog ». In *F-Secure.com*. En ligne. < http://www.f-secure.com/weblog/archives/00002171.html >. Consulté le 15 juin 2011.

Townsend, bridgette. 2010. « Integrationg GPS + ANDROID ». < http://kibilogic.com/wordpress/?p=73 >. Consulté le 22 juin 2011.

US-cert/NIST. 2009. « Vulnerabilities ». In *National Vulnerability Database*. En ligne. < http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-1185 >. Consulté le 5 juillet 2011.

Venkatesan, Dinesh. 2011. « Blog ». In *Total Defense*. En ligne. < http://totaldefense.com/securityblog/2011/08/29/Dynamic-Analysis-of-Golddream-A-Trojan.aspx >. Consulté le 5 Octobre 2011.

Wright, Chris , Crispin Cowan, James Morris, Stephen Smalley et Greg Kroah-Hartman. 2002. « Linux Security Module Framework ». In *Ottawa Linux Symposium*. (Ottawa, Canada, 26-29 juin 2002).