TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

API	Application Programming Interface
APK	Android Application Package
AOP	Aspect-Oriented Programming
APSL	Android Policy Specification Language
BNC	Banque National du Canada
CPU	Central Processing Unit
DVM	Dalvik Virtual Machine
GPS	Global Positioning System
IMEI	International Mobile Equipment Identity
IoT	Internet of Things
JSON	JavaScript Object Notation
OCR	Optical Character Recognition
OS	Operating System
SDK	Source Developing Kit
SMS	Short Message Service
TI	Technologie de L'Information
UMA	User-Managed Access
UI	User Interface
UID	Unique Identifier
XACML	Xtensible Access Control Markup Language
XML	Extensible Markup Language

LISTE DES SYMBOLES ET UNITÉS DE MESURE

UNITÉS DE TEMPS

h	heure
min	minute
s	seconde
ms	milliseconde
µs	microseconde

UNITÉS DE CAPACITÉ

bit	Bit (Unité de base)
O	Octet (1 octet = 8 bits)
Ko	Kilooctet (1 Ko = 2^{10} octets)
Mo	Mégaoctet (1 Mo = 2^{20} octets)
Go	Gigaoctet (1 Go = 2^{30} octets)

UNITÉS DE FRÉQUENCE

Hz	Hertz (Unité de base)
KHz	Kilohertz (1 KHz = 10^3 Hz)
MHz	Mégahertz (1 MHz = 10^6 Hz)
GHz	Gigahertz (1 GHz = 10^9 Hz)

INTRODUCTION

0.1 Mise en contexte

De nos jours, le Smartphone est devenu un accessoire essentiel dans la vie de l'Homme contemporain. Vu cette indispensabilité et l'augmentation de la concurrence entre les différents fabricants, les développeurs sont obligés de focaliser leurs efforts sur le développement des plateformes des Smartphones. Cette effervescence de développement, a fait que le marché est envahi par un nombre énorme d'applications mobiles offrant plusieurs avantages pour les utilisateurs (Auffray, Chiffres clés : les ventes de mobiles et de smartphones, 2017).

Ce pic de croissance des applications mobiles s'explique par deux facteurs principaux :

1. Une croissance des ventes des Smartphones durant les cinq dernières années (*Voir* Figure 0.1),
2. Un accès de téléchargement facile aux nombreuses applications sur le magasin d'Android.

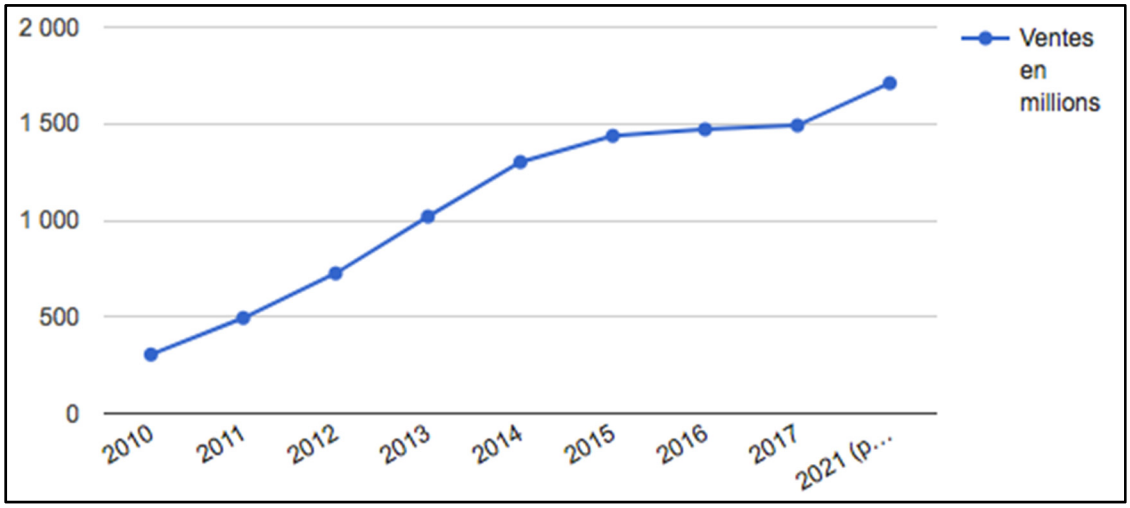


Figure 0.1 Ventes des smartphones de 2010 à 2017 (Auffray, 2017)

Dans un marché de mobile très compétitif, les entreprises devront agir en toute flexibilité pour répondre à la complexité des clients en développant des dizaines de milliers

d'applications. Ces applications doivent être disponibles sur les plateformes de téléchargement, tels que Google Play et App Store. Android est devenu le meneur des plateformes mobiles en dominant le marché avec une part estimé à 85,1% en 2017 (Auffray, Chiffres clés : les OS pour smartphones, 2017) (*Voir* Figure 0.2). Cette position de leadership s'est manifestée par l'augmentation significative des applications Android disponibles pour téléchargement et installation. En contrepartie, les taux d'attaques qui ciblent cette plateforme ont significativement augmenté. En effet, vers les 40% des applications existantes dans le store d'Android sont des malwares (Belkaab, 2017).

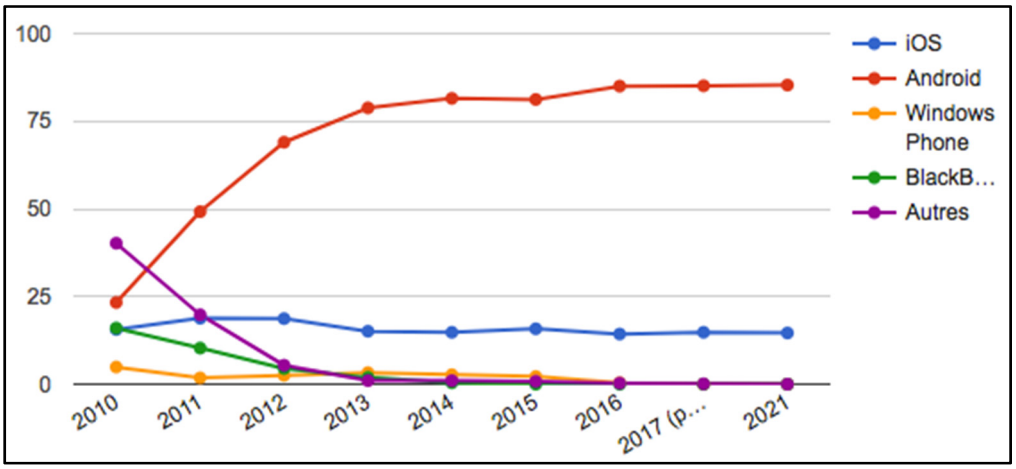


Figure 0.2 Part de marché mondial des OS mobiles (%)

Cet important flux d'attaques demeure un énorme risque affectant la confidentialité de plusieurs utilisateurs ainsi que l'économie des différentes organisations utilisant Android dans leurs infrastructures TI. C'est pourquoi la sécurité de cette plateforme est d'une nécessité importante.

Ce travail consiste à la proposition d'une solution visant à instrumenter les applications Android dans le but de fournir un système de surveillance assurant l'application d'un ensemble de politiques de sécurité. Celles-ci ne nécessitent pas l'intervention de l'utilisateur et assurent une exécution sécuritaire.

0.2 Problématique

Le fait de répondre aux besoins de ces utilisateurs n'est guère suffisant pour évaluer les applications, car elles peuvent également avoir un effet indésirable qui pourrait survenir sans le consentement de l'utilisateur, communément connu comme « la divulgation des données privées » (Enck, et al., 2010). Dans ce contexte, des études antérieures (Enck, et al., 2010), (Hornyack, Han, Jung, Schechter, & Wetherall, 2011), (Enck, Octeau, McDaniel, & Chaudhuri, 2011), (Wu, Zhou, Patel, Liang, & Jiang, 2014) ont révélé que de nombreuses applications, bénignes ou malveillantes, envoient des informations privées à des serveurs distants sans l'approbation des utilisateurs qui ne sont généralement pas conscients de la situation. De plus, le système d'autorisation de la plateforme Android actuel présente encore certaines limitations. Prenons l'exemple, le fait que les utilisateurs doivent accorder toutes les autorisations requises par une application pour l'installer, avec la possibilité de modifier par la suite seulement les autorisations qu'Android classifie comme dangereuses. Par conséquent, la prévention du risque de divulgation d'informations privées vers des utilisateurs non autorisés est une préoccupation majeure sur la plateforme et les applications Android actuelles.

Pour renforcer la sécurité sur la plateforme et les applications Android, plusieurs chercheurs ont proposé une variété de systèmes pour détecter, évaluer et atténuer les divulgations de confidentialité d'Android. Les approches existantes ont été examinées dans le présent rapport et sont classées en trois grandes catégories, à savoir :

La première catégorie porte sur les approches qui analysent le code source des applications Android d'une manière statique (Enck, et al., 2010), (Mann & Darmstadt, A Framework for Static Detection of Privacy Leaks in Android Applications, 2012) ou dynamique (Rasthofer, Arzt, & Lovat, DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android, 2014), (Yang, Li, Zhang, Li, Shu, & Gu, 2014), (Yang, Yang, Zhang, Gu, Gu, &

Wang, 2013), (Felt, Finifter, Chin, Hanna, & Wagner, 2011), (Rasthofer, Asrar, Huber, & Bodden, 2015), (Arena, Catania, & Torre, 2013) afin d'identifier toute activité malveillante avant ou durant l'exécution. Citons l'exemple de AppIntent (Yang, Yang, Zhang, Gu, Gu, & Wang, 2013) qui fournit un cadre pour détecter et analyser les divulgations de confidentialité par les deux manières : d'une manière statique qui consiste à analyser la possibilité des attaques sur les applications sans avoir besoin d'exécuter l'application ou d'une manière dynamique qui se fait par l'analyse des applications tout en accédant à leurs informations durant leur exécution. Toutefois, la plupart de ces approches n'arrivent à détecter que les attaques sur les flux de données intra-procédural. En outre, un pirate assez spécialisé dans le développement de logiciels malveillants arrive souvent à trouver des moyens pour outrepasser leur confinement.

Par exemple, les logiciels malveillants peuvent filtrer les informations privées à travers des canaux secondaires (tels que les flux implicites et les canaux de synchronisation). De plus, les techniques mentionnées ci-dessus ne peuvent pas traiter les composants natifs et les appels réflectifs dans Java d'une manière générale.

La deuxième catégorie comporte des solutions qui nécessitent la modification de la plateforme Android pour pouvoir appliquer des politiques de sécurité flexibles (Arena, Catania, & Torre, 2013), (Feth & Pretschner, 2012), (Hornyack, Han, Jung, Schechter, & Wetherall, 2011), (Zhou Y. , Zhang, Jiang, & Freeh, Taming information-stealing smartphone applications (on android), 2011), (Jeon, et al., 2012). Par exemple (Feth & Pretschner, 2012) ont proposé un système de sécurité pour améliorer les permissions standard du système d'exploitation Android. En outre, SecureDroid (Arena, Catania, & Torre, 2013) fournit une extension dans le cadre de sécurité d'Android pour appliquer des politiques contextuelles, spécifiées avec le langage XACML (Jeon, et al., 2012) pour que l'utilisation d'une ressource soit limitée à un nombre maximal de fois par jour .il peut aussi refuser l'accès à une ressource pendant quelques heures de la journée. Le plus important avantage de cette catégorie c'est qu'elle ne nécessite pas la génération de nouvelles politiques pour chaque nouvelle application installée. Toutes les politiques qui sont générés par le système seront

appliquées directement et facilement sur toutes les applications installées. Néanmoins, cette catégorie présente un certain nombre d'inconvénients majeurs tels que la nécessité de construire différentes versions de microprogrammes et de codes de plateforme. Dans ce cas, le langage de spécification des politiques de sécurité sera compilable et les applications seront limitées par les politiques de sécurité gérées par le système d'Android customisé.

Une troisième catégorie comprend plusieurs solutions qui se basent sur la réécriture d'applications Android (Xu, Saıdi, & Anderson, 2012), (Zhang & Yin, 2014). L'avantage de ces solutions c'est qu'elles ne nécessitent aucune modification de la plateforme Android. Ils peuvent être facilement déployées. Notamment, elles fournissent des mécanismes de contrôle d'accès qui surveillent les applications et appliquent les politiques sur les API sensibles. Par exemple (Zhang & Yin, 2014) étaient parmi plusieurs chercheurs qui ont développé une approche de réécriture du byte-code des applications Android en mettant en œuvre un prototype appelé Capper pour tracer et atténuer les failles de confidentialité dans ces applications. De plus, (Davis, Sanders, Khodaverdian, & Chen, 2012) ont conçu et implémenté un Framework de réécriture appelé I-arm-droid qui intégrait des moniteurs de référence In-App dans des applications Android. Cependant, cette approche présente aussi quelques limites, à savoir, la réécriture qui ne peut être appliquée que pour chaque application séparément.

Les trois catégories, mentionnées ci-dessus, ont fourni des cadres de sécurité qui peuvent appliquer des politiques de sécurités contextuelles, mais elles ne peuvent pas fournir des politiques de sécurité contextuelles inter-applications. Certains cadres permettent à de nombreuses parties de définir des stratégies de sécurité, mais n'arrivent pas à satisfaire les besoins de tous les utilisateurs. Nous avons donc besoin d'un langage de spécification de politique qui permet aux utilisateurs réguliers ainsi qu'aux entreprises d'interpréter et d'appliquer d'une manière fluide et facile leurs règles complexes sur leurs applications mobiles Android. En outre, un nouveau cadre d'application de politiques inter-application est requis. Il comprendrait notre définition d'un premier langage de spécification des politiques

de sécurité native dédiée pour le système d'exploitation d'Android et compatible avec les langages de spécification des politiques existants comme XACML et UMA.
Durant cette étude, nous allons adopter la troisième catégorie qui consiste à réécrire les applications Android pour appliquer efficacement des différentes politiques de sécurité dépendant du contexte actuel à chaque application individuellement sans modifier le système Android. Ainsi, les applications modifiées seront obligées de communiquer avec notre moniteur d'applications centralisé afin d'éviter toute activité malveillante, sans que les utilisateurs en soient conscients. De plus, nous allons nous concentrer sur les applications qui sont non nécessairement malveillantes, dans certains cas, mais qui peuvent causer un danger extrême dans le cas du changement de contexte lors de leurs exécutions.

Enfin, cette proposition discutera une variété de solutions connexes, fournira une analyse, et soulignera certaines de leurs défaillances et limitations à envisager pour des futures améliorations.

0.3 Objectifs et réalisations

Cette recherche vise plusieurs objectifs, toutefois, l'objectif principal est d'élaborer un Framework de définition des politiques de sécurité dépendantes du contexte pour les applications Android. Nous nous proposons de réécrire ces applications pour communiquer avec notre moniteur d'applications centralisé afin d'avoir accès aux appels d'API sensibles. De plus, les utilisateurs auront la possibilité d'appliquer leurs propres politiques en utilisant notre langage de spécification des politiques de sécurité inventée et dédiée pour le système d'Android. Ce dernier est capable de communiquer avec d'autres langages existants. Cependant, il faut noter que fournir des politiques et des solutions à ces problèmes s'avère une tâche très difficile. À notre connaissance, les approches actuelles de la littérature ne sont pas entièrement satisfaisantes, car ces approches ne permettent pas un contrôle centralisé des données privés inter-application pendant leur exécution. Elles ne peuvent fournir qu'un contrôle indépendant pour chaque application unique. En conséquence, plusieurs sous objectifs sont manipulés dans notre présente étude, à citer :

- **Proposer un Framework de réécriture,** ce Framework permet d'instrumenter les applications à contrôler pour qu'elles soient capables de recevoir les décisions de contrôles des permissions avant d'accéder aux APIs sensibles. En outre, il appliquera de nombreuses stratégies de sécurité au moment de l'exécution des applications dans un contexte non fiable ou lorsque plusieurs applications partagent les mêmes ressources du périphérique, etc.
- **Développer un contrôleur d'applications centralisé,** cela permettra aux utilisateurs de :
    - Contrôler tous les appels API effectués par les applications installées sur le périphérique.
    - Définir des politiques de sécurité contextuelles précises en fonction de divers attributs de contexte, tels que l'heure, le lieu, la date, le partage de fonctionnalités avec d'autres applications, etc.
    - Importer des politiques existantes définies par d'autres langages de spécification des politiques et les appliquer par notre contrôleur.
    - Exporter des politiques de sécurité en adoptant le format de notre langage ou d'autres formats des langages des politiques comme XML, JSON, XACML etc. afin de les partager avec d'autres utilisateurs de notre contrôleur ou avec d'autres systèmes mobiles ou Cloud etc.
- **Définir un langage de spécification de politique**, les politiques de sécurité ne sont pas fixes et peuvent changer au fil du temps afin de s'adapter aux besoins des utilisateurs. Par conséquent, nous avons besoin d'un langage de politiques avec une interface utilisateur pour permettre aux utilisateurs réguliers et aux entreprises d'interpréter et d'appliquer leurs règles complexes sur leurs applications mobiles Android. Puisque le système d'Android n'est pas capable de compiler les langues de spécification des politiques existants comme XACML (Parducci, Lockhart, & Levinson, 2017), nous avons inventé notre propre langage de spécification des politiques de sécurité pour Android qui est compatible avec les autres langages existants pour garder l'utilité d'échange des politiques avec le monde extérieur comme les nuages informatiques. Les politiques

doivent être considérées à différents niveaux de priorité afin de décider lesquelles appliquer en cas de conflit entre eux.

0.4 Méthodologie suivie

Pour atteindre nos objectifs, nous devons d'abord nous concentrer sur la structure des applications Android, les permissions définies, leurs appels API et la possibilité de modification dans le code source de ces applications à sécuriser pour obtenir le contrôle souhaité. Comme plusieurs approches de réécriture du byte Code existent déjà, nous avons basé notre Framework développé sur la réécriture du code source Java des applications à contrôler. Nous considèrerons alors les méthodologies suivantes dans notre travail :

- Appliquer l'approche de réécriture du code pour instrumenter les applications Android, en injectant du code de contrôle avant l'appel de chaque méthode API qui demande l'autorisation de l'utilisateur. Dans notre approche de réécriture proposée, et pour éviter de lire et de modifier le code source de chaque application, nous avons expérimenté le compilateur AspectJ (The AspectJ Team, 2003) pour gérer la compilation et le tissage des aspects ainsi qu'injecter une dépendance de bibliothèque requise par les aspects. Du coup, nous avons défini toutes les APIs à contrôler qui demandent l'autorisation de l'utilisateur et leurs fonctions dans un fichier AspectJ. Avec l'insertion de ce dernier dans n'importe quelle application, les contrôles seront injectés automatiquement avant les appels APIs lors de compilation de cette application.
- Définir un langage de spécification de politique pour exprimer des politiques de sécurité afin de contrôler l'accès et de gérer les risques. De plus, ce langage sera capable d'exprimer des politiques pour les conflits de ressources. Pour atteindre cet objectif, nous avons développé notre propre langage de spécification des politiques de sécurité. Ce dernier est compilable sur Android sans besoin de modifier le système d'origine. Ce langage est développé nativement sur java et il est extensible sur plusieurs formats comme XML, JSON etc. .Il est de plus compatible avec tous les autres langages existants tels que XACML (Parducci, Lockhart, & Levinson, 2017), UMA (Maler, 2014) etc. Maintenant, en nous concentrant sur les politiques définies par l'utilisateur, nous

adopterons ce langage pour développer une application de contrôle centralisée. Elle permettra de définir des politiques contextuelles en spécifiant divers contextes incluant le temps, l'emplacement, les applications en cours d'exécutions, les ressources utilisées, et l'état du périphérique, etc.

- Réaliser une application de contrôle centralisée. Après avoir maitrisé la réécriture des applications, elles seront instrumentées et, par la suite, pourront communiquer avec le contrôleur. À partir de ce moment, ce dernier sera en mesure de surveiller les autorisations des APIs sensibles à un contexte actuel et mettre en objet des politiques de sécurité basées sur la décision des utilisateurs. Dans le but de fonder une communication entre les applications instrumentées et le contrôleur, nous pouvons personnaliser les intentions entre applications (Chilowicz, 2012). L'utilisateur commence par définir ces politiques de sécurité dans l'interface graphique du contrôleur. Une fois un changement du contexte à était détecté par le contrôleur, ce dernier utilise notre langage de spécification des politiques définis pour exécuter la liste des politiques de l'utilisateur et déterminer quelles sont celles de sécurité qui sont concernées par ce changement du contexte et déterminer la décision du contrôle. Dans le cas où la décision de la politique demande un contrôle sur l'API, le contrôleur notifie l'application concernée pour autoriser ou retirer l'accès à cet API.
- Évaluer la solution proposée à travers des expérimentations. Nous effectuerons nos expérimentations sur les meilleures applications Android existantes sur le marché afin d'évaluer leur performance, leur compatibilité et leurs fonctionnalités.

0.5 Organisation du rapport

Notre rapport est organisé de la façon suivante : présentation des différents concepts de base de la plateforme Android. Le premier chapitre présente un regard rapide sur l'architecture de la plateforme Android, ses mécanismes de bases et ses failles de sécurité.

Dans le deuxième chapitre, on va étudier et analyser les différentes solutions qui accordent la sécurisation de cette plateforme. Cette analyse contient plusieurs solutions faisant part des

deux axes de sécurisation, soient, la modification du système d'Android, et la modification des applications.
Notre solution proposée sera expliquée dans le troisième chapitre. Nous allons donc présenter le Framework de réécriture d'application ainsi que le contrôleur basé sur le langage de spécification des politiques de sécurité.

Le chapitre quatre va comporter les paramètres d'évaluation et les ensembles de données adoptés dans nos tests. Cette partie renfermera les différents tests accomplis dans notre solution (réécriture des applications, contrôleur d'accès des APIs et langage de spécification des politiques de sécurité), et illustrera les résultats obtenus.

Et finalement, une conclusion et des perspectives closeront notre rapport.

CHAPITRE 1

SÉCURITÉ SOUS ANDROID

1.1 Introduction

De nos jours, le smartphone devient une chose primordiale dans notre vie. C'est pourquoi, son marché devient de plus en plus vaste. L'amélioration de ces appareils intelligents et l'accroissement de la requête des clients encouragent les développeurs à se concentrer sur l'évolution de ces différentes plateformes. Par conséquent, le marché est envahi par un énorme nombre d'applications mobiles. Ces derniers garantissent, bien évidemment, des avantages multiples, mais, comportent en contrepartie des risques importants.

La sécurité du système d'exploitation mobile Android est la motivation de la présente recherche. De ce fait, des informations brèves sur la plateforme Android, les applications, les logiciels hostiles et les fonctionnalités de sécurité seront présentées dans cette section.

1.2 Le Plateforme Android

En tant que système d'exploitation open source pour les Smartphones, Android est basé sur le noyau Linux et supporte plusieurs plateformes embarquées comme ARM, x86, MIPS, etc. Il a été fondé par la startup Android puis racheté et développé par Google (Gibler, Crussell, Erickson, & Chen, June 13-15, 2012). Les utilisateurs et les développeurs ont la possibilité de télécharger et installer des applications à des fins légitimes, grâce à Android. La plateforme Android est une pile logicielle conçue principalement afin de prendre en charge les appareils mobiles tels que les téléphones et les tablettes (Rasthofer, Arzt, Lovat, & l, DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android, 2014). La Figure 1.1, montre l'architecture de système Android qui se présente sous la forme de quatre couches fondamentales. Ces couches allant des services de système d'exploitation de bas niveau qui dirigent le périphérique mobile jusqu'aux applications, telles que le numéroteur téléphonique et le navigateur Web (Espiau, L'architecture d'Android, 2017).

En bas de l'architecture, il y a la couche noyau Linux qui apporte les services de base sur lesquels s'appuiera tout l'appareil Android, tel que la mémoire, la gestion de l'alimentation, les restrictions d'accès aux données et aux ressources. Au-dessus de la couche du noyau, il existe les bibliothèques du système qui sont généralement écrites en langage C et C++ qui sont souvent appelés les bibliothèques natives. Ces dernières ont été améliorées pour effectuer plusieurs processus tels que les calculs mathématiques et les allocations de mémoire. Cette couche comporte, en outre, le système d'exécution Android qui assume l'écriture et l'exécution d'applications Android. Le moteur d'exécution Android comprend deux constituants importants : les bibliothèques Java principales et la machine virtuelle Dalvik. Les applications Android sont habituellement écrites en Java, et le compilateur Java compile le fichier de code source Java en plusieurs fichiers Java Byte-code. Puis, un outil appelé DX (Mann & Starostin, A Framework for Static Detection of Privacy Leaks in Android Applications, 2012) transforme les fichiers Java Byte-code en un seul fichier DEX Byte-code tel que Classes .DEX. Ensuite, le fichier DEX est empaqueté avec d'autres ressources des applications qui sont installées sur le périphérique.

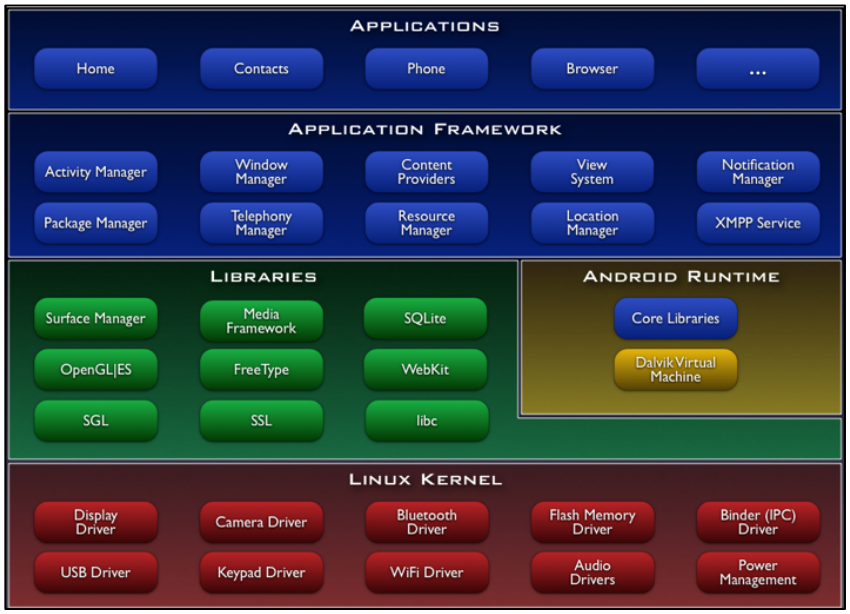


Figure 1.1 L'architecture de système d'exploitation Android
(Espiau, L'architecture d'Android, 2017)

Lors de la dernière étape, lorsque l'utilisateur lance l'application installée, la machine virtuelle Dalvik exécute le fichier DEX Byte-code. La 3^{éme} couche est le cadre d'application pour soutenir le développement de nouvelles applications. Enfin, tout en haut du cadre, Android fournit des applications standard telles que la camera, la messagerie, les courriels et les contacts.

1.3 La machine Dalvik

La machine virtuelle Dalvik intégrée dans le système Android est destinée pour autoriser une exécution de plusieurs applications simultanément sur un appareil qui a des performances limitées en mémoires et en puissance. Dalvik fait partie du runtime, créé par Dan Bornstein, qui présente un moteur permettant l'exécution des applications Android. Les fichiers de byte-code de JAVA sont transformés auparavant et renforcés dans un fichier « Dalvik executable.dex ».

Dalvik réalise un byte-code différent vu que ses instructions sont fondées sur des registres. Cependant, l'ancienne machine virtuelle du Java est fondée sur la technique de la pile (Scriptol, 2017). Cette machine à registre demande le minimum d'instructions afin d'accomplir les opérations semblables faites par une machine à pile. Par conséquent, elle correspond le mieux à un appareil de faible puissance de calcul. Vu cette dissemblance, Dalvik ne peut pas exécuter les fichiers de byte-code Java ordinaire à l'identique. Par contre, il existe un programme dans le SDK Android qui convertit ces fichiers en d'autres DEX (Dalvik Exécutable). Ces derniers ont été créés par renforcement des fichiers de byte-code. La taille du fichier.dex est typiquement la moitié de la taille des fichiers de byte-code Java duquel il est issu.

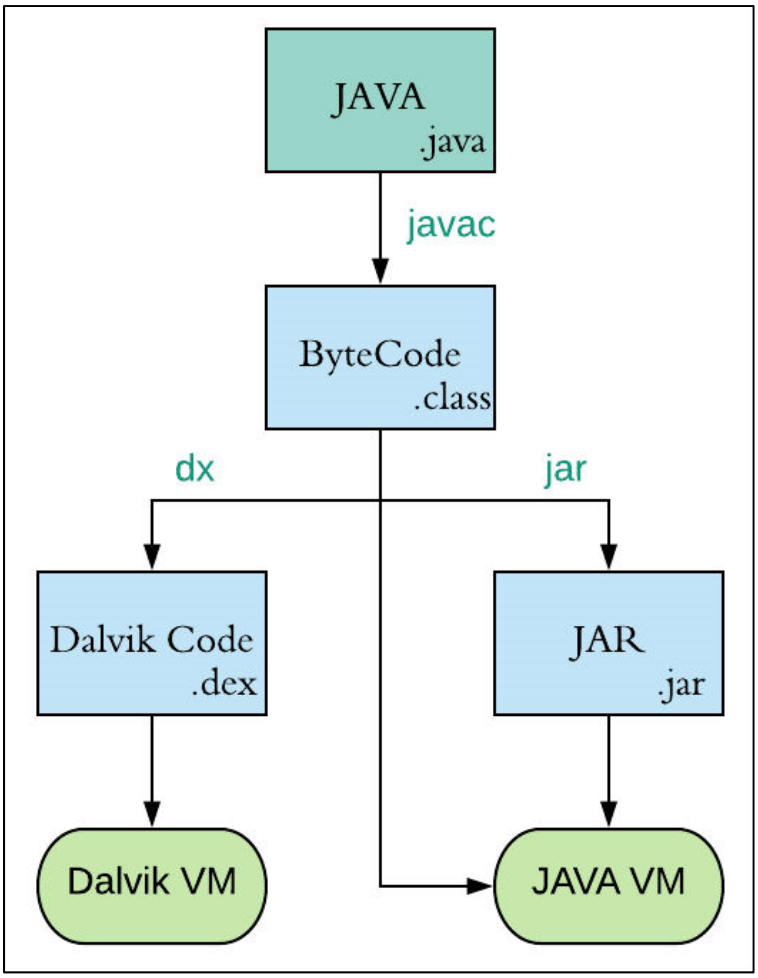


Figure 1.2 L'architecture de la machine
Dalvik (Scriptol, 2017)

Comme le montre la Figure 1.2, nous remarquons bien l'existence de la différence entre les applications Java et le système compilé Android. Le byte-code Java classique (.class) est obtenu, en premier lieu, par la compilation du code source des applications Android en Java qui, par la suite, a été recompilé en un byte-code connaissable par la machine Dalvik. Par conséquent, les fichiers .class sont transformés en fichier.dex. En guise de conclusion, nous pouvons conclure que cette machine joue le rôle d'un écran cachant les caractéristiques techniques de l'appareil d'exécution.

1.4 Les Applications Android

L'application Android est générée comme étant un fichier d'installation intitulé fichier Android Package (ou APK). Ce dernier est une collection des paquets compressés par le système d'exploitation Android dont chacun comprend le fichier qui manifeste l'application, les ressources, et le code d'application codé par la machine virtuelle Dalvik (DVM) sous la forme d'un fichier classes.dex. Le fichier APK doit être signé pour vérifier son authenticité. Comme le dévoile la Figure 1.3 , lorsqu'un appel de méthode est effectué pour accéder à une ressource privilégiée, l'appel passe par l'API Android et l'infrastructure d'application vérifie l'autorisation de poursuivre la demande de l'application d'origine (Google, 2016).

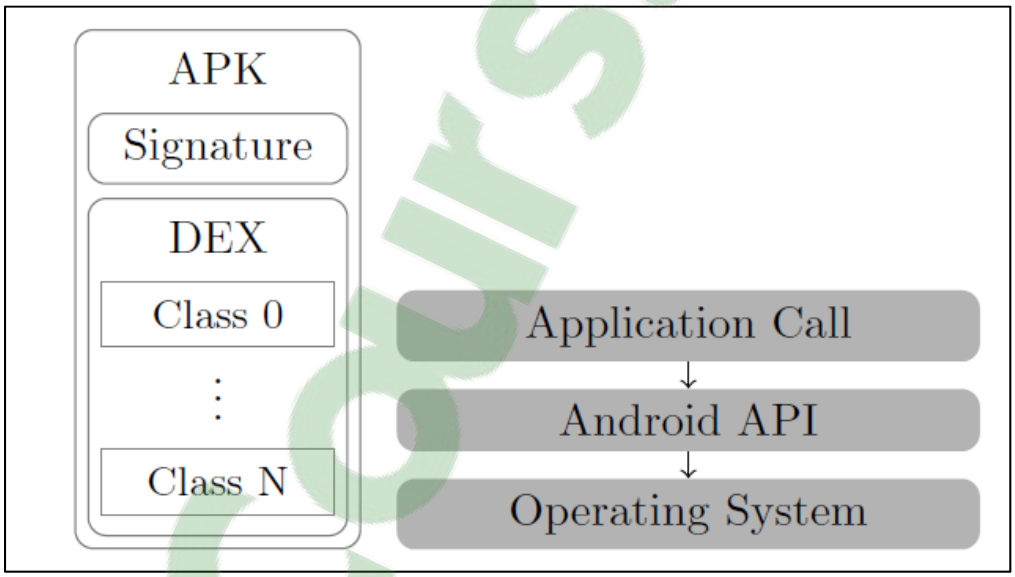


Figure 1.3 Information générale sur le fonctionnement d'une application Android (El-Harake, Falcone, Jerad, Langet, & Mamlouk, 2014)

Toute application a cinq composants fondamentaux : Activités, Services, Content Providers, BroadcastReceivers et Intents (Google, 2016) et (Alkurdi, 2014).

Figure 1.4 Les composants d'une application Android (Tutorial-all, 2017)

Les détails de ces composants (*Voir* Figure 1.4) sont indiqués comme suit :

- **Activités** : Une activité représente un seul écran dans l'interface utilisateur. Presque toutes les activités interfèrent avec l'utilisateur. La classe Activité s'occupe donc de créer une fenêtre dans laquelle il est possible de placer l'interface utilisateur avec setContentView(View). Les utilisateurs implémentent cela en subdivisant la classe Activité et en adaptant les rappels de cycle de vie nécessaires. Chaque application a de nombreuses activités avec des objectifs différents. Ces activités sont indépendantes les unes des autres. Par exemple, l'application SMS a deux activités, la 1 ère est de montrer une liste des SMS reçus, la 2eme est de rédiger un SMS, ajouter des émoticônes, des images, des enregistrements et les envoyer.
- **Services** : Un service est un composant d'application qui représente le souhait d'une application de réaliser une opération plus longue sans interagir avec l'utilisateur ou de fournir des fonctionnalités pour d'autres applications à utiliser. Le service est généralement démarré par une activité, bien qu'il puisse également être démarré par d'autres composants. Contrairement au composant d'activité, le composant Service ne

fournit pas d'interface utilisateur. Par contre, il fonctionne en arrière-plan pour exécuter des opérations de longue durée. Citons l'exemple des fonds d'écrans animés, qui sont des services lancés par l'application fond d'écran et qui reste en exécution en arrière-plan dans le Smartphone pendant que l'adjudicataire utilise plusieurs autres applications.

Il faut noter que les services, comme les autres objets d'application, s'exécutent dans le thread principal de leur processus d'hébergement. C'est-à-dire, que si votre service doit effectuer des opérations intensives (telle que la lecture de fichiers MP3) ou de blocage (telle que la mise en réseau), il devrait générer son propre thread dans lequel effectuer ce travail.

- **Content Providers** : les contents providers peuvent aider une application à gérer l'accès aux données stockées par elle-même, stockées par d'autres applications, et fournir un moyen de partager des données avec d'autres applications. Ils enferment les données et fournissent des mécanismes pour définir la sécurité des données. Les contents providers présentent l'interface standard qui relie les données d'un processus au code exécuté dans un autre processus. Le content provider peut être configuré permettant à d'autres applications d'accéder en toute sécurité et de modifier les données d'une application. Ce producteur de contenu est considéré comme étant une solution favorable pour le partage des données entre les applications, par ailleurs, il est généralement utilisé comme solution de sauvegarde des données.
- **BroadcastReceiver** : Un BroadcastReceiver (récepteur des diffusions) est un composant Android qui permet au développeur d'enregistrer pour des événements système ou d'application. Une fois qu'un événement se produit, tous les destinataires enregistrés pour cet événement sont notifiés par le moteur d'exécution Android.

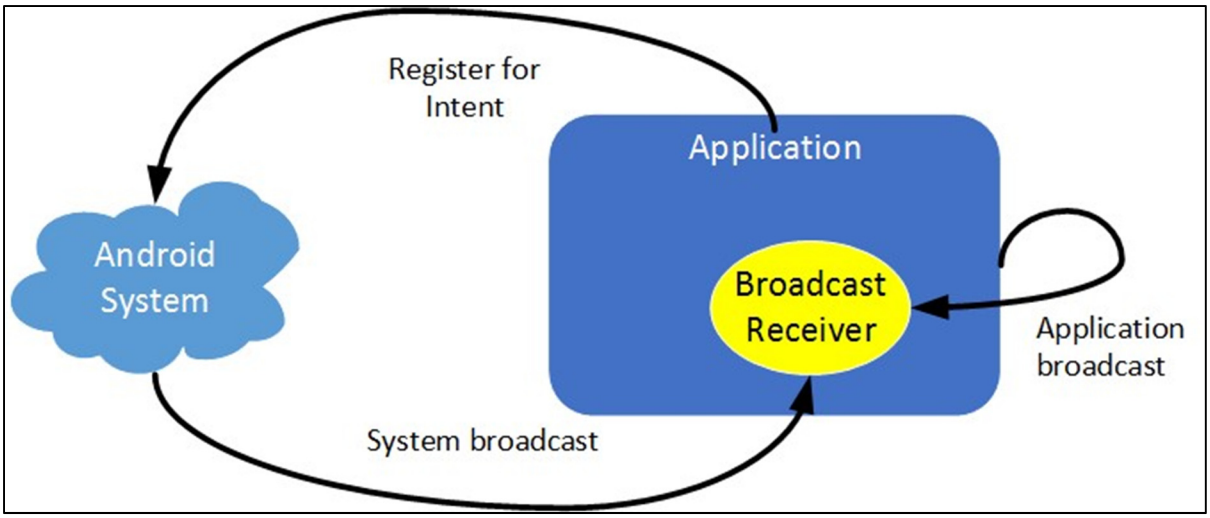


Figure 1.5 Diffusion des données avec BroadcastReceiver (Align Minds, 2015)

Tout simplement et comme la montre la Figure 1.5, afin de recevoir des Intents dans la même application ou d'autres, Android nous donne la possibilité de créer une classe qui implémente BroadcastReceiver. Ces objets sont conçus dans le but de recevoir des Intents (Intentions) et appliquer des comportements spécifiques à notre code. Par exemple, les applications peuvent s'inscrire à l'événement système ACTION_BOOT_COMPLETED qui est déclenché une fois que le système Android a terminé le processus de démarrage, la batterie est faible ou une capture d'image, etc. Tous ces événements sont originaires du système. Les applications elles-mêmes peuvent aussi lancer des diffusions. Prenant à titre d'exemple, l'application SMS en diffusant la réception d'un SMS et en permettant aux autres applications de connaître cet événement afin qu'elles puissent déclencher certaines actions.

Contrairement aux activités, le BroadcastReceiver n'est pas une interface utilisateur, mais il peut créer une notification de la barre d'état. Son rôle est de minimiser le travail au maximum et de déléguer des emplois difficiles aux services. Par ailleurs, il reçoit un objet Intent, qui sera bien détaillé ultérieurement.

De ce fait, l'essentiel c'est que les récepteurs de diffusion sont considérés comme des composants d'application inactifs qui peuvent s'inscrire à divers événements de système ou

d'application (Intent). Une fois que l'un de ces événements s'est produit, le système notifie tous les récepteurs de la diffusion enregistrés et les met en action.

L'interface BroadcastReceiver possède uniquement une seule méthode onReceive() qui devra être implémentée.

Sa durée de vie est limitée au temps du traitement de votre onReceive(). Une fois l'onReceive() de la classe de réception terminé, le système Android est autorisé à recycler le récepteur et supprimer l'instance par le Garbage Collector.

Deux étapes importantes sont nécessaires pour que le BroadcastReceiver soit capable d'intercepter les Intentions diffusées par le système :

- **Création du BroadcastReceiver** : Un BroadcastReceiver est implémenté en tant que sous-classe de la classe BroadcastReceiver et que chaque message reçu dans la méthode onReceive() contient un objet Intent en paramètre.
- **Enregistrement du BroadcastReceiver** : Dès que votre appareil Android est démarré, il sera intercepté par BroadcastReceiver. Plusieurs événements générés par le système sont définis en tant que champs statiques finaux dans la classe Intent tels que : BATTERY_CHANGED, BATTERY_LOW, BOOT_COMPLETED, CALL etc.

Pour que l'application génère et envoie des Intentions personnalisées, ces dernières devraient être créées et envoyées en utilisant la méthode sendBroadcast() dans votre classe d'activité. Afin d'intercepter la diffusion de ces Intents, il suffit d'enregistrer un BroadcastReceiver quelles que soient votre application ou d'autres applications.

- **Intents** : Une Intention Android est une description abstraite d'une opération à effectuer. Elle peut être utilisé pour lancer une activité, envoyer des données à tout composant

BroadcatReceiver intéressé, ou pour communiquer avec des services en arrière-plan (Espiau, Créez des applications pour Android, 2017).

De plus, un Intent permet d'effectuer une liaison d'exécution tardive entre le code dans des différentes applications. Son utilisation la plus significative et la plus typique est dans le lancement d'activités, où il peut être considéré comme le lien entre les activités. Il s'agit essentiellement d'une structure de données passive contenant une description abstraite d'une action à effectuer. Un intent est bel et bien un objet contenant plusieurs champs, bien illustrés ci dissous (*Voir* Figure 1.6).

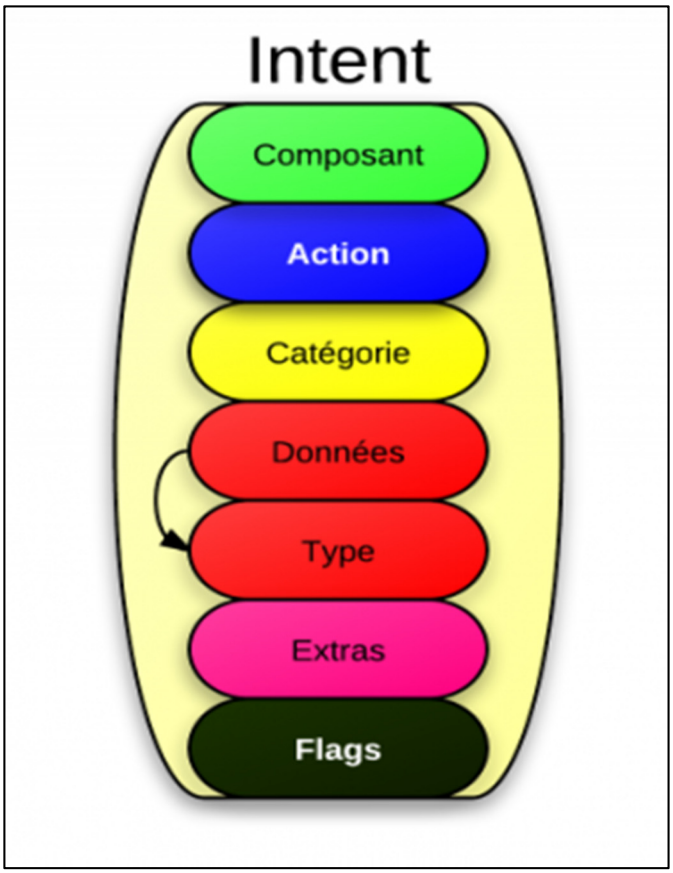


Figure 1.6 L'architecture de l'Intent
(Espiau, Créez des applications pour Android, 2017)

La nature et les objectifs de l'intent sont déterminés à partir de la façon dont sont expliqués ces champs. En effet, un intent ayant un champ composant renseigné est dit « explicite ». Ce dernier indique la destination que l'intent doit gérer. Un champ se constitue de deux informations : le package où le composant est situé et le nom du composant. Par conséquent, à l'exécution de l'intent, Android peut retrouver plus précisément le composant de destination.

De plus des intents explicites, il existe des intents dits « implicites ». À cet égard, nous ne pouvons pas connaitre de façon précise le destinataire de l'intent. C'est pourquoi, alors, nous allons nous intéresser à informer d'autres champs dans le but de laisser Android déterminer lequel est capable de recevoir cet intent. Il faut fournir au moins six informations essentielles, à savoir :

- **Une action** : ce que doit faire le destinataire.
- **Un ensemble de données** : permet de savoir les données sur lesquelles le destinataire doit actionner.
- **La catégorie** : accorde l'apport des informations supplémentaires sur l'action à réaliser ainsi que le type de composant géré par l'intent.
- **Le type** : sers à révéler le type des données incluses. Généralement, ce type demeure dans les données, néanmoins, nous pouvons désactiver la vérification automatique et imposer un type particulier tout en précisant cet attribut.
- **Les extras** : servent à enrichir le contenu des intents dans le but de les circuler entre les composants.
- **Les flags** : offre la possibilité de changer le comportement de l'intent.

Il est possible d'introduire des données dans un intent suivant deux différents types :

- **Types standards** : les Intents possèdent un champ « extra » ; ils permettent d'héberger des données à circuler entre les applications. Un extra est une clé dont nous pouvons joindre une valeur. Il suffit juste utiliser la commande putExtra() pour insérer un extra.
- **Les parcelables** : Il est bien de noter qu'un parcelable demeure un objet à transmettre à un parcelayant dans le but de transférer des messages entre différents processus du système.

1.5 Les vulnérabilités du système Android

D'après les statistiques de CVEDetails, Android est le système d'exploitation le plus vulnérable en 2016. Aujourd'hui, l'OS de Google comptabilise trois fois plus de lacunes que celui d'Apple (Pétrod, 2017).

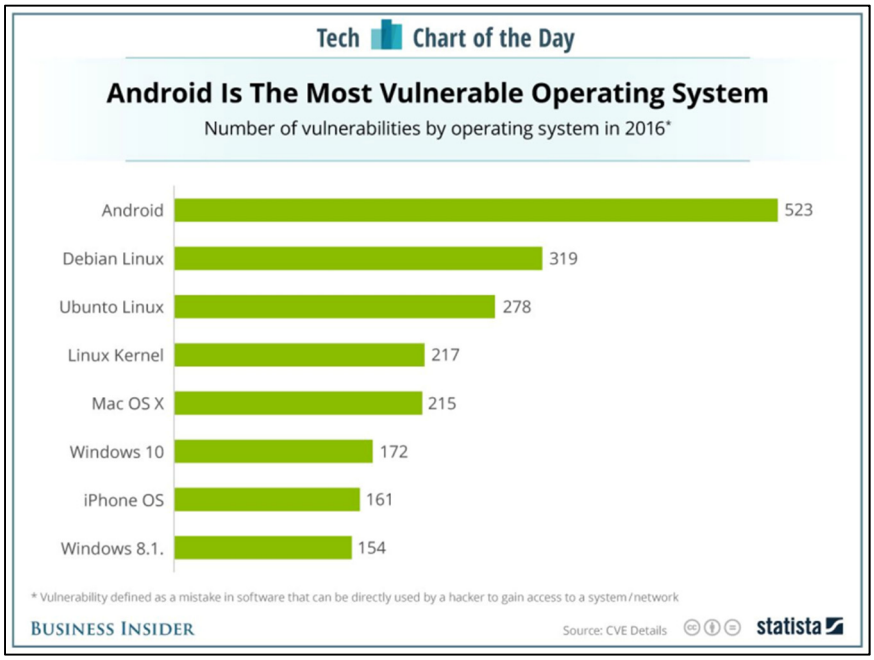


Figure 1.7 Nombre de vulnérabilités par système d'exploitation en 2016 (Pétrod, 2017)

En analysant de plus près les chiffres d'Android au cours des dernières années (*Voir* Figure 1.7), nous constatons que la vulnérabilité de ce système d'exploitation a augmentée de quatre fois à l'année 2016 par rapport aux années précédentes (Ketfi, 2017). Ceci est expliqué par la succession de plusieurs facteurs néfastes. Citons à titre d'exemple, l'émergence du virus Godless qui a pu toucher plus de 90% des appareils Android, ou encore la découverte de backdoors installés par des fabricants chinois.

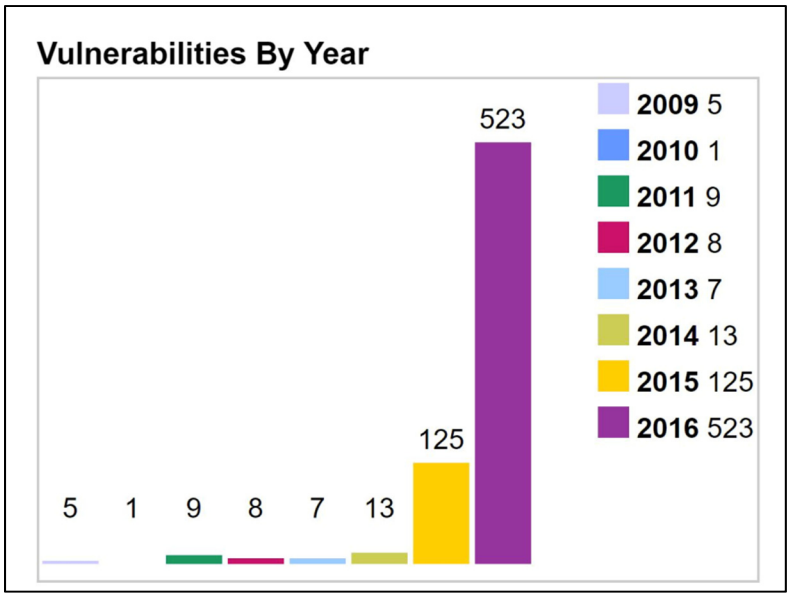


Figure 1.8 Nombre de vulnérabilités par année dans
le système d'exploitation Android (Ketfi, 2017)

Un utilisateur de Smartphone est exposé à différentes menaces lors de l'utilisation de son appareil (*Voir* Figure 1.8). Ces menaces peuvent atteindre le bon fonctionnement d'un Smartphone, ou transporter ou encore remanier les données de l'utilisateur. Pour cela, chaque application qui est développée doit avoir des garanties de confidentialité et d'équité par rapport aux informations altérées. En effet, différents logiciels peuvent être considérés comme malveillants, leurs activités et leurs fonctionnements devront être restreints. (Chercher une position GPS, consulter le carnet d'adresses, l'envoi d'un SMS facturé, etc.).

Les cibles préférées par les attaquants sont les suivantes :

- **Les données** : étant donné que nous pouvons considérer les Smartphones comme des appareils de gestion de données, alors, ils peuvent posséder des données classées sensibles à noter : les numéros de carte bancaire, des informations d'authentification, et des journaux d'activités (calendrier, journal d'appel).
- **L'identité :** les Smartphones sont considérés très personnalisables vu que leur contenu est bien lié à une personne précise. À titre d'exemple, chaque appareil mobile est bel et

bien lié au propriétaire du contrat du mobile et peut faire l'objet d'une attaque voulant voler son identité.

- **La disponibilité :** lors d'une attaque d'un Smartphone, nous pouvons réduire l'accès au propriétaire tout en lui forçant de passer en privé ou hors service.

1.6 Les Malwares sur le système Android

Un Malware est un logiciel malveillant développé dans le but de gêner, d'abuser de la vie privée de l'utilisateur ou d'effectuer toute autre activité illégale. Généralement, un logiciel malveillant est installé sur l'appareil de la victime à son insu en acceptant des autorisations qui pourraient ouvrir la porte à des comportements malveillants inattendus. La Figure 1.9 montre les effets des malwares (Picard, 2012).

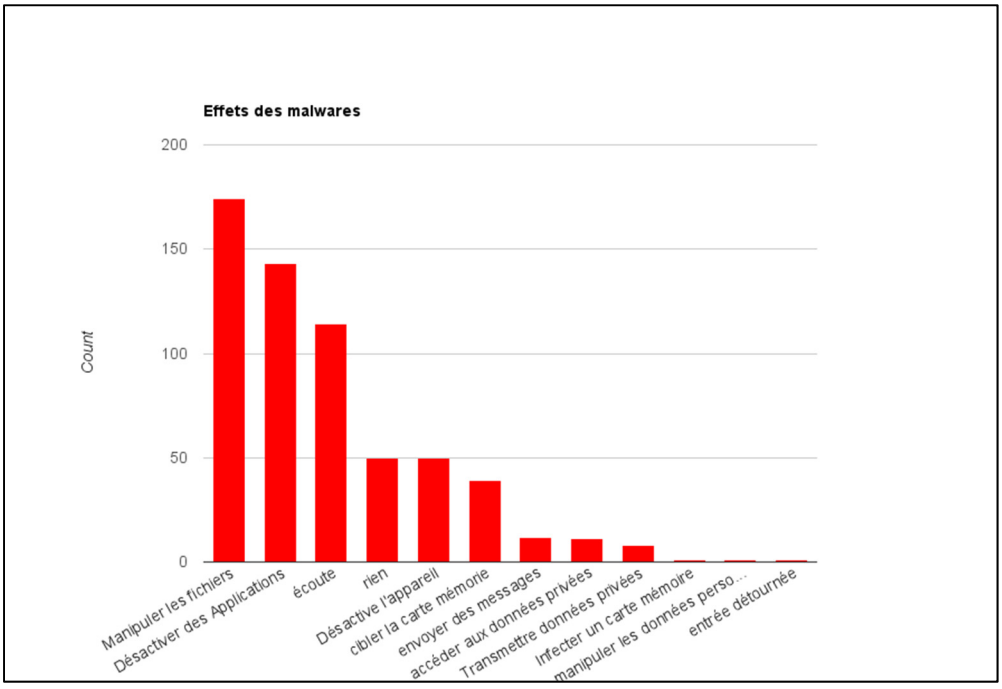


Figure 1.9 Les effets et actions des logiciels malveillants sur les smartphones Android (Picard, 2012)

Selon leurs aspects, les logiciels malveillants Android ont été classés dans différents groupes. Par exemple, les auteurs (Jiang & Zhou, 2013) ont systématiquement classé les logiciels malveillants Android existants en fonction de leurs caractéristiques, notamment l'installation, le reconditionnement et les méthodes d'activation. Dans une autre perspective, (Cooper, Shahriar, & Haddad, 2014) ont classé les applications malveillantes en fonction de leur capacité à effectuer des opérations spécifiques sur la plateforme Android, comme indiqué dans le Tableau 1.1.

Tableau 1.1 Types de malware, actions et autorisations nécessaires
(Cooper, Shahriar, & Haddad, 2014)

Type du Malware	Exemple	Permission
Changement du fond d'écran	Nouveauté et amusement en changeant le fond d'écran par défaut.	SET_WALLPAPER BIND_WALLPAPER
Vente des informations d'identification de l'utilisateur	Accédez aux informations de contact de l'appareil et envoyez les informations via Internet.	USE_CREDENTIALS READ_PROFILE MANAGE_ACCOUNTS
Appels payants et SMS	Facture la victime en initiant arbitrairement des appels téléphoniques et des messages texte à des numéros premium.	READ_SMS WRITE_SMS SEND_SMS
Changement de configuration du téléphone	Modification de la configuration par défaut, telle que l'alerte sonore, le verrouillage des téléphones et d'autres paramètres.	WRITE_SETTINGS
Piratage des comptes de réseaux sociaux	Secrètement accéder et mettre à jour les informations de profil d'utilisateur.	READ_SOCIAL_STREAM WRITE_SOCIAL_STREAM

En plus des classifications de logiciels malveillants mentionnés ci-dessus, de nombreuses autres applications malveillantes peuvent exécuter différents scénarios, tels que l'affichage de publicités bien que certains chercheurs ne les considèrent pas comme des applications malveillantes lorsqu'elles sont utilisées par des entreprises fiables pour vendre leurs produits. Les techniques de pénétration malwares Android peuvent être classifiés en trois classes (Zhou & Jiang, Dissecting Android Malware: Characterization and Evolution, 2012.)

- **Technique de réécriture de l'application** : Cette technique est la plus populaire, elle consiste à injecter le code malveillant dans des applications propres en conservant la même structure que l'application d'origine. Comme l'application reconditionnée a la même structure que l'application d'origine, il devient difficile de détecter le processus de reconditionnement.
- **Technique de mise à jour** : Au lieu d'injecter l'ensemble du code malveillant dans une application propre, dans un premier temps, cette technique consiste à injecter a un composant mis à jour dans l'application propre. En deuxième lieu, le code malveillant sera injecté lors du processus de mise à jour, ce qui rend la détection du code injecté plus difficile, surtout lorsqu'il s'agit de reconditionner des applications populaires.
- **Technique de téléchargement** : Cette technique consiste à motiver les utilisateurs à télécharger des applications intéressantes et attrayantes ce qui engendrera le téléchargement des applications malveillantes.
- Les Malwares peuvent être trouvés dans plusieurs types comme présentés dans (Alkurdi, 2014), (Saini, 2015):
- **Virus** : un code qui s'insère dans un autre programme et se réplique, c'est-à-dire, se copie et infecte d'autres machines. De nos jours, il est généralement utilisé comme un terme générique qui inclut également les vers et les chevaux de Troie.
- **Worm** : logiciel malveillant auto-répliquant qui se copie sur d'autres nœuds d'un réseau en utilisant les vulnérabilités du système en dehors de l'intervention de l'utilisateur. Contrairement aux virus, les vers ne s'attachent pas à une application.
- **Cheval de Troie** : programme malveillant qui se fait passer pour une application. Contrairement aux virus et aux vers, le cheval de Troie ne se réplique pas.
- **Rootkit** : logiciel qui permet de continuer à accéder à un ordinateur avec un accès privilégié tout en masquant activement son activité malveillante auprès des administrateurs en modifiant les fonctionnalités du système d'exploitation.
- **Backdoor** : cheval de Troie spécialisé se faisant passer pour un programme installé afin d'accéder à distance à un système et contourner l'authentification normale. De plus, les portes dérobées tentent de ne pas être détectées.

- **Spyware** : Logiciel qui révèle des informations privées sur l'utilisateur ou le système informatique à des oreilles indiscrètes.
- **Bot** : un logiciel malveillant qui permet au botmaster, plus précisément à l'auteur des commandes, de contrôler à distance le système infecté. Un groupe de systèmes infectés contrôlés sont appelés botnets, chargés par le botmaster d'effectuer plusieurs activités malveillantes telles que l'arrêt des services, le vol d'informations privées et l'envoi de spam.

1.7 La sécurité sous Android

Comme mentionné précédemment, le système Android est une plateforme open source où les applications sont publiées sur différents marchés sans être surveillées ou analysées pour conserver leur comportement. A ce titre, les mécanismes de protection de la plateforme Android sont utilisés au lieu de ceux du marché (Kim, Yoon, Yi, & Shin, 2012).

1.7.1 Protection par bacs à sable

Le modèle d'application Android garantit aux développeurs un ensemble complet de fonctionnalités et un environnement relativement sécurisé, particulièrement, pour les applications simples. Chaque application sous Android est " bacs à sable " dans son propre processus et l'espace du système de fichiers (Schiefer, 2014). Profitant des mécanismes de protection de l'espace de processus du noyau Linux, Android affecte à chaque application installée un ID utilisateur unique. A l'opposé des systèmes d'exploitation de bureau traditionnels (Linux, Windows, Mac, etc.), Android utilise généralement le concept de l'ID utilisateur pour représenter une application plutôt qu'un utilisateur humain. Cela permet au noyau de garder les applications confinées en mémoire, de restreindre d'une part l'accès au matériel ou aux services sous-jacents et d'autre part l'accès au système de fichiers sur le périphérique. Par défaut, les données et ressources de chaque application sont hébergées dans un emplacement auquel seuls l'application et le Framework de base peuvent accéder : une conception essentielle pour le modèle de sécurité Android (*Voir* Figure 1.10).

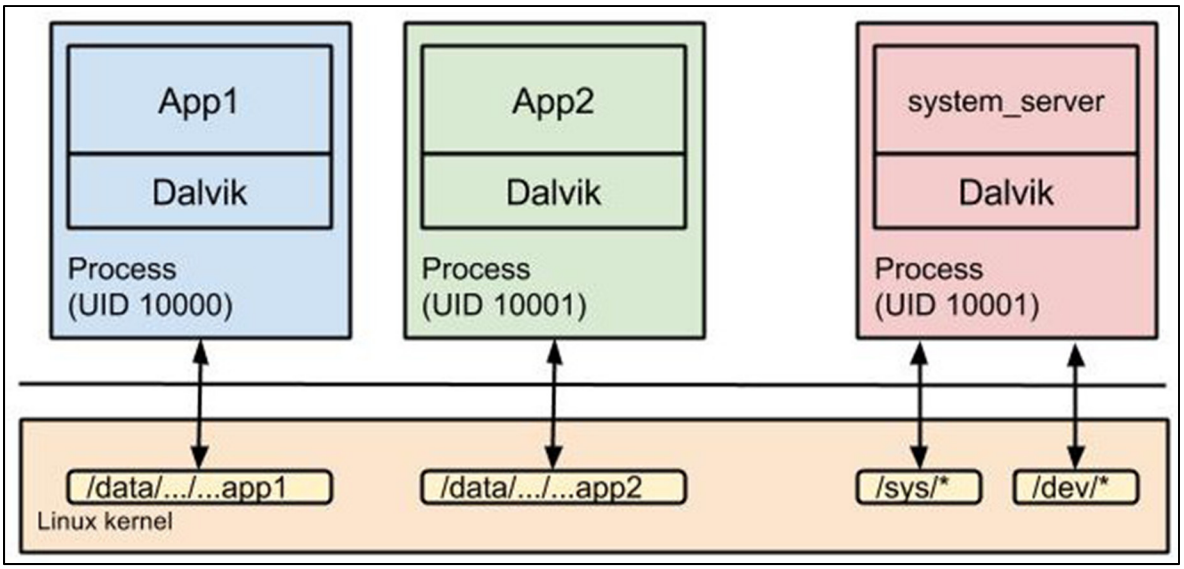


Figure 1.10 La protection par bacs à sable (Sandboxing) sur les applications Android (Schiefer, 2014)

Les applications les plus avancées deviennent courantes, ces applications communiquent entre elles ou avec des appareils / serveurs via des réseaux.

Globalement, lors de l'installation des applications Android, chaque application recevra un identifiant utilisateur unique (UID). Le noyau Linux est responsable de l'application du contrôle d'accès aux ressources du système. Aucune application dès lors, ne pourra accéder aux fichiers d'autres applications. Par ailleurs, chaque application est exécutée dans des machines virtuelles distinctes. Ainsi, aucune application vulnérable n'affectera d'autres applications (Karmakar, 2012)

1.7.2 Le modèle des permissions

Android protège les APIs sensibles en leur attribuant des autorisations afin d'amplifier les privilèges des applications sur l'appareil, y compris l'accès aux données et services stockés, tels que réseau, mémoire, etc. Toutes les autorisations requises pour accéder aux API sont protégées dans le fichier manifeste de chaque application (AndroidManifest.xml). Ils sont définies nécessairement par les développeurs d'application Android.

Les Permissions dans Android sont divisées en groupes d'autorisations organisées qui sont liées aux fonctionnalités d'un périphérique (*Voir* Figure 1.11). Sous ce système, les demandes d'autorisation sont gérées au niveau du groupe et un seul groupe d'autorisations correspond à plusieurs déclarations d'autorisations dans le manifeste de l'application. Prenant à titre d'exemple, le groupe SMS qui inclut à la fois les déclarations READ_SMS et RECEIVE_SMS (Google , 2017).

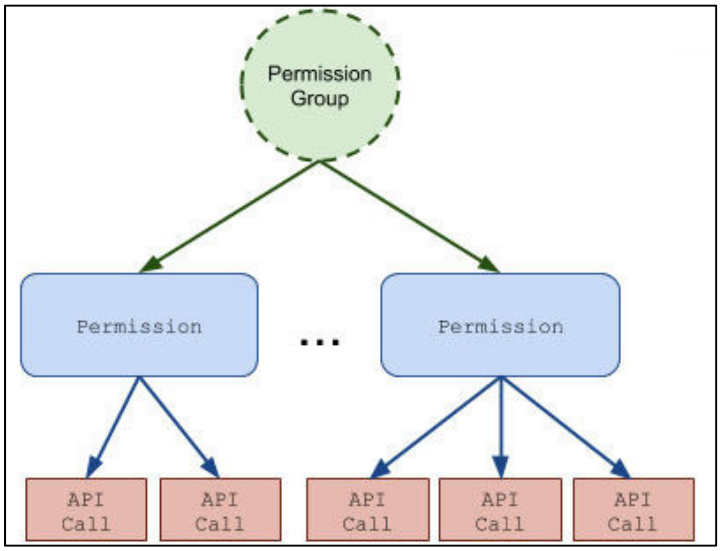


Figure 1.11 Les groupes des permissions des APIs sur Android (Google , 2017)

Les permissions Android sont classées en quatre niveaux de protection :

- **Permissions de protection normale** : Les autorisations normales concernent les APIs dans lesquelles l'application doit accéder à des données ou des ressources en dehors du sandbox de l'application. Cependant, les risques pour la vie privée de l'utilisateur ou le fonctionnement des autres applications est très limité. Par exemple, l'autorisation de définir le fuseau horaire est une autorisation normale. Si une application déclare qu'elle a besoin de cette autorisation, le système accorde automatiquement l'autorisation à l'application. La liste complète des autorisations normales actuelles est la suivante (*Voir* Tableau 1.2).

Tableau 1.2 La liste complète des permissions de protection normale

Nom de permission
• ACCESS_LOCATION_EXTRA_COMMANDS
• ACCESS_NETWORK_STATE
• ACCESS_NOTIFICATION_POLICY
• ACCESS_WIFI_STATE
• BLUETOOTH
• BLUETOOTH_ADMIN
• BROADCAST_STICKY
• CHANGE_NETWORK_STATE
• CHANGE_WIFI_MULTICAST_STATE
• CHANGE_WIFI_STATE
• DISABLE_KEYGUARD
• EXPAND_STATUS_BAR
• GET_PACKAGE_SIZE
• INSTALL_SHORTCUT
• INTERNET
• KILL_BACKGROUND_PROCESSES
• MODIFY_AUDIO_SETTINGS
• NFC
• READ_SYNC_SETTINGS
• READ_SYNC_STATS
• RECEIVE_BOOT_COMPLETED
• REORDER_TASKS
• REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
• REQUEST_INSTALL_PACKAGES
• SET_ALARM
• SET_TIME_ZONE
• SET_WALLPAPER
• SET_WALLPAPER_HINTS
• TRANSMIT_IR
• UNINSTALL_SHORTCUT
• USE_FINGERPRINT
• VIBRATE
• WAKE_LOCK
• WRITE_SYNC_SETTINGS

- **Permissions dangereuses :** Les permissions dangereuses concernent des domaines dans lesquels l'application cherche à disposer des données ou des ressources qui impliquent

des informations privées de l'utilisateur. Ils peuvent aussi affecter les données stockées de l'utilisateur ou le fonctionnement d'autres applications. La possibilité de déceler les contacts de l'utilisateur est par exemple une autorisation dangereuse. Si une application signale qu'elle a besoin d'une autorisation dangereuse, l'utilisateur doit accorder explicitement l'autorisation à l'application. La prévalence des autorisations dangereuses pour toutes les applications Android analysées en juillet 2014 (Gandhi, 2014) est perceptible dans le graphique suivant (*Voir* Figure 1.12).

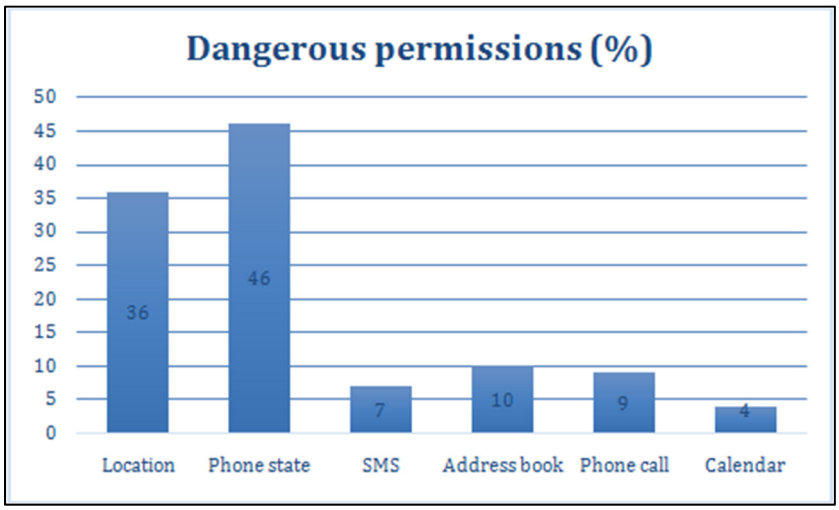


Figure 1.12 La prévalence des autorisations dangereuses
des applications Android (Gandhi, 2014)

Ce graphique montre que parmi les 75K applications Android analysées, 36% des applications exigent des autorisations de localisation. 46% des applications demandent l'autorisation d'état du téléphone, afin d'accéder aux informations des cartes SIM, numéro IMEI, etc. 7% des applications exigent le calendrier.

Le Tableau 1.3 suivant montre la liste complète des autorisations dangereuses actuelles :

Tableau 1.3 La liste complète des permissions dangereuses

Groupe de permission	Permissions	Rôle	Pourquoi c'est dangereux ?
Calendrier	• READ_CALENDAR • WRITE_CALENDAR	Lire les évènements stockés dans le calendrier. Modifier les anciens évènements et créer en de nouveau.	Lors de l'utilisation d'une agenda numérique, l'application va tout savoir sur votre routine quotidienne et pourra même la partager avec des criminels. En plus , une application maligne pourrait effacer les réunions importantes du calendrier.
Caméra	• CAMERA	L'accès à la caméra permet à l'application de prendre des photos et enregistrer des vidéos sur le téléphone.	Une application peut secrètement enregistrer des vidéos ou prendre des photos à tout moment.
Contacts	• READ_CONTACTS • WRITE_CONTACTS • GET_ACCOUNTS	Lire les contacts, les modifier ou en ajouter de nouveaux. Accéder à la liste des comptes.	Une application peut accéder à votre carnet d'adresses. Ces données sont ciblées par les arnaqueurs. Cette autorisation permet en outre d'accéder à la liste de tous les comptes utilisés dans les applications de cet appareil (Google, Twitter , Instagram ..).
Emplacement	• ACCESS_COARSE_LOCATION • ACCESS_FINE_LOCATION	Accès à votre position approximative, fournie sur la base des données des stations de base cellulaires et des points d'accès Wi-Fi. Accès à votre emplacement exact, fourni sur la base des données GPS.	L'application sait toujours votre emplacement actuel. Cela pourrait faire savoir aux malfaiteurs que vous êtes loin de chez vous.
Microphone	• RECORD_AUDIO	Enregistrer l'audio du microphone.	L'application peut enregistrer non seulement toutes vos conversations quand vous parlez au téléphone, mais aussi tout ce qui se passe près de votre appareil toute la journée.
Téléphone	• READ_PHONE_STATE • CALL_PHONE • READ_CALL_LOG • WRITE_CALL_LOG • ADD_VOICEMAIL • USE_SIP • PROCESS_OUTGOING_CALLS	La lecture de l'état du téléphone permet à l'application de connaitre le numéro de l'appareil de téléphone, les informations sur le réseau cellulaire actuel, l'état des appels en cours, etc. Faire des appels téléphoniques. Lire la liste des appels et la changer. Ajouter un message vocal. Utilisez VoIP. Traiter les appels sortants ce qui permet à l'application d'afficher qui appelle, de raccrocher le téléphone ou de le rediriger vers un autre numéro.	Dès votre accord des autorisations de téléphone, vous autorisez l'application à prendre les actions associées aux communications vocales. L'application saura tout à propos de votre appel elle pourra également appeler n'importe où, y compris les numéros payants, à votre charge.

Tableau 1.4 La liste complète des permissions dangereuses (Suite)

Groupe de permission	Permissions	Rôle	Pourquoi c'est dangereux ?
Capteur	• BODY_SENSORS	Cette autorisation permet d'accéder à vos données de santé à partir de certains capteurs, tels qu'un moniteur de fréquence cardiaque.	Si vous utilisez des accessoires avec des capteurs corporels (pas les capteurs de mouvement intégrés du téléphone), l'application reçoit des données sur ce qui se passe avec votre corps.
SMS	• SEND_SMS • READ_SMS • RECEIVE_SMS • RECEIVE_WAP_PUSH • RECEIVE_MMS	Envoyer des messages SMS, les recevoir, lire et renvoyer Recevoir des messages Push WAP et des messages MMS entrants.	Il permet à l'application de recevoir et de lire vos messages SMS entrants ainsi que de les envoyer). Par exemple, les fraudeurs peuvent utiliser cette autorisation en poussant des utilisateurs à abonner à des services payants non désirés.
Espace de stockage	• READ_EXTERNAL_STORAGE • WRITE_EXTERNAL_STORAGE	Lire une carte SD ou un autre stockage dans lesquels on peut, enregistrer les enregistrements	L'application peut lire, modifier ou supprimer tous les fichiers stockés sur le téléphone.

- **Permissions de signature** : Une autorisation peut être accordée par le système, uniquement, si l'application qui demande l'accès est signée par la même certificat que l'application qui a déclaré l'autorisation. Si les certificats correspondent, le système accorde automatiquement l'autorisation sans avertir l'utilisateur sinon il demande l'approbation explicite de l'utilisateur.

- **Permissions de signature ou système** : Ce type d'autorisation est accordé par le système uniquement aux applications figurant dans l'image système Android ou qui sont signées avec le même certificat que l'application ayant déclaré l'autorisation. Cette option est à éviter, car le niveau de protection de la signature doit être suffisant pour la plupart des besoins et fonctionne indépendamment du lieu d'installation des applications. L'autorisation "SignatureOrSystem" est utilisée pour certaines situations spéciales où plusieurs fournisseurs ont des applications intégrées dans une image système et ont besoin de partager des fonctionnalités spécifiques explicitement parce qu'elles sont construites ensemble.

1.7.3 Le système de permission d'Android Marshmallow

Depuis octobre 2015, Google a développé la version 6.X du système d'exploitation mobile connu par Android Marshmallow. Ce dernier est décrit par le niveau d'API 23. Principalement, Marshmallow est basé sur l'amélioration de l'expérience globale de l'utilisateur introduite par Android Lollipop et apporte quelques fonctionnalités supplémentaires (Google, 2016), à titre d'exemples :

- Le système de permissions granulaires (accès aux contacts, localisation, etc.)
- L'assistance vocale peut désormais être lancée depuis l'écran de verrouillage.
- Introduction de Google Now on Tap, permettant, en appuyant sur le bouton Home, d'afficher des informations relatives à l'écran courant.
- Introduction d'un mode ne pas déranger programmable remplaçant le mode silencieux.
- Les touches de navigation rapide changent de place sur les tablettes.

Ce qui nous intéresse dans Android Marshmallow, c'est que nous avons plus de contrôle sur les autorisations que les applications demandent dans l'appareil mobile. C'est parce qu'Android utilise maintenant ce que nous appelons un modèle d'autorisation granulaire. Au lieu de tout demander au moment où vous téléchargez une application, vous êtes autorisé à accorder ou refuser l'accès à des fonctionnalités particulières au besoin.

Flux de travail des permissions :

Auparavant, l'unique chose qui était requise pour les développeurs était de cocher une case dans les paramètres du projet pour les autorisations dont l'application avait besoin (Google, 2016). Bien que cette partie de la configuration de l'application soit la même, ils doivent incorporer un moyen de demander des autorisations non normales aux utilisateurs et désactiver la fonctionnalité de l'application en fonction de leur réponse. La Figure 1.13 montre une comparaison entre le flux de travail des permissions avant le SDK 23 et avec Marshmallow après le SDK 23 (Montemagno, 2015).

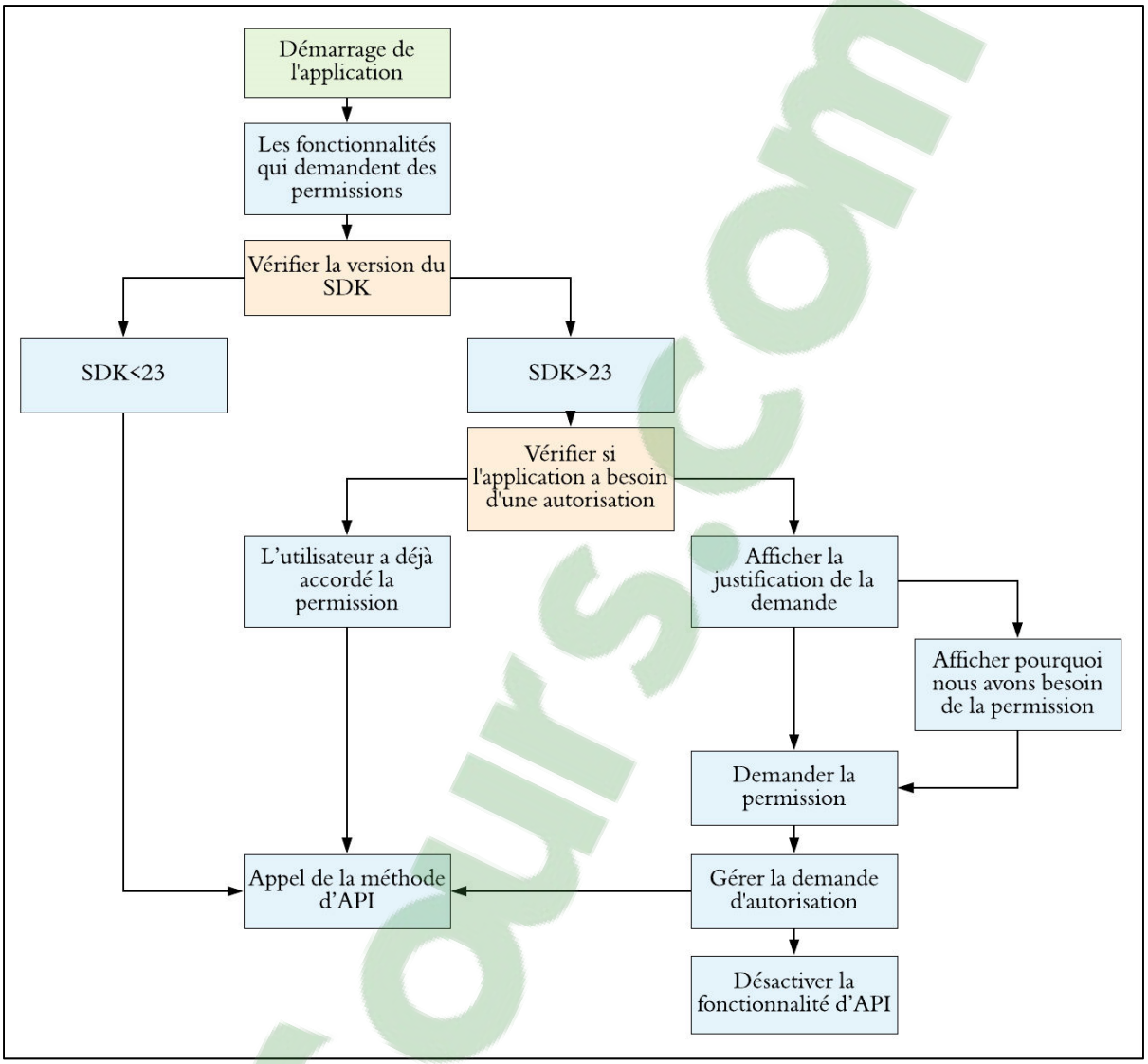


Figure 1.13 Le flux de travail de modèle de permissions actuel sur Android (Montemagno, 2015)

L'exécution des permissions avec Android Marshmallow

Une fois nous cliquons sur installer, l'application dans le Play Store ne sera jamais installée, un certain nombre d'autorisations doit être tout d'abord accepté avant que l'installation proprement dite puisse avoir lieu. Avantageusement, les autorisations ne seront plus demandées à ce stade.

Donc, si nous cliquons sur Installer dans le Google Play Store, l'application sera installée, après avoir accepté ces nombreuses autorisations en premier.

Si nous demandons une autorisation, une simple boîte de dialogue s'affiche, demandant aux utilisateurs d'autoriser la fonctionnalité spécifiée ou de la refuser comme le montre la Figure 1.14 dans la capture d'écran ci-dessous, l'application OneDrive tente d'accéder à vos fichiers multimédias. Si vous êtes d'accord, vous pouvez appuyer sur Autoriser.

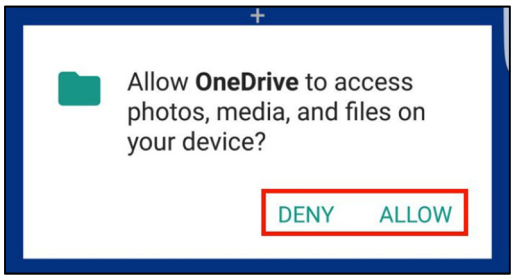


Figure 1.14 Boite de dialogue d'autorisation (Says, 2017)

Une fois vous appuyez sur refuser, l'application ne pourra pas voir et ne pourra donc pas sauvegarder vos fichiers multimédias sur le lecteur. Par défaut, le système se souviendra de votre choix. Donc, ultérieurement, lorsque vous lancerez l'application à nouveau, elle ne demandera pas la même autorisation à nouveau. Si vous avez refusé l'autorisation par erreur ou si vous souhaitez supprimer les autorisations d'application accordées, vous pouvez toujours le faire. Autrement dit, gérer les autorisations des applications sur vos appareils Android Marshmallow est faisable et facile à tout moment.

1.8 Récriture des applications Android

Plusieurs outils existants permettent de recouvrer le code source de java de l'application Android depuis le byte-code. La principale différence entre la machine virtuelle Dalvik et la machine virtuelle Java, est que la machine virtuelle Java est une technique de base de pile. En variante, le code de la machine virtuelle Dalvik est une technique de base de registres. Cette

différence causera un problème lors de la récupération ou de la conversion du code source, parce que ces mécanismes peuvent détruire plusieurs données. De ce fait, pour réécrire l'application, le fichier Dalvik Byte-code doit être converti en Java Byte-code et ceci en utilisant des outils existants, tels que dex2jar, ou réécrire le fichier Dalvik Byte-code directement. Le premier scénario peut être réalisé efficacement en utilisant le langage AspectJ. Pour le deuxième scénario, nous pouvons utiliser un byte-code intermédiaire appelé Smali.

1.8.1 La Programmation orientée aspect (AOP)

Développé dans les années 90, AOP est un paradigme de programmation établi chez Xerox PARC (Zhou Y. , Zhang, Jiang, & Freeh, Taming information-stealing smartphone applications (on android), 2011), son objectif est d'augmenter la modularité via l'utilisation d'Aspects, un Aspect étant une expression de « préoccupation transversale », c.-à-d. code de programme qui s'appuient sur ou affectent d'autres parties du système.

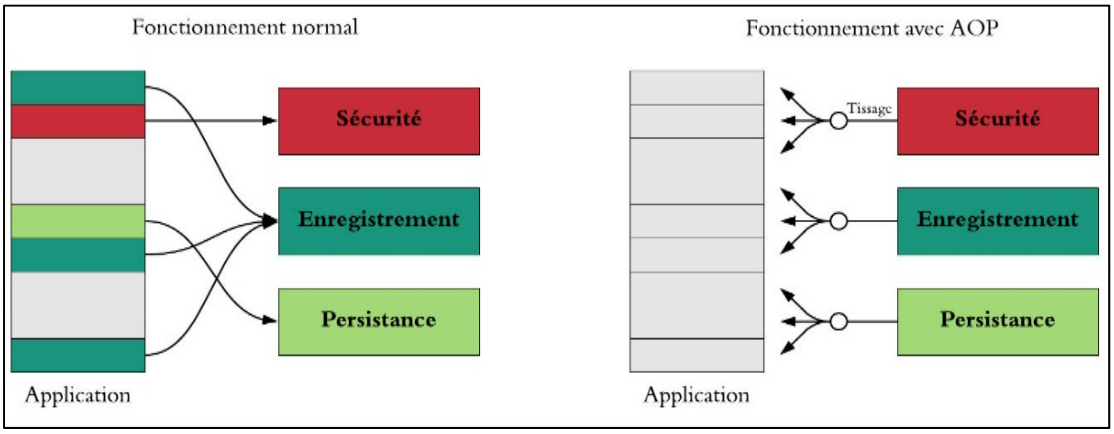


Figure 1.15 Fonctionnement de la programmation orienté aspect (Schenk, 2007)

Comme le démontre la Figure 1.15, AOP peut être très utile lors de la conception d'une application qui implique un contrôle sur ces fonctionnalités (Schenk, 2007). Les aspects sont mis en œuvre grâce à l'utilisation de ces trois concepts :

- **Joint-Point** : un point de jointure est un point identifiable au lors de l'exécution du programme, comme par ex : l'invocation d'une méthode.
- **PointCut** : Un pointCut s'agit d'une expression pour sélectionner un ensemble des points communs.
- **Advice** : un advice est une portion de code pouvant être associé à certains raccourcis. Par exemple, avec un conseil autour, nous décidons de procéder ou non avec une invocation de méthode.

AspectJ est une programmation orientée aspect (AOP) créé à Xerox PARC for Java (The AspectJ Team, 2003). Des aspects peuvent être utilisés afin de remanier des applications compilées sans avoir accès au code source grâce au compilateur AspectJ permettant de tisser dans le byte-code JVM.

Une application AspectJ comporte des classes et des interfaces Java, représentées dans le code de base, et les aspects représentés dans le code aspect. Le code aspect et le code de base sont tissés ensemble par un tisseur d'aspect, ce qui donne un code intermédiaire qui est ensuite compilé pour créer le byte-code java.

1.8.2 Le langage Smali

La machine virtuelle Dalvik utilise Smali (Gruver, 2017) comme langage d'assemblage basé sur une syntaxe connue sous le nom de jasmin/dedexer qui n'est qu'un assembleur de code octet Java. Ce langage utilise des descriptions ASCII des classes de la machine virtuelle Dalvik. Multiples sont les compilateurs / décompilateurs qui permettent la conversion du code octet Dalvik (dex) en code octet Smali, exemple ApkTool (Apktool, 2016).
ApkTool est un outil d'ingénierie inverse permettant de décoder les applications Android en code Smali. ApkTool sert à modifier ou à injecter du nouveau code dans l'application. Premièrement, cet outil décompile l'application en plusieurs composants, à savoir le fichier manifeste, les classes Java en langage Smali, les ressources, les bibliothèques et les ressources.

1.9 Les langages de spécifications des politiques

La définition d'un langage de spécification de politique permet d'exprimer les politiques de sécurité dans le but de vérifier l'accès et de gérer les risques. De plus, le langage de spécification des politiques de sécurité sera capable d'exprimer des politiques pour les conflits de ressources. Plusieurs langages de spécification des politiques de sécurités existent, nous pouvons citer les plus connus parmi eux comme le XACML (Parducci, Lockhart, & Levinson, 2017) et UMA (Maler, 2014).

1.9.1 XACML (eXtensible Access Control MarkupLanguage)

XACML est connu comme étant une spécification définissant un langage nécessaire dans le contrôle de l'accès, l'administration de la politique de sécurité et la circulation des règles des systèmes d'information. XACML est un outil souvent utilisé pour maintenir l'autorisation dans les architectures SOA (Parducci, Lockhart, & Levinson, 2017). XACML est une norme OASIS qui exprime les politiques de contrôle d'accès et de sécurité basées sur le format XML. Elle définit les composants suivants, comme le montre la figure 4.4, à savoir :

- **Point d'administration de politique (PAP) :** c'est le point où les politiques de contrôle d'accès sont lancées.
- **Point de décision de politique (PDP) :** Le PDP est le moteur de l'architecture. C'est l'emplacement exact où les politiques sont estimées et comparées contre les requêtes d'autorisation.
- **Point de renforcement de politique (PEP) :** Le PEP protège toute l'application ciblée.
- **Gestionnaire de contexte (CH) :** Ce composant contrôle et relie tous les autres composants.

- **Point d'information sur les politiques (PIP) :** Cette entité système agit comme une source de valeurs d'attributs.

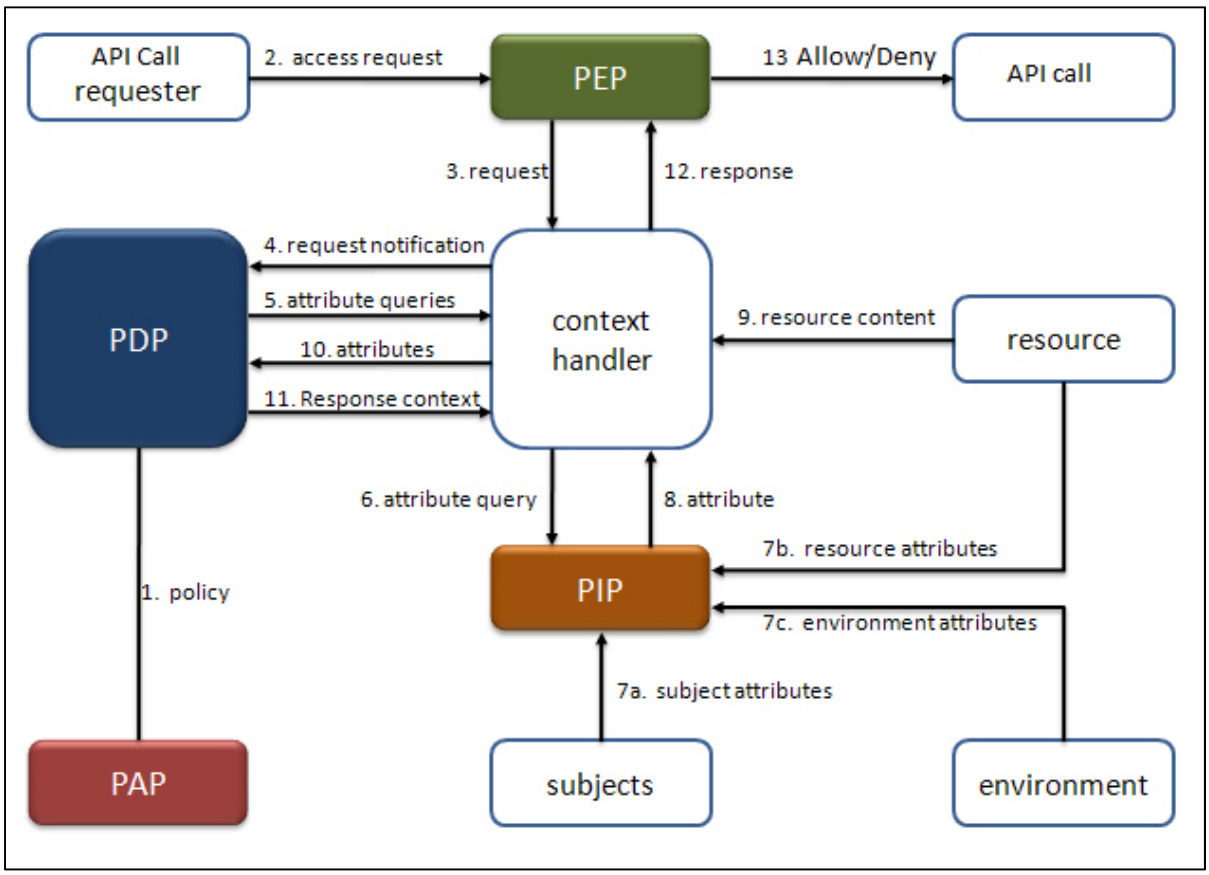


Figure 1.16 Architecture standard de l'implémentation d'une solution XACML (Parducci, Lockhart, & Levinson, 2017)

En outre, le langage de politique XACML (*Voir* Figure 1.16) fournit des normes pour les politiques de contrôle d'accès et le format de la demande-réponse aux contrôles d'accès. Les compilateurs XACML (XACML 2.0 et 3.0) compilent les politiques de XACML à la source Java. La compilation peut être effectuée de manière statique (hors ligne) ou dynamiquement pour charger les politiques en vue de leur exécution.

1.9.2 UMA (User-Managed Access)

UMA présente un protocole de gestion d'accès standard qui se base sur OAuth, qui permet à l'utilisateur de contrôler en ligne les autorisations d'accès à ces ressources protégées lorsque la requête est faite par un demandeur autonome. Cette initiative a des implications quant à la vie privée, le consentement des applications Web et l'internet des objets (IoT) (Maler, 2014).

UMA a les acteurs et l'architecture de base suivants, avec des entités qui s'alignent étroitement sur les entités OAuth principales (*Voir* Figure 1.17) :

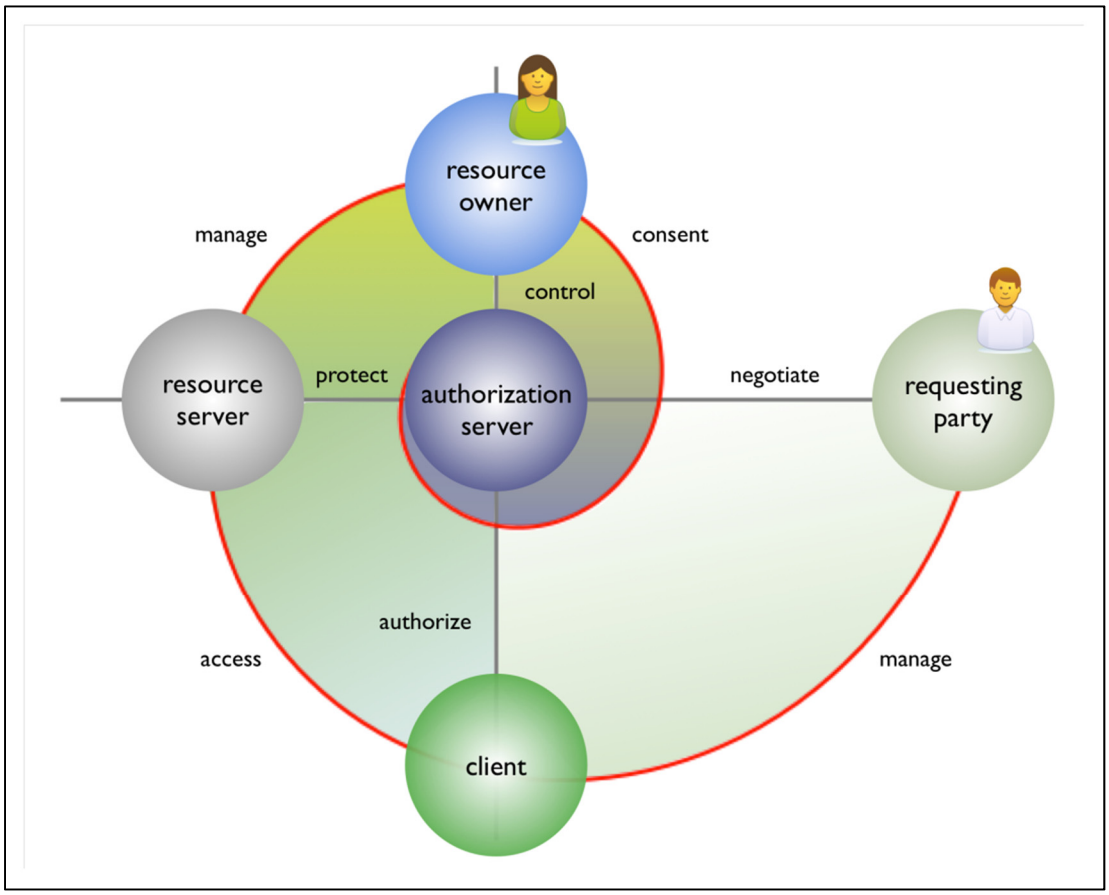


Figure 1.17 L'architecture de base de UMA (Maler, 2014)

Le standard pour la gestion des données personnelles et le consentement sont composés de :

- **Contexte** : Prendre la décision de partager de l'information au moment opportun.
- **Contrôle** : Partager uniquement ce qui est nécessaire.
- **Choix** : La possibilité de refuser et de revenir sur sa décision.
- **Respect** : Prendre en compte les souhaits et les préférences de chacun.

1.10 Conclusion

Dans ce chapitre, nous avons présenté une idée du système Android, dans lequel nous avons révélé le cadre de cette plateforme et ses composants. Les différentes vulnérabilités d'Android ont été abordées, pour clarifier la méthode d'attaque qui cible ce système d'exploitation. En outre, nous avons détaillé les techniques de sécurité Android utilisées pour atténuer les risques de fuite d'informations confidentielles. Par ailleurs, certains outils pour la réécriture et la modification de l'application Android ont également été présentés dans ce premier chapitre.

CHAPITRE 2

REVUE DE LITTÉRATURE

2.1 Introduction

Ce chapitre est consacré à la présentation des différents travaux connexes élaborés. Nous commençons par une discussion à propos des différentes solutions réalisées en présentant leurs principales contributions et limites. Par la suite, nous présentons les trois activités du domaine de la sécurité Android à savoir la détection, l'évaluation et l'atténuation des divulgations de la confidentialité d'informations privées.

2.2 Détection de divulgation d'informations privées

2.2.1 Attaques des logiciels malveillants

De nos jours, l'utilisation généralisée des smartphones a incité les institutions financières à considérer les applications bancaires mobiles comme un service nécessaire pour leurs clients. Les vulnérabilités des applications mobiles, les menaces signalées et les failles de sécurité détectées chaque année encouragent les utilisateurs et les institutions à faire attention à la sécurité de leurs services mobiles. Quelques études ont montré que les smartphones sont vulnérables à l'exposition de la vie privée en utilisant différentes attaques. Par exemple, CaptureMe (El-Serngawy & Talhi, 2015) est une attaque par capture d'écran pour les applications bancaires mobiles sur la plateforme Android . Comme l'illustre la Figure 2.1, l'attaque CaptureMe utilise différentes techniques connues pour prendre des captures d'écran et applique ,par la suite, une analyse OCR (Optical Character Recognition) très efficace à l'aide du moteur tesseract-ocr pour extraire les informations d'identification. En outre, ils explorent les mécanismes de protection possibles contre CaptureMe avec plus de 130 applications bancaires mobiles existent dans Google Play Store.

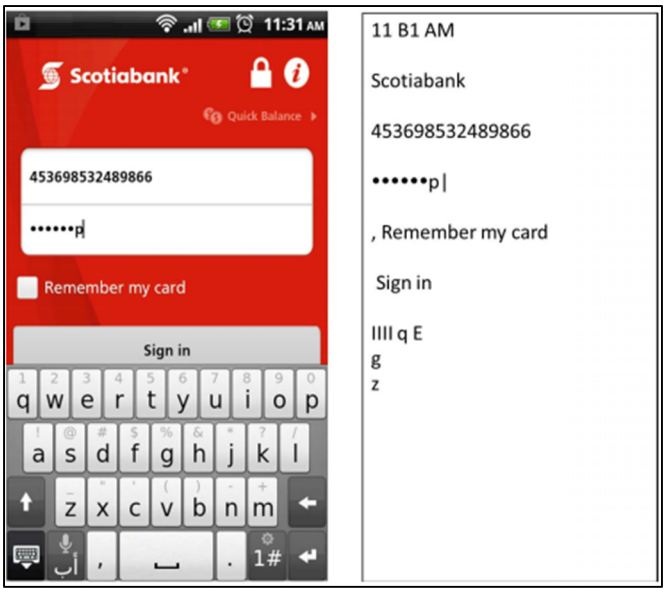


Figure 2.1 Les résultats de la phase d'analyse
par OCR (El-Serngawy & Talhi, 2015)

Dans une autre étude, Android / BadAccents (Felt, Finifter, Chin, Hanna, & Wagner, 2011) a présenté une autre technique utilisant un hameçonnage afin de voler des informations d'identification de compte bancaire. Il s'agit d'une attaque de cheval de Troie. En effet, la victime sera invitée à entrer ses données confidentielles dans une interface utilisateur graphique (GUI) ressemblant à celle d'une application bancaire mobile bénigne. Cependant, cette interface malveillante est conçue pour voler les informations privées de l'utilisateur. Les principaux composants du malware Android / BadAccents sont présentés dans la Figure 2.2.

L'attaquant bloque alors tous les SMS et les appels entrants. Si le message contient des commandes spéciales, l'attaquant va divulguer les informations à son serveur via HTTP et E-mail. Les informations d'identification du compte de messagerie sont masquées dans le code natif, ce qui rend l'identification difficile pour les approches d'analyses statiques qui fonctionnent sur le byte-code Dalvik. En outre, dans le composant Banking Trojan, une application malicieuse, qui contient des composants malveillants comme Android / BadAccents, est installée sur l'appareil de la victime, ce qui imitera l'application d'origine.

Dans une autre situation, le logiciel malveillant Android / BadAccents tente d'obtenir des privilèges d'administration des appareils Android à l'insu de l'utilisateur. Dans ce scénario, si l'attaquant était capable d'obtenir de tels privilèges, il pourra exécuter de nombreuses activités malveillantes, telles que le cryptage des données de l'utilisateur, le verrouillage de l'écran de l'appareil et/ou la réinitialisation de l'appareil.

(Rasthofer, Asrar, Huber, & Bodden, 2015) ont également discuté un autre type de malware appelé Tapjacking Attack. L'idée de base de ce type d'attaque est de montrer une interface utilisateur graphique sécurisée au premier plan et de masquer l'application malicieuse réelle en arrière-plan. Lors d'une telle attaque, toutes composants de l'interface utilisateur Android (UI) sont visés. De telles attaques peuvent être effectuées par des différentes manières. La principale raison étant de produire un composant d'interface utilisateur qui peut être superposé sur des applications sous-jacente.

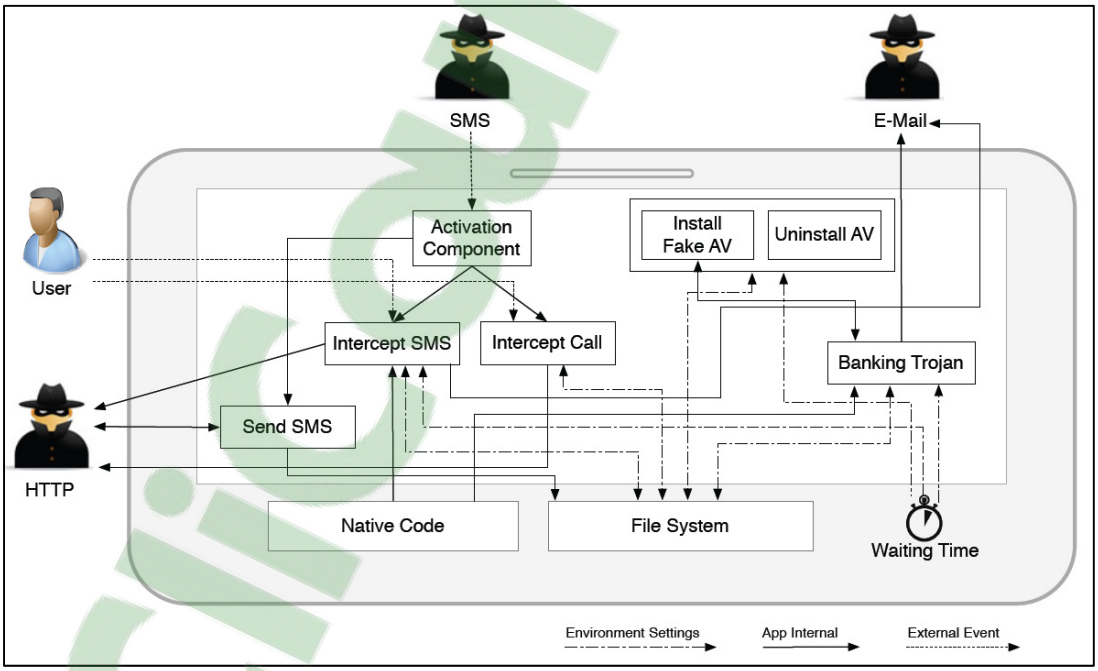


Figure 2.2 Le Malware Android / BadAccents
(Felt, Finifter, Chin, Hanna, & Wagner, 2011)

2.2.2 Détection de logiciels malveillants

(Enck, et al., 2010) ont proposé un outil d'analyse statique et dynamique appelé TaintDroid pour détecter et analyser en temps réel la divulgation des informations privées dans les applications Android. Cet outil permet, en effet, aux applications de suivre le flux d'informations avec les valeurs de registre en remplaçant la machine virtuelle Dalvik par une implémentation de suivi qui nécessite une modification spécifique du système d'exploitation sous-jacent.

(Mann & Starostin, A Framework for Static Detection of Privacy Leaks in Android Applications, 2012) ont proposé un cadre pour la détection statique des fuites de confidentialité dans les API Android et ont identifié certaines politiques de confidentialité critiques. Plusieurs autres travaux de recherches ont été également proposés pour détecter et analyser les fuites de confidentialité statiquement ou dynamiquement. Nous pouvons citer les travaux effectués par (Gibler, Crussell, Erickson, & Chen, June 13-15, 2012), (Kim, Yoon, Yi, & Shin, 2012), (Lu, Li, Wu, Lee, & Jiang, 2012) et (Yang, Yang, Zhang, Gu, Gu, & Wang, 2013). Ces approches peuvent analyser les applications pour des attaques potentielles sans avoir besoin d'exécuter l'application dans certaines situations. Elles peuvent aussi analyser des applications tout en accédant à des informations sensibles à l'exécution.

DroidForce (Rasthofer, Arzt, Lovat, & l, DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android, 2014) a fourni une technique pour détecter et signaler les flux de données en appliquant des politiques complexes centrées sur les données des applications Android. Cette approche permet aux utilisateurs de spécifier quelles sont les données accessibles sur leurs appareils. Les stratégies prises en charge par cette approche peuvent être modifiées dynamiquement au moment de l'exécution et ne nécessitent aucune modification du système d'exploitation sous-jacent. Par exemple, en utilisant ce système, nous pouvons appliquer une politique qui limite le nombre ou la fréquence des messages SMS qui peuvent être envoyés à un ensemble spécifique de numéros de téléphone. Cette politique permettra de détecter et d'empêcher l'une des principales falsifications qui volent l'argent de l'utilisateur via des messages SMS à tarif élevé.

Dans le même contexte de recherche, l'outil APKLancet proposé par (Yang, Li, Zhang, Li, Shu, & Gu, 2014) permet de diagnostiquer automatiquement l'application Android et de découvrir des fragments de code indésirables tel que des bibliothèques publicitaires ou un code malveillant. L'approche proposée ne nécessite pas de modification du système diagnostiqué. La Figure 2.3 montre les quatre étapes principales du flux de travail d'APKLancet.

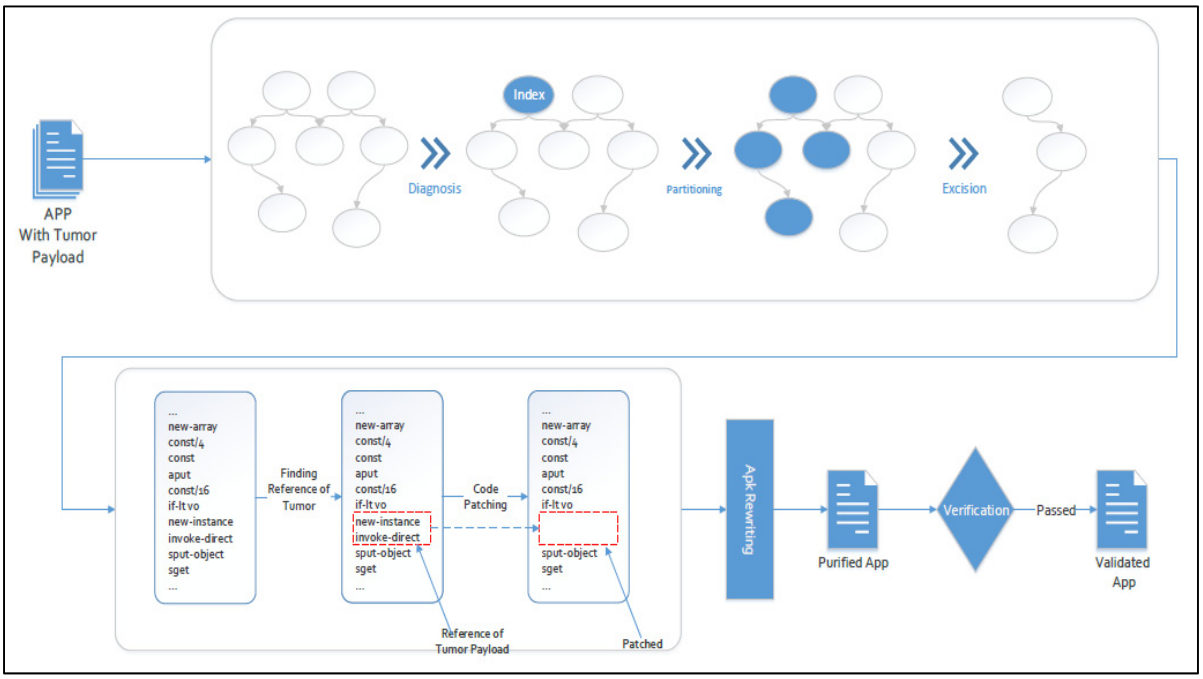


Figure 2.3 Processus de purification d'APK
(Yang, Li, Zhang, Li, Shu, & Gu, 2014)

La première étape du workflow ci-dessus consiste à diagnostiquer le fichier d'application. Ensuite, tout code suspect filtré, sera indexé et séparé du code bénin en utilisant l'analyse statique du programme. Au troisième stade, l'exactitude du flux de contrôle normal est prouvée après les dernières étapes. La dernière étape consiste à vérifier l'application purifiée en enregistrant dynamiquement le log affiché correspondant à chacun des événements, tels que l'accès au réseau, et en identifiant toute exception indésirable. En effet, APKLancet diagnostique les applications en appliquant une approche qui s'appuie sur la base de connaissances existante au lieu d'analyser les caractéristiques du code malveillant. Afin de

répondre à cette exigence, les auteurs ont résumé la mise en valeur du code malveillant connu et des bibliothèques tierces populaires dans une seule base de données. Enfin, APKLancet a été évalué et appliqué sur de vraies applications Android. Les résultats analysés ont indiqué que APKLancet est capable de purifier les codes bénins des codes malveillants indésirables.

2.2.3 Détection du reconditionnement des applications

De nos jours, le reconditionnement de byte-code Dalvik qui a été utilisé dans la machine virtuelle Dalvik n'est pas un problème pour de nombreux développeurs d'applications Android. Par conséquent, le reconditionnement des applications Android est devenu un problème sérieux qui menace la vie privée des utilisateurs et la propriété intellectuelle des développeurs. Afin de détecter les applications reconditionnées et d'empêcher leur propagation, de nombreux chercheurs ont proposé des algorithmes de détection du reconditionnement pour les applications Android. Par exemple (Crussell, Gibler, & Chen, 2012) ont proposé un outil appelé DNADroid qui compare l'architecture de dépendance du programme pour deux applications Android afin de détecter toute similarité ou clonage.

(Zhou, Zhou, Jiang, & Ning, 2012) ont appliqué une technique de hachage spécialisée, appelée 'hachage flou' développée par l'outil DroidMOSS. Comme le montre la Figure 2.4, cet outil peut détecter systématiquement une application Android reconditionnée en divisant les instructions du programme en des petites unités. Par la suite, une valeur de hachage calculée sera pour chaque unité locale au lieu de calculer un hachage sur tout l'ensemble d'instructions du programme.

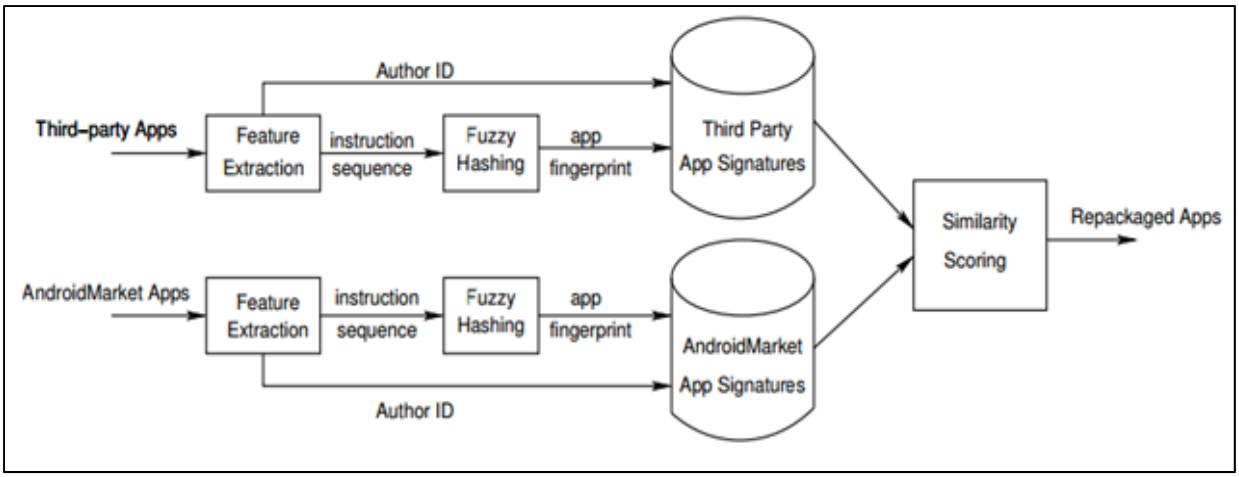


Figure 2.4 Principes de génération d'empreinte par Droid MOSS (Zhou Y. , Zhang, Jiang, & Freeh, Taming information-stealing smartphone applications (on android), 2011)

Récemment (Zhou, Zhang, & Jiang, AppInk: watermarking android apps for repackaging deterrence, 2013) ont développé un prototype appelé Appink qui applique un mécanisme de filigranage pour empêcher les applications Android de reconditionner les attaques. La figure 3.4 montre l'architecture globale d'AppInk. L'architecture a produit un concept nommé manifest-app pour rendre le mécanisme de filigranage applicable lors du développement des applications. L'application Manifest encapsule automatiquement une série d'opérations d'entrée pour l'application cible sans demander l'intervention de l'utilisateur. Plus précisément, l'architecture est principalement composée de :

(1) Génération de code de filigranage : consiste à encoder la valeur de filigrane spécifiée par le développeur en une structure de graphe spéciale avant de transformer ce dernier en code de filigranage.

(2) Génération d'une application de manifest : surveille l'exécution du filigrane incorporé en envoyant les entrées de l'utilisateur prédéterminées à l'application.

(3) Instrumentation de code source : responsable de l'injection du code de filigranage dans les opérations d'entrée identifiées, et de leurs gestionnaires correspondants en fonction du code source original de l'application examinée.

(4) Reconnaissance de filigrane : ce composant est responsable de la correspondance de la structure de filigranage potentiel en utilisant un motif spécial. Si l'outil AppInk était

capable de trouver une telle structure de filigranage, alors le processus de génération de prose inversée du filigrane récupèrera le code de filigranage et vérifiera son originalité. En conséquence, l'outil AppInk peut effectivement être utilisé pour détecter les menaces de reconditionnement communes dans l'application (*Voir* Figure 2.5).

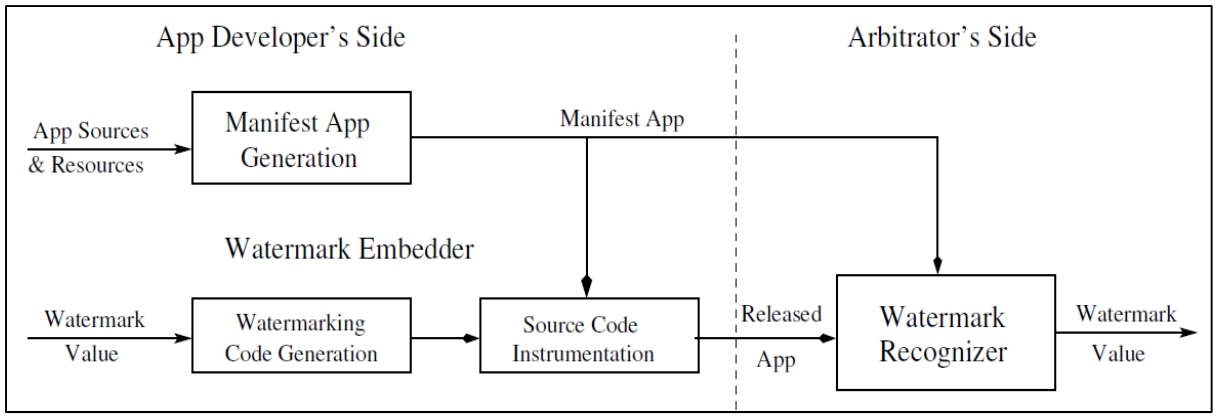


Figure 2.5 L'architecture global d'AppInk (Zhou, Zhang, & Jiang, AppInk: watermarking android apps for repackaging deterrence, 2013)

2.3 Atténuation de divulgation d'informations privées

2.3.1 Modification de la plateforme d'Android

De nombreux chercheurs ont introduit des cadres de sécurité qui reposent sur la modification de la plateforme Android open-source pour développer des versions personnalisées d'Android. Les chercheurs (Nauman, Khan, & Zhang, 2010) ont développé une pile logicielle Android appelée Apex pour améliorer le contrôle de la sécurité et de la confidentialité sur Android en permettant des révocations d'autorisations dynamiques.

TISSA (Zhou Y. , Zhang, Jiang, & Freeh, Taming Information-Stealing Smartphone Applications (on Android), 2011) est un modèle de confidentialité conçu et mis en œuvre pour améliorer la sécurité de la plateforme Android en permettant aux utilisateurs de contrôler facilement les types d'informations privées disponibles pour une application. De plus, les auteurs (Hornyack, Han, Jung, Schechter, & Wetherall, 2011) ont fourni l'outil

AppFince qui applique une solution basée sur les flux d'informations. Cette solution nécessite des modifications sur la pile de stockage des applications pour appliquer les politiques de confidentialité afin d'atténuer les fuites d'informations privées.

Plus récemment (Nauman, Khan, & Zhang, 2010) ont proposé un système de sécurité plus précis pour développer les permissions standard du système d'exploitation Android. Ce travail s'appuie sur le prototype TaintDroid (Enck, et al., 2010) afin de détecter et rapporter les flux de données pour les applications Android en appliquant des politiques complexes qui ont été construites dans des conditions spéciales. Cette approche permet aux utilisateurs d'interdire certaines actions telles que la lecture de film plus de deux fois même si plusieurs copies identiques ont été créées. La Figure 2.6 montre l'architecture de cette approche qui se base deux principaux composants :

- Le moniteur du système Android comprend également deux autres composants, à savoir le point d'application de la politique (PEP) et le point de décision politique (PDP). Le PEP s'exécute en tant que service système qui a révisé les évènements en tant que stratégies d'application tandis que le PDP prend la décision d'autoriser ou refuser un évènement.
- La deuxième composante représente l'application du gestionnaire de sécurité qui est principalement une interface utilisateur permettant de gérer des règles de confidentialités.

L'application du contrôle de sécurité dans ce système est déployée en tant qu'une partie interne dans le système Android. L'utilisateur n'est pas capable de la désinstaller. Enfin, le système peut aider à se protéger contre certaines applications malveillantes en appliquant des stratégies telles que l'envoi de la liste des contacts d'utilisateur et la limitation du nombre de SMS pouvant être envoyés par heure.

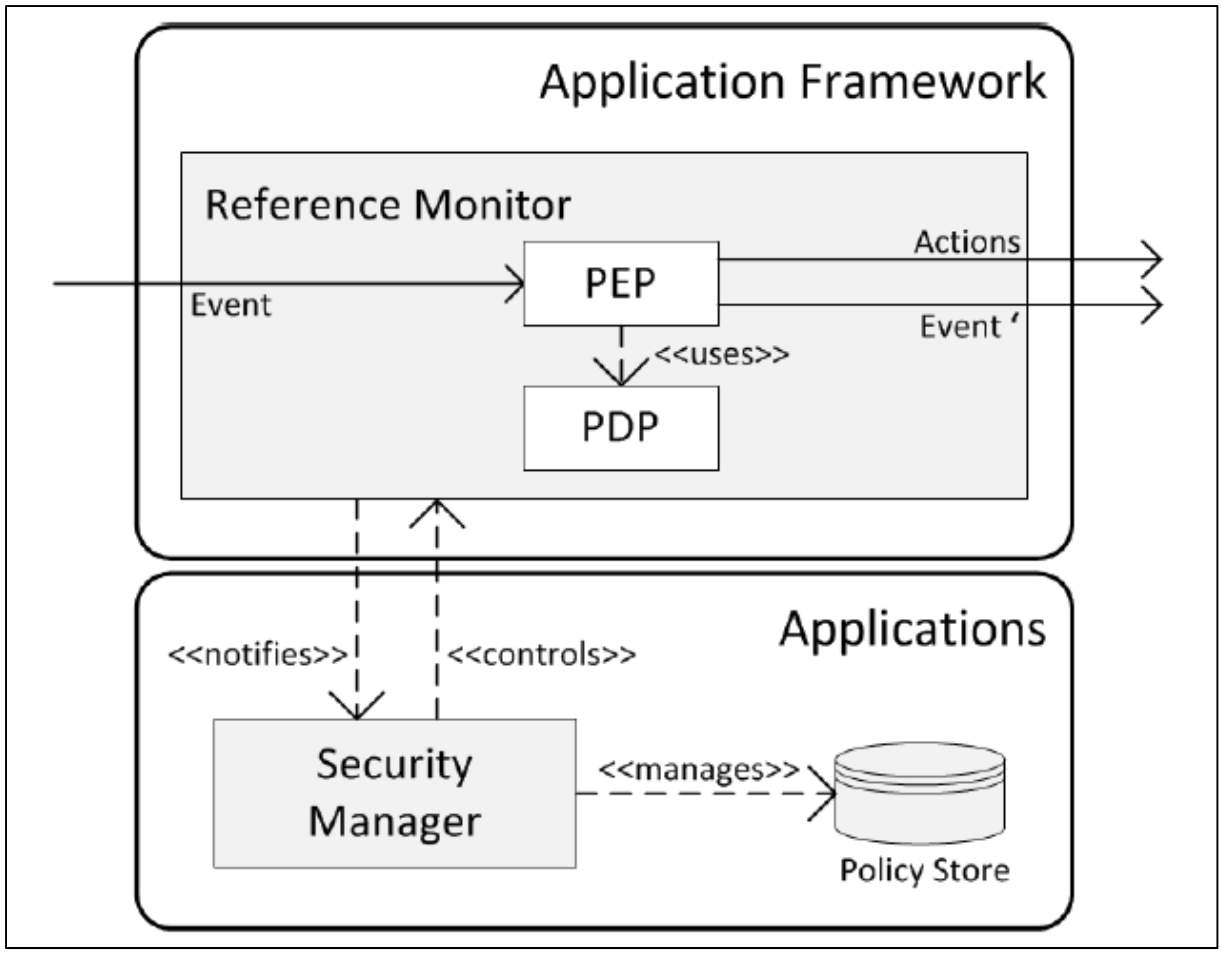


Figure 2.6 Architecture de haut niveau du système
(Cooper, Shahriar, & Haddad, 2014)

En résumé (Felt, Finifter, Chin, Hanna, & Wagner, 2011) ont appliqué des politiques plus précises pour améliorer les permissions standards du système d'exploitation Android afin d'atténuer les fuites de confidentialité causées par certaines applications et utilisateurs malveillants. La principale limitation de ce travail est la nécessité d'une modification dédiée sur le système d'exploitation Android.

SecureDroid (Arena, Catania, & Torre, 2013) ont abordé la question du contrôle des politiques de sécurité pendant que les applications s'exécutent dans l'environnement Android. Lors de l'installation d'une application, Android permet à l'utilisateur d'autoriser une application d'utiliser certaines fonctionnalités du système. Ce travail a introduit une extension

du cadre de sécurité d'Android afin d'améliorer le contrôle d'autorisation standard fourni par le système d'exploitation. Pour atteindre cet objectif, ils ont introduit un nouveau mécanisme de contrôle ajoutant de la granularité et de la flexibilité. En plus, leur cadre de politique est basé sur la personnalisation du standard XACML (Parducci, Lockhart, & Levinson, 2017) afin de l'adapter sur le système Android. En outre, ils ont fourni la possibilité d'ajouter ou de modifier une politique via un service système dédié. Cela permettra aux utilisateurs de spécifier les autorisations à accorder ou à refuser pour chacun des contextes définis.

Les principaux inconvénients de la modification de la plateforme Android

Dans cette partie, nous allons présenter les principaux inconvénients de la modification de la plateforme Android à savoir :

- Nécessite la création d'un micro logiciel personnalisé et d'un code de plateforme : Bien qu'il y ait certains avantages à avoir une plateforme personnalisée telle que l'accessibilité pour des fonctionnalités supplémentaires, cela est toujours considéré comme un inconvénient parce que les utilisateurs peuvent perdre leur garantie ou encore certaines fonctionnalités de l'appareil ne fonctionnent plus.
- Nécessite le routage de l'appareil pour obtenir des privilèges administratifs : Ce service permet aux utilisateurs d'exécuter des applications spéciales nécessitant un accès root sur l'appareil, d'un autre côté, cela pourrait réduire la sécurité de l'utilisateur.
- Différents appareils et différentes versions de la plateforme : Ce problème empêche certaines fonctionnalités des applications de s'exécuter convenablement.
- Les applications à sécuriser sont limitées aux mécanismes de contrôle pris en charge par le système d'exploitation Android customisé.
- La modification de la plateforme Android rendra difficile son déploiement sur des millions d'appareils mobiles.

2.3.2 Récriture des applications Android

Récemment, il y a des séries de travaux de recherche et des mécanismes de sécurité proposés qui ne nécessitent pas la modification de la microprogrammation. Dans ce contexte, de

nombreux travaux de recherches ont été élaboré pour réécrire les applications Android afin d'imposer certaines politiques de sécurité. Par exemple, (Xu, Saıdi, & Anderson, 2012) ont proposé une approche concurrente qui réécrit l'application Android en sandbox avec des méthodes API natives importantes et surveille le comportement de l'application pour détecter toute violation de sécurité. (Jeon, et al., 2012) ont utilisé l'approche de réécriture de byte-code de Dalvik pour s'interposer sur toutes les invocations des méthodes d'API afin d'injecter les politiques de sécurité désirées. (Davis, Sanders, Khodaverdian, & Chen, 2012) ont conçu et mis en œuvre un cadre de réécriture appelé I-bras-droid. Ce prototype est capable d'identifier un ensemble de méthodes API sensibles à la sécurité et de spécifier leurs politiques de sécurité, qui peuvent être modifiées pour satisfaire les besoins de sécurité de chaque application Android.

(Davis & Chen, Retroskeleton: Retrofitting android apps, 2013) ont fourni un autre prototype de réécriture appelé RetroSkeleton basé sur leur travail précédent I-arm-droid afin d'insérer, supprimer ou modifier le comportement de l'application Android sans modifier la plateforme Android. Cette approche de réécriture prend en charge la transformation de diverses stratégies utiles telles que le contrôle d'accès flexible et précis et améliore certaines politiques de sécurité des communications réseau.

L'idée (Styp-Rekowsky, Gerling, Backes, & Hammer, 2013) fournit une approche de surveillance de référence en ligne pour appliquer les politiques de sécurité à l'exécution, en remplaçant les références aux méthodes de sécurité dans la représentation interne de la machine virtuelle Dalvik. Les chercheurs (Zhang, Ahlawat, & Du, 2013) ont proposé un prototype appelé AFrame. Ce dernier utilise l'approche de réécriture de byte-code pour isoler les publicités des applications Android afin de séparer le code tiers non approuvé des applications. En outre, d'autres chercheurs (Pearce, Felt, Nunez, & Wagner, 2012) et (Shekhar, Dietz, & Wallach, 2012) ont proposé deux autres systèmes appelés AdDroid et Adsplit. Ils ont également été utilisés pour séparer la publicité des smartphones des applications.

(Zhang & Yin, 2014) ont déployé un mécanisme d'application de politique sensible au contexte pour atténuer les fuites de confidentialité dans les applications Android. Ce mécanisme applique la politique de confidentialité en fonction des préférences de l'utilisateur. De plus, ils ont mis en œuvre un prototype appelé Capper (Figure 2.7). En utilisant ce système, lorsqu'un utilisateur tente d'installer une application Android, le moteur de réécriture byte-code BRIFT réécrit le programme de cette application en insérant de manière sélective le code d'instrumentation le long des tranches de propagation pour surveiller et prévenir toute fuite d'information. En résumé, pour réécrire une application donnée par Capper, ils convertissent d'abord le fichier Dalvik DEX en byte-code Java en utilisant l'outil dex2jar (Zhang & Yin, 2014). Ensuite, ils ont utilisé le Framework d'optimisation de byte-code Java appelé Soot (Zhang & Yin, 2014) pour traduire le byte-code Java en IR intermédiaire afin de réaliser une analyse de flux de données statique et une instrumentation byte-code. Pour l'instrumentation statique, ils créent une signature pour chaque entité dans les tranches de propagation de la souillure individuellement. Ensuite, ils optimisent le code d'instrumentation ajouté pour supprimer tout byte-code redondant. Enfin, ils convertissent le byte-code de réécriture en un nouveau paquet avec les anciennes ressources pour créer un nouveau fichier apk.

La solution proposée dans ce système consiste en deux techniques d'activation suivantes :

- La première technique est la réécriture byte-code pour le contrôle du flux d'information : Pour atteindre cet objectif, ils effectuent d'abord une analyse statique de flux de données pour mesurer la tranche totale de programme utilisés dans la propagation de la contamination. Ensuite, ils insèrent des instructions de byte-code le long des tranches de programme pour garder trace de la propagation de la souillure à l'exécution. La principale contribution de cette solution, c'est qu'elle ne nécessite aucun changement dans le système Android et n'entraîne qu'un impact minimal sur les performances.
- La deuxième technique est la création d'une application qui contient une politique contextuelle permettant à l'utilisateur d'accepter ou de refuser certains flux d'informations dans un contexte spécifique. En outre, ils ont évalué le prototype Capper sur 4723 applications Android réelles. Les résultats obtenus ont prouvé l'utilité et l'efficacité de ce

prototype pour atténuer les fuites de confidentialité dans la plupart des applications Android actuelles.

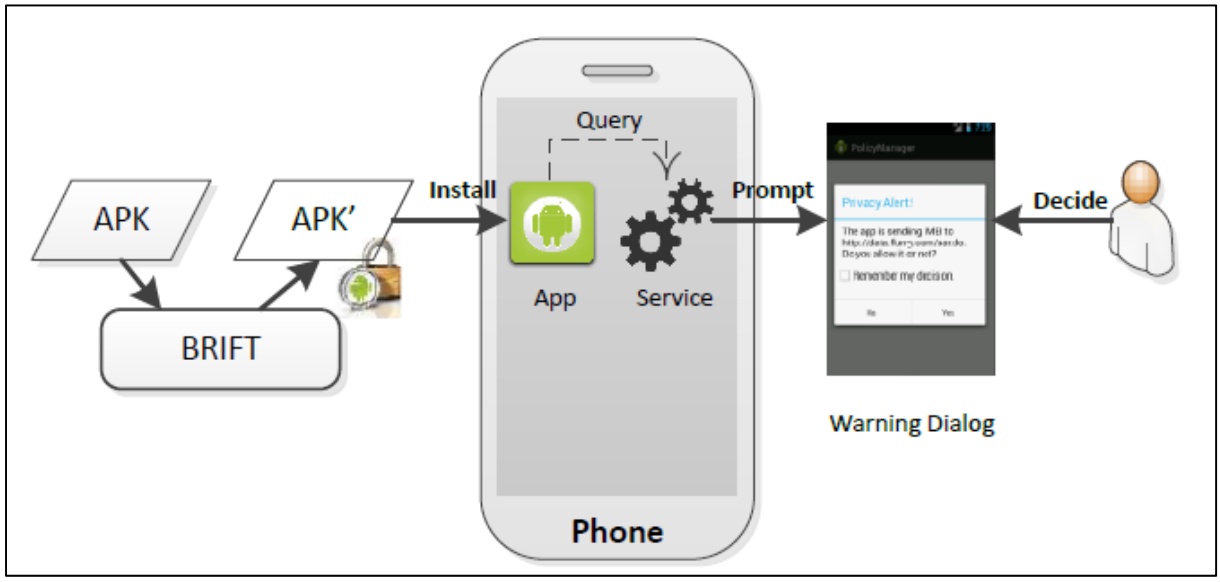


Figure 2.7 Architecture de Capper (Zhang & Yin, 2014)

Bien que cette approche fournisse une solution utile qui peut être utilisée pour bloquer les fuites de confidentialité dans la plupart des applications Android contre les programmes malveillants, les pirates peuvent toujours trouver des moyens pour contourner leur confinement. Par exemple, les logiciels malveillants peuvent filtrer les informations privées via des canaux secondaires (tels que les flux implicites et les canaux de synchronisation). De plus, la technique proposée ne peut pas gérer les composants natifs et les fonctions Java prédéfinis d'une manière générale.

WeaveDroid (Pétrod, 2017) a fourni un cadre pour intégrer les aspects AspectJ dans une application Android. Le cadre prend deux entrées au début : APK et un ensemble d'aspects qui seront tissés dans le fichier APK. Le processus de tissage sera effectué sur le smartphone Android. Comme le montre la Figure 2.8, le processus WeaveDroid comprend cinq étapes :

1. Conversion de l'APK d'entrée du format DVM byte-code au format JVM. Ce processus utilise la bibliothèque dex2jar.

2. Tissage des aspects d'entrée avec le byte-code JVM.
3. Utilise l'outil dx pour convertir le byte-code JVM au format DVM byte-code.
4. Intégration de byte-code modifié avec l'entrée APK.
5. Signature de l'APK modifié

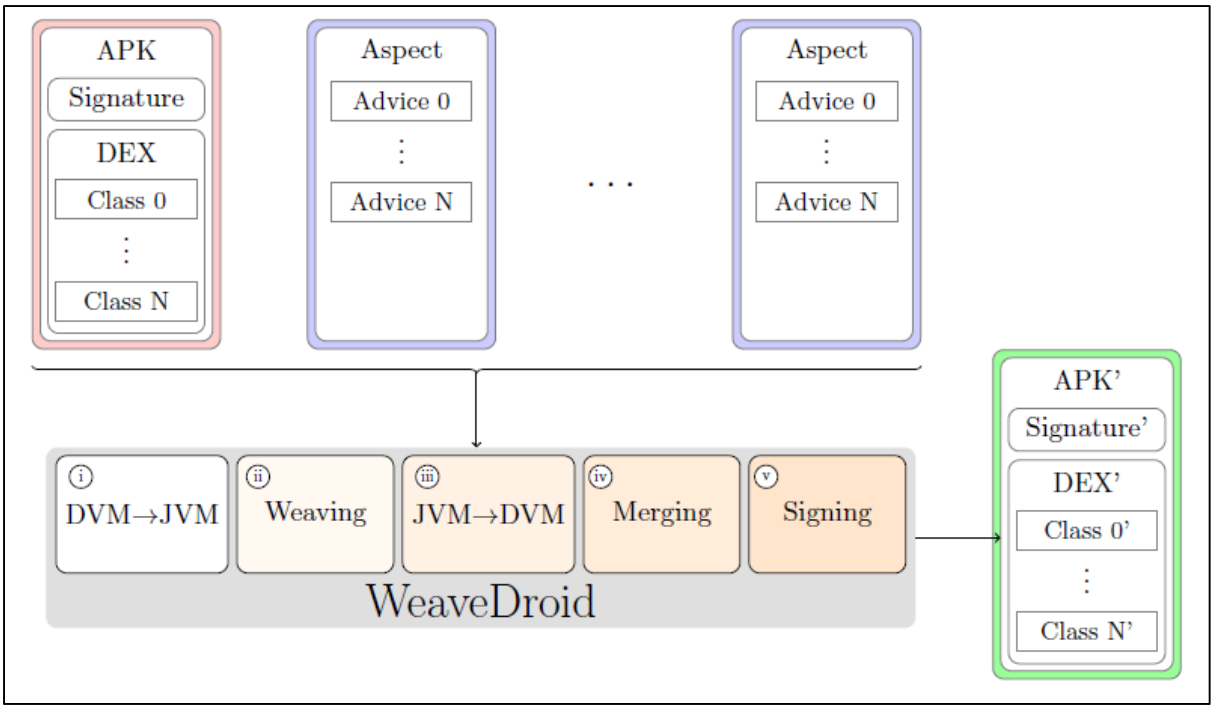


Figure 2.8 Architecture de Weave Droid (Pétrod, 2017)

Récemment , (Boudar, 2016) ont proposé une approche qui comporte deux axes principaux, comme le montre la Figure 2.9. Le premier axe est un Framework pour réécrire des applications tout en introduisant des portions de code nécessaires pour pouvoir communiquer avec le contrôleur de l'application. Le second axe est le contrôleur de la solution proposée. Ce dernier permet la gestion de toutes les applications s'exécutant sur le périphérique et contrôle les méthodes API demandées par ces applications.

L'utilisateur doit impérativement introduire tout un ensemble de politiques de sécurité sur lesquelles le contrôle requis par le contrôleur de l'application est principalement basé.

En effet, ces politiques accordent une définition des différents scénarios malveillants en fonction de diverses conditions comme l'heure, la position du GPS, les méthodes de l'API, le niveau de la batterie etc. Le contrôleur procure une voie efficace et simple dans la définition de politique de sécurité.

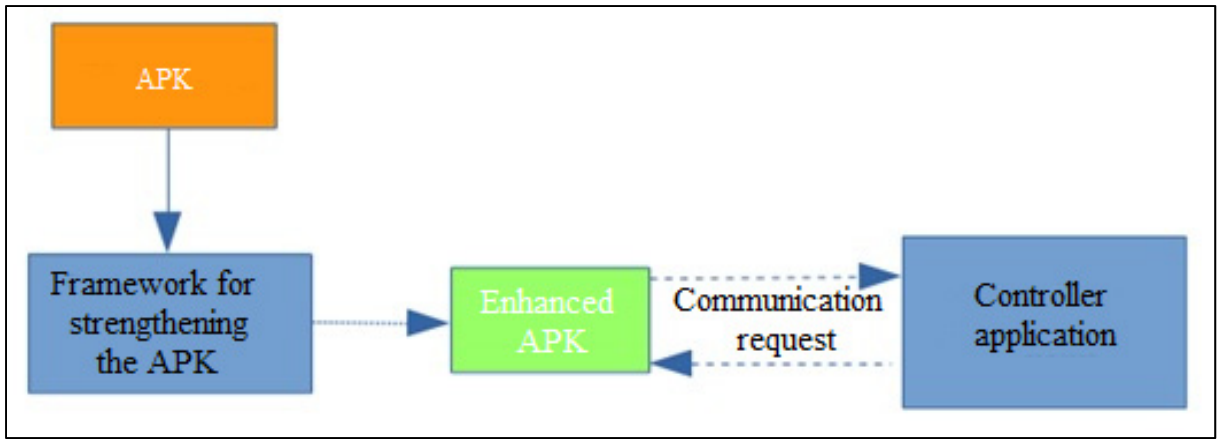


Figure 2.9 Aperçue générale de la solution proposée (Boudar, 2016)

Les résultats expérimentaux du Tableau 2.1 montrent que l'approche développée fournit un moyen efficace pour contrôler les appels API, ainsi que la consommation des ressources de l'appareil et le taux de contrôle. Le cadre de réécriture fournit un pourcentage de réussite de 99,78%, ce qui est assez élevé par rapport d'autres solutions étudiées. En outre, le temps demandé pour réécrire une application a été mesuré, il est de 51s. De plus, la taille moyenne ajoutée par la réécriture du cadre a également été calculée.

Tableau 2.1 Les taux de réussite des applications réécrites (Boudar, 2016)

Type des applications	**Nombre des applications**	**Taux de succès de réécriture**
Application du Google Play	950	99,78% (948)
Applications malicieuses	450	99,77% (449)
Toutes les applications	1400	99.78% (1397)

Le contrôleur d'application nécessite un examen sur tous les appels des méthodes d'API. Le temps d'analyse était mesuré de 20% du temps d'invocation de l'API avant d'appliquer le contrôle. Le moniteur présente une application qui contrôle et empêche les tentatives des applications malicieuses d'accéder aux APIs. En outre, cette approche est capable d'empêcher les programmes malveillants à exécuter des fonctions natives parce qu'ils ne sont pas capables de toucher les fonctions d'APIs. La création des politiques de sécurité qui contrôlent ces API peut jouer un protecteur solide contre ces attaques.

2.4 Discussion

En se basant sur le contenu des approches examinées, les techniques d'analyse des logiciels malveillants utilisées sont toujours limitées. Ces approches n'arrivent à détecter que les attaques sur les flux de données intra-procédurales. Les logiciels malveillants peuvent filtrer les informations privées à travers des canaux secondaires vue que ces techniques ne sont pas capables de traiter les composants natifs. En outre, un pirate assez spécialisé dans le développement de logiciels malveillants arrive toujours à trouver des moyens pour pénétrer le confinement de ces approches. Du coup, des nombreux défis sont posés afin de déclencher un logiciel malveillant, ce qui montre la nécessité de poursuivre les recherches dans ce

domaine. Par conséquent, nous discuterons des politiques de sécurités qui peuvent être appliquées par notre solution proposée. Les approches examinées sont classifiées dans le Tableau 2.2.

Tableau 2.2 Taxonomie des approches examinées

Système/Approche	Méthodologies utilisées	Modification requise	Supporte des politiques de sécurité dépendantes au contexte	Fournis un langage de politique
TaitDroid (Enck, et al., 2010)	Analyse dynamique	Plateforme Android	Non	Non
Appink (Zhou, Zhang, & Jiang, AppInk: watermarking android apps for repackaging deterrence, 2013)	Watermarking	Application	Non	Non
Apex (Nauman, Khan, & Zhang, 2010)	Renforcement des politiques	Plateforme Android	Non	Non
Tissa (Zhou Y. , Zhang, Jiang, & Freeh, Taming Information-Stealing Smartphone Applications (on Android), 2011)	Contrôle des accès des politiques	Plateforme Android	Non	Non
AppFince (Hornyack, Han, Jung, Schechter, & Wetherall, 2011)	Analyse dynamique et Contrôle des accès des politiques	Plateforme Android	Non	Non
Aurasium (Xu, Saıdi, & Anderson, 2012)	Réécriture de byte-code java	Application	Non	Non
I-arm-droid (Davis, Sanders, Khodaverdian, & Chen, 2012)	Réécriture du Dalvik code java	Application	Non	Non
AFrame (Zhang, Ahlawat, & Du, 2013)	Isoler la publicité	Application	Non	Non
Capper (Zhang & Yin, 2014)	Réécriture de byte-code java	Application	Non	Non
SecureDroid (Arena, Catania, & Torre, 2013)	Renforcement des politiques	Plateforme Android	Oui	Oui
Weave Droid (El-Harake, Falcone, Jerad, Langet, & Mamlouk, 2014)	Isoler la publicité	Application	Non	Non
Oussama (Boudar, 2016)	Renforcement des politiques	Application	Oui	Non

Tout d'abord, la plupart des logiciels malveillants existants interceptent les messages SMS entrants (par exemple, pour bloquer les informations d'avertissement ou de facturation). Ce problème peut être dû au manque de contrôle précis sur les permissions autorisant l'accès aux APIs associées (par exemple, RECEIVE_SMS). Dans ce cas, nous pouvons proposer une politique de sécurité qui contrôle les autorisations d'accès à l'API SMS. De plus, nous pouvons offrir des politiques centrées sur les données qui contrôlent l'accès à l'emplacement des données privées de l'utilisateur telles que la liste des contacts, les courriels, emplacement, etc. pour empêcher les tentatives d'envoi des requêtes par les logiciels malveillants dans le but d'accéder aux données sans avoir la permission.

En second lieu, les logiciels malveillants visent à cacher l'activité malveillante dans les composants natifs. Si les informations d'authentification de l'utilisateur de l'application bancaire sont cachées dans le code natif, la détection sera difficile pour les approches d'analyse statique existantes. Par conséquent, nous pouvons appliquer des politiques de sécurité pour contrôler et empêcher de telles activités à accéder à ces fonctions natives.

Ensuite, nous envisageons de résoudre le conflit des ressources entre les applications. Cela signifie qu'une application installée accédant à une certaine ressource comme le capteur de GPS peut envoyer l'emplacement actuel de l'utilisateur à la compagnie d'assurance. Par contre, une application malveillante accédant à la même ressource peut envoyer des fausses données de l'emplacement. Dans ce cas, nous pourrions être en mesure d'appliquer des politiques de sécurité pour contrôler le conflit des ressources utilisées dans le même temps.

De plus, les logiciels malveillants enferment les exploits au niveau de la plateforme Android pour augmenter leurs privilèges. Par exemple, l'attaquant tente d'obtenir des privilèges d'administration du smartphone Android de l'utilisateur. Dans ce cas, nous pouvons contrôler et appliquer une politique de sécurité sur l'autorisation d'accès à l'API d'administration de périphérique Android (BIND_DEVICE_ADMIN_PERMISSION).

Enfin, d'autres types de logiciels malveillants peuvent être traités, tels que les attaques qui génèrent un composant d'interface graphique de l'application de l'utilisateur. Ils peuvent être superposés sur des applications et acheminer des gestes tactiles vers l'application sous-jacente. Dans ce cas, nous devons appliquer une politique sur les APIs qui accèdent à l'interface graphique d'application de l'utilisateur (UI) en contrôlant et en appliquant les politiques sur la communication entre les applications. Cette politique peut être considérée comme l'un des principaux objectifs de recherche.

2.5 Conclusion

Dans le deuxième chapitre, nous avons exposé une revue littérature basée sur les travaux utilisés afin de sécuriser le système d'exploitation Android. Ces approches sont divisées sur deux techniques : une première technique qui se base sur la customisation du système Android original et ajouter des mécanismes de sécurité. La deuxième technique consiste à réécrire les applications Android pour appliquer des stratégies de sécurité sans modifier le système d'exploitation.

Les travaux présentés ont démontré une efficacité significative pour défendre la confidentialité des données de l'utilisateur. Cependant, ils présentent certains inconvénients, tels que l'impossibilité de contrôler l'exécution de code natif et le conflit des ressources partagées entre plusieurs applications qui peuvent contenir des mécanismes malveillants. Dans le chapitre suivant, nous présentons notre propre solution qui surmontera la plupart des limitations des approches étudiées à savoir la résolution du conflit des ressources partagées entre les applications. La création d'un langage de spécification des politiques de sécurité permet aux utilisateurs d'exprimer des politiques textuelles en utilisant des contrôles sécuritaires de l'application.

CHAPITRE 3

ARCHITECTURE ET IMPLÉMENTATION

3.1 Introduction

Dans ce présent chapitre, nous allons proposer et développer une solution nécessaire qui fait l'objet de notre étude. Notre solution étant centralisée, elle est capable d'appliquer des politiques de sécurité flexibles et déclaratives. Notre solution est composée de deux principales contributions à savoir l'approche de réécriture et le contrôleur centralisé des applications Android basé sur le langage de spécification des politiques de sécurité.

3.2 Solution proposée

Notre solution, vise à fournir un système de contrôle d'accès centralisé. En particulier, nous nous concentrons sur un langage de contrôle d'accès des politiques qui permet à l'utilisateur de définir ses propres politiques dans le but de contrôler les applications installées dans son smartphone Android d'une façon automatique et selon le contexte actuel.

Rappelons que pour maintenir la sécurité du système et des utilisateurs, Android contrôle l'utilisation des APIs permettant l'accès des applications aux données privées de l'utilisateur et aux fonctionnalités critiques du smartphone. Selon la sensibilité de la zone, le système peut automatiquement accorder l'autorisation de l'utilisation de ces APIs, comme il peut demander à l'utilisateur du Smartphone d'approuver la demande.

Les développeurs ont la responsabilité de spécifier les permissions demandées pour accéder aux APIs dans chaque application Android, de sorte qu'ils accordent l'accès aux fonctionnalités de l'appareil, aux données personnelles de l'utilisateur de smartphone et à diverses propriétés sensibles procurées par le système Android. Nous pouvons citer l'exemple de l'accès aux répertoires, aux données GPS, aux photos et au répertoire

téléphonique, l'envoi des SMS, l'utilisation de la caméra, etc. C'est pourquoi, ces autorisations doivent être bien maitrisées afin de garantir un environnement sécuritaire d'exécution des applications.

Pour mieux les contrôler, nous avons proposé un contrôleur d'accès qui permet à l'utilisateur de consulter la liste des permissions des applications installées dans son smartphone Android. Ce contrôleur permet par la suite d'accorder ou de retirer l'accès à ces permissions. L'adoption de notre contrôleur permet à l'utilisateur de définir des politiques de contrôle qui s'exécutent automatiquement selon le contexte actuel comme le temps, la localisation, l'utilisation des ressources en même temps, la consommation de la batterie, la consommation du CPU, etc.

Afin que l'utilisateur soit capable de créer ses propres politiques de sécurité, nous avons défini un langage de spécification des politiques de sécurité permettant de contrôler l'accès, gérer les risques et d'exprimer des politiques pour les conflits de ressources. En se basant sur ce langage, nous avons développé notre contrôleur qui permet de définir des politiques contextuelles en spécifiant divers paramètres notamment l'heure, l'emplacement, les applications, l'état de l'appareil, etc. Les conditions à vérifier dans chaque politique seront basées sur ces paramètres définis afin d'appliquer le contrôle.

En effet, lors de la détection d'un changement du contexte actuel, le contrôleur examine la liste des politiques sauvegardés dans sa base de données à partir de notre algorithme de lecture des politiques. Le but de cette manipulation est d'identifier celles qui sont concernées par ce contexte et de s'assurer que les règles sont valides au contrôle souhaité. Dans le cas où la règle d'une politique est validée, le contrôleur détermine la liste des applications identifiées par leurs noms de package, la liste des politiques de sécurité concernées par application et la nouvelle décision à appliquer pour chaque permission.

La décision de l'exécution des méthodes qui demandent la permission de l'utilisateur est saisie dans le moniteur. Elle peut être modifiée plusieurs fois durant l'exécution de

l'application, exactement, au moment que l'utilisateur veut empêcher l'accès à l'API manuellement ou lorsqu'une politique de sécurité est vérifiée dans un contexte actuel.

Suivant un mécanisme existant dans Android permettant la communication des Intents entre des activités dans des différentes applications, nous envoyons des Intents personnalisés du contrôleur vers les applications concernées par ce contrôle. Avec un BroadcastReceiver paramétré implanté dans chacun de ces derniers, nous interceptons l'Intent envoyé qui contient les permissions à gérer et les décisions à appliquer pour mettre à jour la liste des décisions des politiques de sécurité. Dépendamment de la décision prise par le contrôleur, chaque application va connaitre l'action nécessaire (autoriser, interdire) pour chaque permission.

Afin de permettre cette communication et la collaboration entre les applications et le contrôleur, nous allons réécrire chaque application installée sur le Smartphone. Plus précisément, nous allons insérer un code qui va jouer le rôle d'un écouteur des intents du contrôleur et qui va mettre à jour les permissions selon leur contenu (décisions). Nous avons adopté une approche orientée aspects en utilisant le compilateur AspectJ (The AspectJ Team, 2003).

3.3 Langage de politiques de sécurité de contrôle d'accès APSL

Dans cette section, nous définissons un langage de spécification des politiques sur Android intitulé APSL (Android Policy Specification Langage) pour exprimer les politiques de sécurité pour les activités de contrôle d'accès et de gestion des risques. De plus, le langage permet d'exprimer des politiques de gestion des conflits de ressources. Le compilateur des applications Android ne supporte aucun langage de contrôle d'accès des politiques. Pour cela, nous avons créé notre langage de contrôle d'accès des politiques APSL basé sur Java. Le langage APSL peut être exporté en plusieurs formats comme XML, JSON etc.

Notre solution se concentre sur des politiques définies par l'utilisateur. Pour assurer leur application, nous avons créé ce langage pour développer un contrôleur centralisé qui permet de définir des politiques contextuelles tout en spécifiant différents paramètres tels que le temps, l'emplacement, les applications, l'état du périphérique, etc.

3.3.1 Les politiques de sécurité

Dans cette étude, nous avons opté pour une solution présentant à l'utilisateur une interface de contrôle qui lui permet d'écrire lui-même une politique de sécurité textuelle en se basant sur le langage APSL. Plus précisément, la politique présente une règle ou un ensemble de règles en se basant sur un ensemble des conditions. Du coup, nous permettons à l'utilisateur de mettre en place les scénarios de contrôle désirés.

Dans notre solution, le contrôleur doit être capable de transformer tous les scénarios dangereux qui menacent la vie privée de l'utilisateur à des contrôles de sécurité. Dans ce qui suit, nous présentons des exemples de scénarios dangereux.

- **Scénario 1** : Un utilisateur utilise un réseau wifi public dans un café et connecte son smartphone sur Internet. Il tente de consulter ses données dans son application bancaire. Étant donné que le réseau public n'est pas sécurisé et qu'il y a d'autres personnes qui utilisent le même réseau, nous avons alors un risque d'envoi des requêtes d'attaques et de vol des données privées (*voir* Figure 3.1).

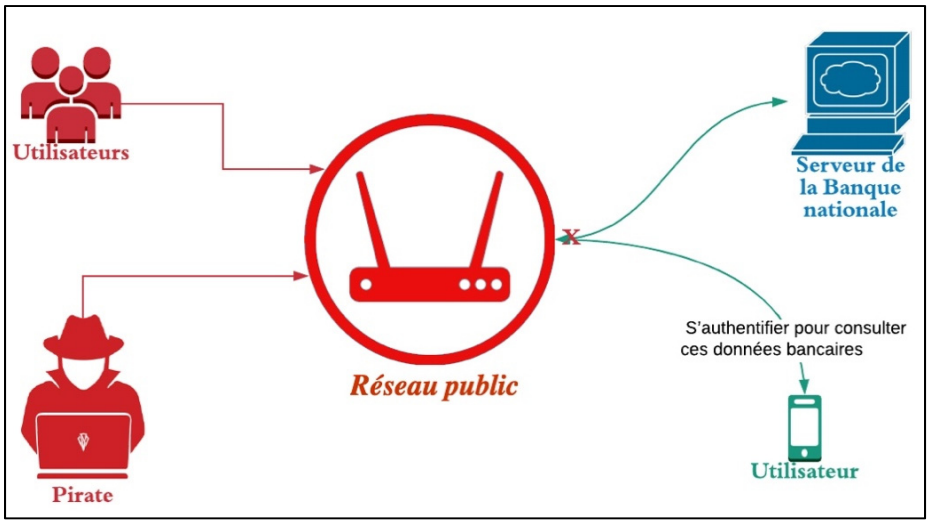


Figure 3.1 Consulter les données bancaires sur le réseau public d'un café

- **Scénario 2 :** Si l'utilisateur est dans une réunion de travail privé chaque lundi de 9h00 jusqu'à 10h00 par Skype alors, beaucoup d'autres applications sont capables d'enregistrer ses discussions et les partager en public (*voir* Figure 3.2).

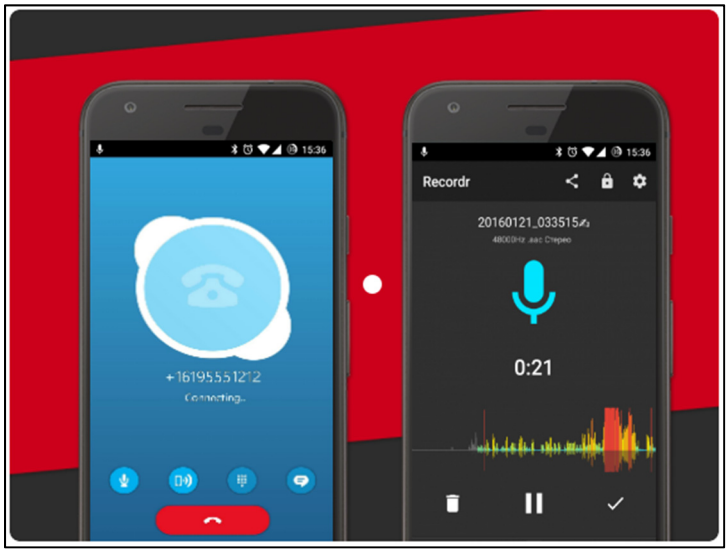


Figure 3.2 Utiliser une conférence Skype dans une réunion confidentielle.

- **Scénario 3** : Si l'utilisateur veut prendre une photo d'un chèque et l'envoyer sur son application bancaire alors qu'il existe d'autres applications qui prennent des captures d'écrans automatiques. Par conséquent, elles peuvent lui prendre une capture d'écran pour son login et son mot de en plus de son chèque et d'autres informations (*voir* Figure 3.3).

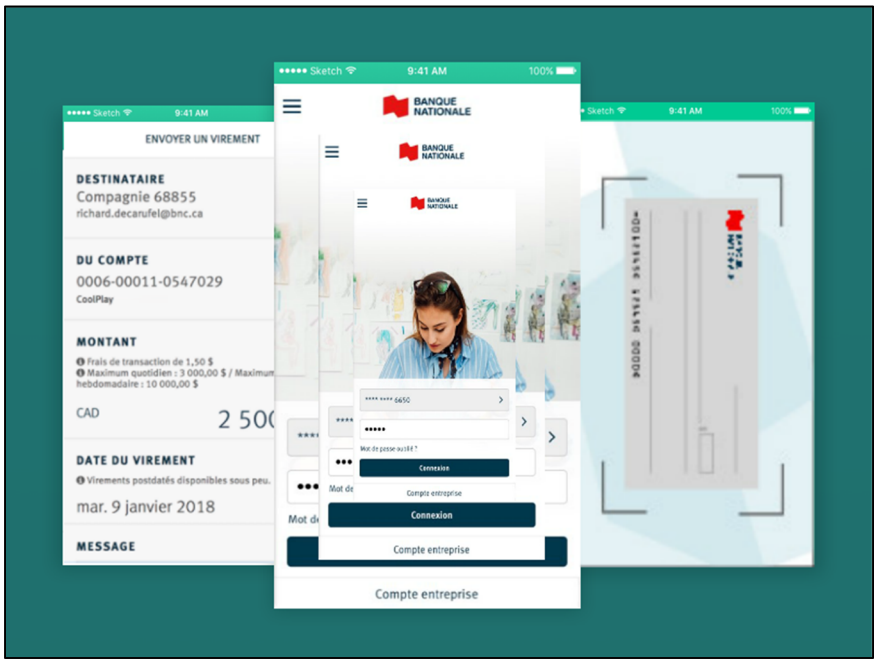


Figure 3.3. Utiliser les services d'une application bancaire

La majorité des applications Android qui demandent les informations personnelles et privées de l'utilisateur sont développées avec de grandes normes de sécurité comme le chiffrement des données, l'authentification en deux étapes, etc. Cependant, d'après les exemples des scénarios que nous avons cités, nous remarquons que le niveau de sécurité des applications est dépendant d'un contexte à l'autre comme l'emplacement, le temps, d'autres applications qui sont en cours d'exécution, des ressources utilisées, etc.

3.3.2 Les contextes

Les contextes jouent un grand facteur dans le changement de la sécurité des applications comme illustré par les scénarios de la section précédente. Pour cela, nous avons essayé d'identifier le maximum des contextes qui affectent les applications comme :

- **Localisation** : le changement de l'emplacement est parmi les contextes les plus importants qui affectent la sécurité d'une application. Par exemple, l'accès à des données privées dans un emplacement qui contient des caméras de surveillance etc.
- **Date** : pour une date spécifique, l'utilisation des ressources des applications peut être non sécurisé, donc l'utilisateur peut ou doit prendre en considération les dates pour contrôler ses applications.
- **Heure** : Le contexte heure est aussi important. Il permet à l'utilisateur de spécifier un intervalle de contrôle. Par exemple, s'abstenir d'utiliser les applications professionnelles hors les heures de travail ou encore utiliser des jeux durant la période de travail, etc.
- **Ressources utilisées** : lorsqu'une application A est en train d'utiliser des ressources en même temps qu'une application B, il peut y avoir un conflit (concurrence). L'accès à une donnée critique par une application peut imposer de nouvelles restrictions sur ses droits d'accès internet, d'envoi de message, etc. Plusieurs autres scénarios motivent le besoin d'adapter les droits d'accès des applications au contexte de leur utilisation des ressources.
- **Applications en cours d'exécution** : Il existe beaucoup d'applications qui envoient de fausses informations à savoir le changement des données GPS ou la réalisation des captures d'écrans automatiquement, etc. Ces dernières ont comme objectif de tromper d'autres applications (par exemple gagner des jeux, etc.), mais réellement elles peuvent être très dangereuses sur les données privées de l'utilisateur. Il est alors important d'adapter des décisions de sécurité à ces applications.
- **CPU** : Certaines applications prennent un grand temps d'exécution ce qui risque de monopoliser le processeur et d'affecter négativement le fonctionnement du Smartphone. Il est donc important de contrôler leur exécution selon le pourcentage de CPU disponible.

- **Batterie** : il y'a des applications qui influencent aussi sur la consommation d'énergie du smartphone. C'est pourquoi, il est important de contrôler leur exécution en se basant sur le niveau de la batterie.

Beaucoup d'autres contextes existent, ils dépendent des ressources du smartphone ainsi que des attentes de l'utilisateur.

3.3.3 Les conditions

Au cours de cette étude, c'est l'utilisateur lui-même qui doit déterminer les conditions. Ces dernières permettent essentiellement la spécification de politiques de sécurité dépendantes du contexte. C'est pour cela que nos conditions dévoient vérifier l'état courant du contexte.

Des opérateurs logiques ont été utilisés pour que nous puissions créer nos conditions (*Voir* Tableau 3.1).

Tableau 3.1 Opérateurs logiques pour définir les conditions

Operateurs logiques	Signification
<	Inférieur
>	Supérieur
<=	Inférieur ou égale
>=	Supérieur ou égale
!=	Diffèrent
&&	ET
||	OU

Les exemples de contextes que nous avons déjà listés dans la section précédente sont des échantillons que nous avons utilisés pour implémenter notre solution et générer les politiques de sécurité. Pour chacun de ces contextes, les exemples suivants montrent la façon de spécifier des conditions appropriées.

- **Localisation** : Cette condition permet à l'utilisateur d'empêcher des applications de s'exécuter ou de retirer l'accès à des méthodes APIs, dans une localisation précise donnée par le GPS. La condition va être présentée sous le format :
  « Localisation_Actuelle == Localisation_définie_par_lutilisateur ».

- **Date :** Cette condition permet à l'utilisateur d'empêcher des applications de s'exécuter ou de retirer l'accès à des méthodes APIs pendant un intervalle de dates donné. La présentation de cette condition va être sous le format :
  « si Date_Actuelle <= Date_debut_définie_par_lutilisateur &&
  Date_Actuelle >= Date_fin_définie_par_lutilisateur ».
- **Intervalle de temps** : Cette condition permet à l'utilisateur d'empêcher des applications de s'exécuter ou de retirer l'accès à des méthodes APIs pendant un intervalle de temps donné. La présentation de cette condition est sous le format :
  « si Temps_Actuel <= Temps_debut_défini_par_lutilisateur &&
  Temps_Actuel >= Temps_fin_défini_par_lutilisateur ».
- **Ressources utilisées** : Elle permet à l'utilisateur de définir une condition qui porte sur l'utilisation d'un ensemble de ressources ou l'exécution d'un ensemble de méthodes API. Cette condition peut prendre la forme suivante : « INTERNET && READ_EXTERNAL_STORAGE && SET_WALLPAPER qui sont utilisés dans APP1 ».
- **Applications en cours d'exécution** : Elle accorde à l'utilisateur de définir un ensemble d'applications qui ne doivent pas être exécutés en même temps que d'autres applications. La présentation de cette condition peut prendre la forme suivante :
  « APP1 && APP2 && APP3 »
- **CPU** : Permet à l'utilisateur de gérer l'exécution des applications ou des fonctions APIs en considérant le pourcentage de la consommation du processeur. La condition va être présentée sous le format :
  « APP1_Pourcentage_CPU == Pourcentage_définis_par_lutilisateur »
- **Batterie** : Permet à l'utilisateur de gérer l'exécution des applications ou des fonctions APIs en considérant le pourcentage de la consommation de la batterie. La condition va être présentée sous le format :
  « Pourcentage_Battery >= Pourcentage_définis_par_lutilisateur »

3.3.4 Conception du langage des politiques de sécurité APSL

Pour développer le langage APSL qui permet à l'utilisateur de définir des politiques contextuelles et les transformer en des contrôles sécuritaires, nous devons se baser sur une conception souple qui varie avec la complexité des politiques, rationnelle, capable d'exécuter toutes les conditions et qui permet d'ajouter facilement de nouveaux contextes.

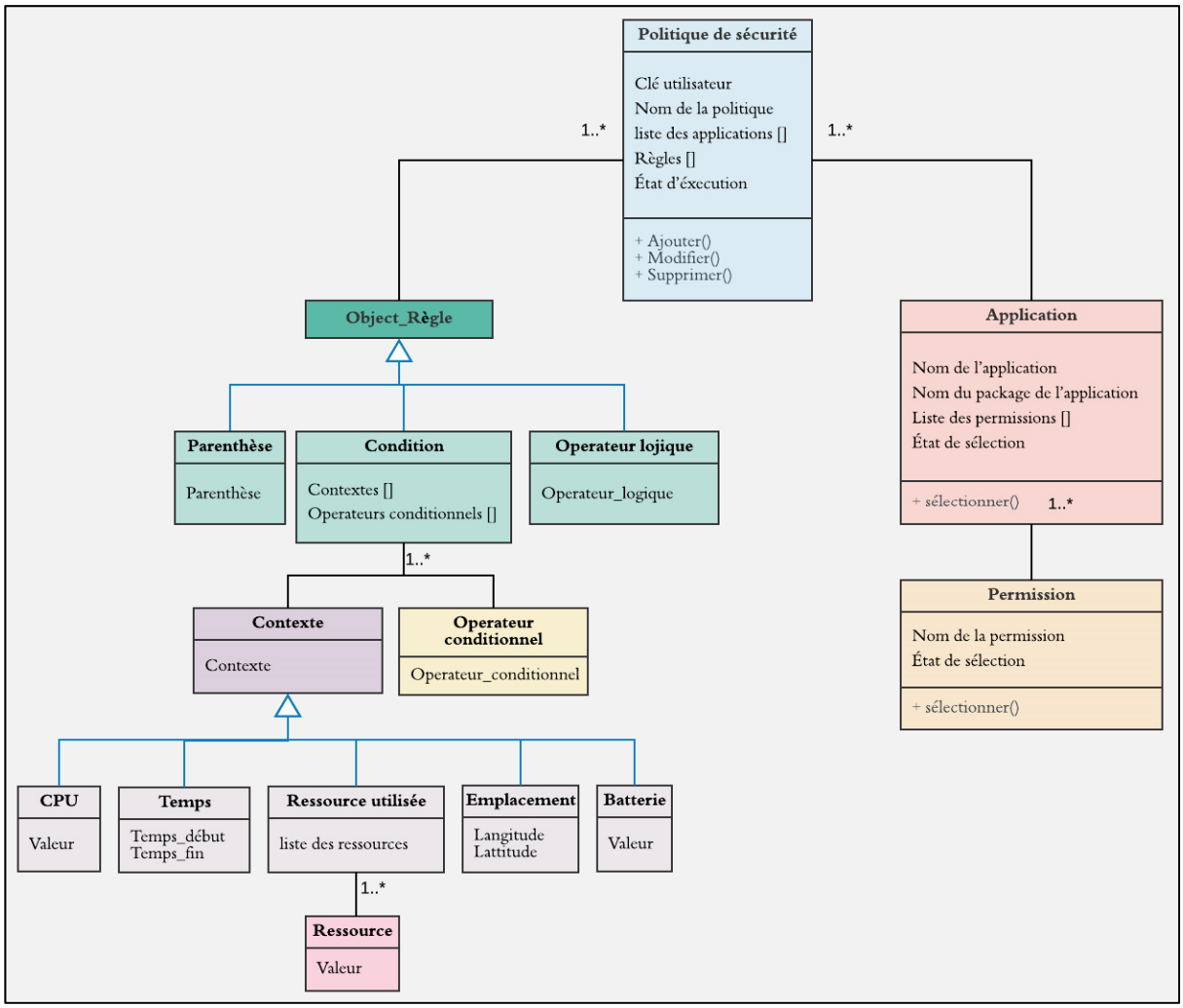


Figure 3.4 Diagramme de classe de la solution APSL

Le langage APSL (Figure 3.4) se base principalement sur la définition d'une politique qui se compose par une clé utilisateur, un nom de politique, un état d'exécution de la politique, une

règle du contrôle et une liste des applications à contrôler. Chaque application est caractérisée par un nom, le nom du package, l'état de sélection et la liste des permissions. Chaque permission est composée d'un nom et d'un état de sélection. Chaque application est sélectionnée, c'est-à-dire qu'elle contient des permissions à contrôler sinon nous somme dans l'obligation de ré-autoriser les permissions dans les applications non sélectionnées. Une règle de sécurité est composée d'une liste des objets de type Object_règle qui peuvent être Parenthèses, Conditions, Operateurs logiques, Operateurs conditionnels, CPU, Temps, Ressource utilisée, ou encore Emplacement ou Batterie. Cette architecture souple permet à l'utilisateur d'ajouter facilement de nouveaux contextes et de définir n'importe quelle règle, quelle que soit sa complexité.

3.3.5 Détermination du point de décision de politique avec APSL

Le présent langage APSL permet à l'utilisateur de définir des politiques de sécurité pour contrôler ses applications. Chaque politique définie par l'utilisateur reste en cours d'exécution jusqu'à ce que le contrôleur déclenche un changement de contexte, c'est-à-dire lorsque le contexte actuel change, le contrôleur doit déterminer la décision du contrôle de la politique. Par exemple : si l'utilisateur a changé son emplacement ou le temps, le contrôleur doit parcourir la liste des politiques pour vérifier si le contexte changé est valide avec les conditions des règles de sécurité ou non. D'une autre façon, lorsque le contexte change, le contrôleur détermine le point de décision de chaque politique et dans le cas où le point de décision est valide le contrôleur applique le contrôle sur l'application concernée, comme indiqué dans la Figure 3.5.

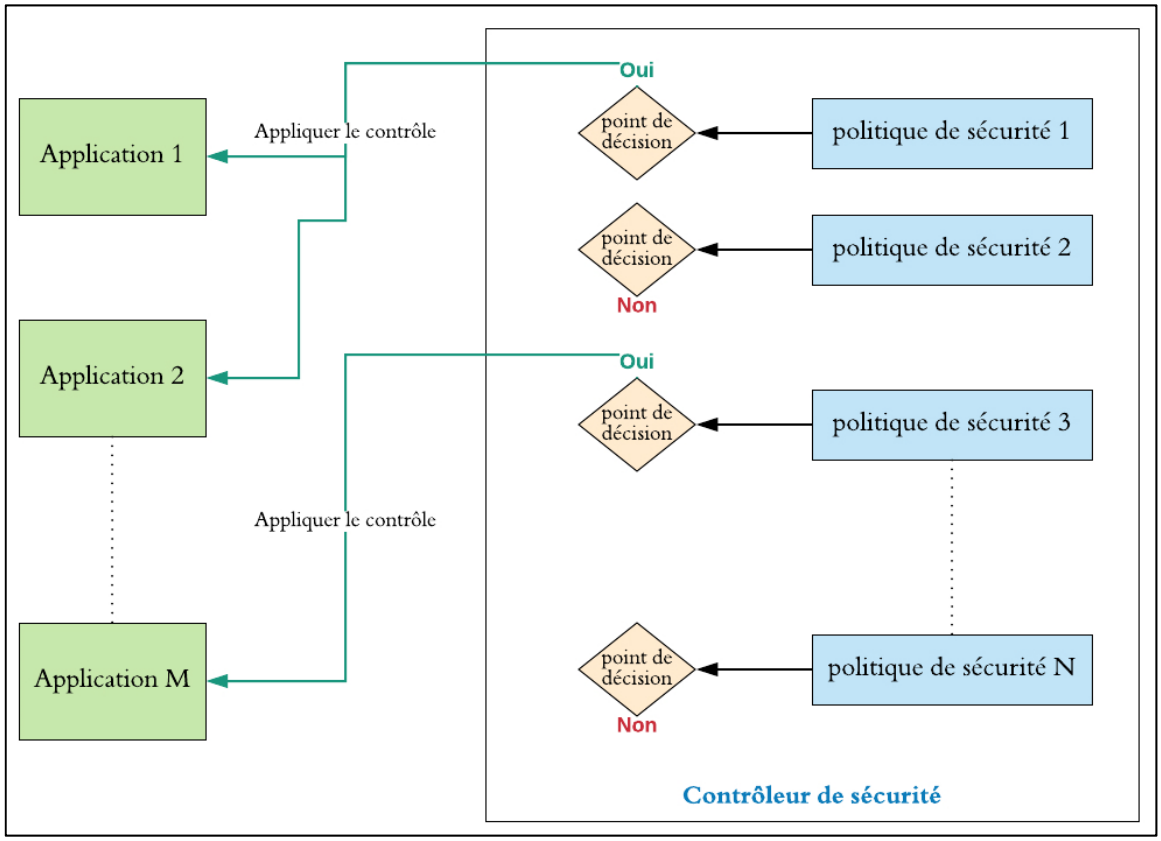


Figure 3.5 Mécanisme de détermination des Points de décision avec APSL

Les règles des politiques de sécurité définies par l'utilisateur peuvent contenir plusieurs conditions, différents contextes, plusieurs opérateurs logiques et parenthèses pour spécifier la priorité entre les conditions. Ces politiques peuvent être exécutées simultanément même dans des contextes différents et complexes. Dans ce cas, la détermination du point de décision de la politique devient plus complexe. Pour faciliter et accélérer l'exécution de la politique, nous avons créé un algorithme qui prend les contextes actuels et la règle de contrôle en paramètres et qui retourne par la suite le point de décision de contrôle. Les arbres sont des structures utilisées pour le classement et l'accès rapide aux données qui sont utilisées dans notre algorithme. Un arbre est une structure constituée de nœuds, qui peuvent avoir des sous-nœuds (qui sont d'autres nœuds). Sur l'exemple, la règle de sécurité a pour sous-nœuds d'autres règles, et sont eux même des sous-règles de la règle principale.

L'élément à gauche de la règle racine présente la sous-règle gauche, l'élément à droite de la règle racine présente la sous-règle droite.

Cette propriété doit être vérifiée récursivement à tous les niveaux jusqu'à obtention des feuilles au dernier niveau qui présentent les conditions. La Figure 3.6 présente un exemple d'une règle selon la structure des arbres adoptée.

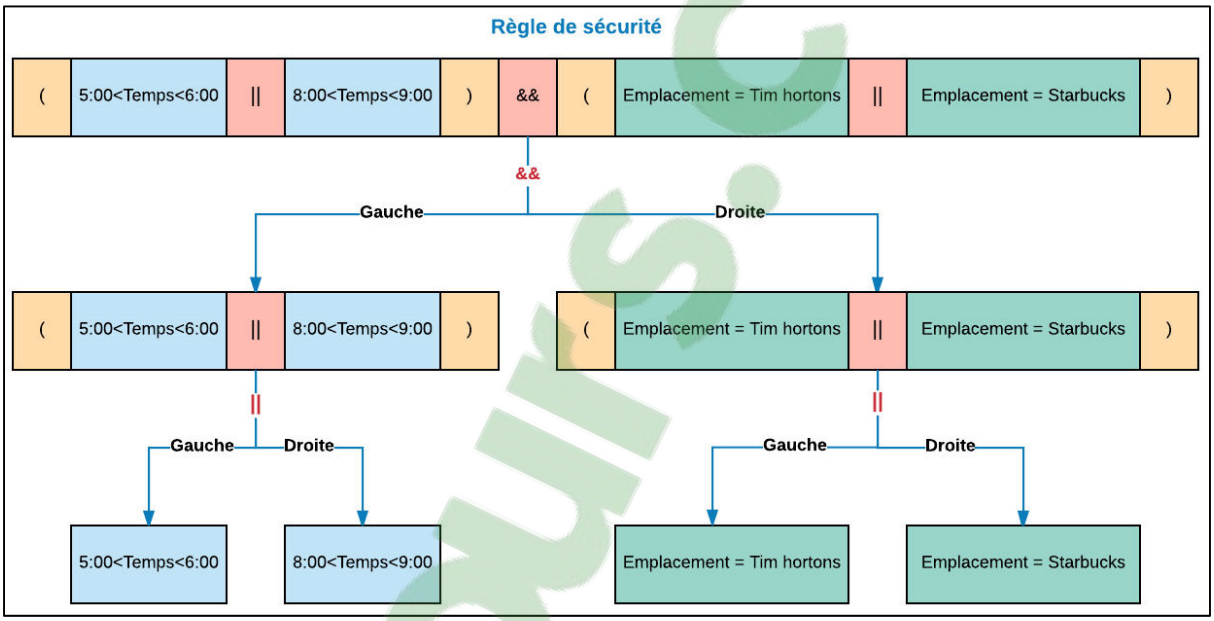


Figure 3.6 La structure d'exécution des politiques avec APSL

Nous avons choisi que notre algorithme soit récursif pour réduire la complexité des règles de sécurité tant que chaque règle est composée de sous-règles. Puisque notre algorithme s'appelle lui-même, il est impératif que nous prévoyions la vérification des conditions d'arrêt de la récursion, sinon le programme ne s'arrête jamais.

3.4 Architecture de la solution proposée

3.4.1 Présentation générale de l’architecture

Notre travail se base principalement sur notre langage de spécification des politiques de sécurité développé afin de satisfaire deux domaines. Le premier étant un axe pour réécrire l'application par l’injection des contrôles nécessaires des APIs, et le deuxième cadre impliquant le développement d'un contrôleur d'applications qui se base sur un langage de politique pour gérer le fonctionnement de toutes les applications sur le smartphone Android. Par conséquent, l'utilisateur crée des règles et des conditions sous la forme de politiques à travers son monitor. Ce dernier communique avec les applications et applique automatiquement le contrôle demandé (*Voir* Figure 3.7).

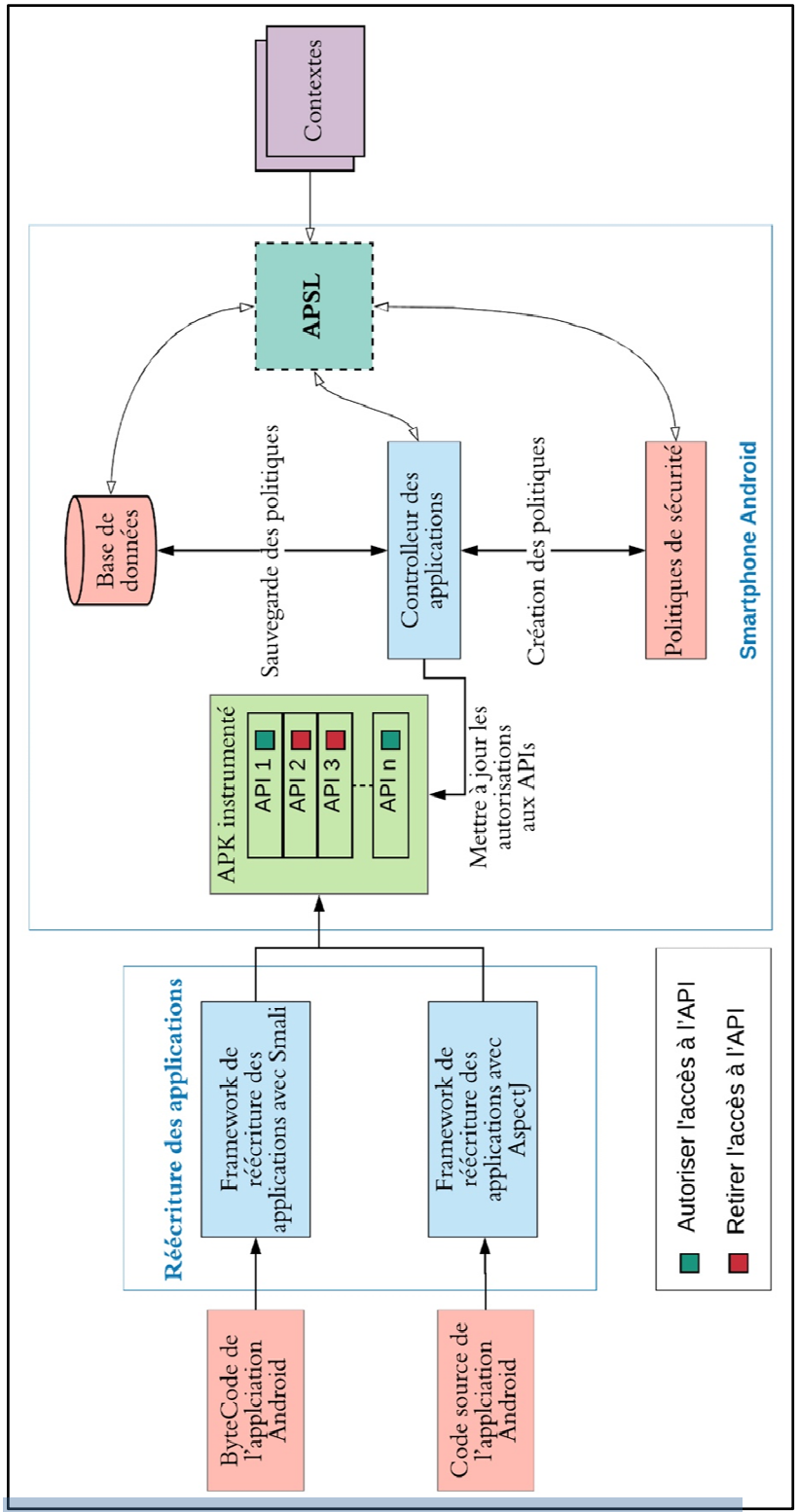


Figure 3.7 Architecture générale de la solution proposée

Cette première partie de la recherche se concentre sur la réécriture des applications Android. Le but est d'ajouter un écouteur qui applique la décision envoyée par le contrôleur afin de bloquer ou libérer les fonctions qui demandent l'accès aux APIs.

La deuxième partie, qui est la plus importante, est le contrôleur. Le contrôleur permet aux utilisateurs d'écrire des politiques de contrôle contextuelles à travers le langage que nous avons développé. Il les exécute en temps réel selon un contexte actuel puis il communique la décision aux applications concernées par ce contrôle.

Dans ce cadre, l'idée principale est d'ajouter des instructions de communication entre le contrôleur et les applications afin de faire respecter certaines politiques de sécurité contextuelles.

À partir d'Android Marshmallow 6.0 (niveau d'API 23), un nouveau système d'autorisation pour les applications est implémenté pour permettre aux utilisateurs d'accorder des autorisations aux applications pendant leur exécution, et non lors de leur installation. Cette approche rationalise le processus d'installation d'applications, car l'utilisateur n'a pas besoin d'accorder des autorisations lorsqu'il installe ou met à jour l'application.

Ce nouveau modèle donne également à l'utilisateur plus de contrôle sur les fonctionnalités de l'application. Par exemple, un utilisateur peut choisir d'autoriser une application de caméra à accéder à la caméra, mais pas à l'emplacement de l'appareil. L'utilisateur peut révoquer les autorisations à tout moment en accédant à l'écran paramètres de l'application.

Les autorisations sont divisées en deux catégories, normales et dangereuses. Les autorisations normales n'engendrent pas un risque direct pour la vie privée de l'utilisateur. Par contre, les autorisations dangereuses peuvent permettre à l'application d'accéder aux données confidentielles de l'utilisateur.

Le système d'autorisation des applications existant dans Android permet de contrôler seulement les autorisations classées comme dangereuses alors que selon notre approche des politiques, nous considérons que, quel que soit le type de permission, il peut causer un risque ou un conflit avec un contexte spécifique. Ceci est un acte non sécuritaire pour l'utilisateur et pour ses données qu'il partage alors il faut que nous contrôlons toutes les autorisations.

Pour que notre contrôleur soit capable d'empêcher les applications Android en cours d'exécution d'accéder à certaines APIs sans avoir l'autorisation, il est nécessaire de modifier les applications originales par l'ajout d'une nouvelle partie de code qui permet de donner et retirer les permissions d'une façon flexible.

La Figure 3.8 montre la nouvelle architecture de l'application à contrôler après la réécriture sur deux versions. La première est dédiée aux systèmes des Android supérieurs à 6.0 qui contiennent le système de permissions granulaires et la deuxième est dédiée aux systèmes Android inférieurs à 6.0.

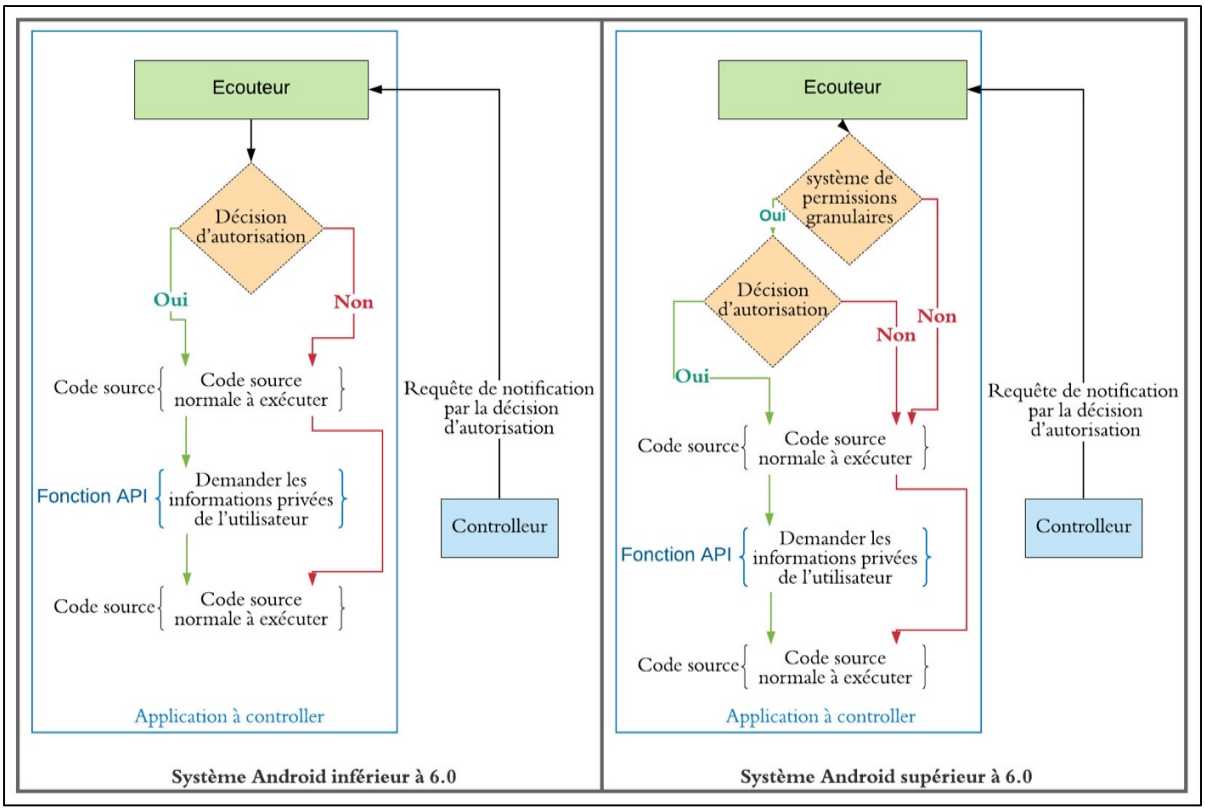


Figure 3.8 Architecture de la solution du contrôle des applications après la réécriture

Comme c'est affiché dans cette figure, le contrôleur va notifier l'application chaque fois qu'il y'a un changement d'accès d'autorisation. Donc, nous ajoutons le BroadcastReceiver qui va rester en écoute et qui va jouer le rôle d'un récepteur des décisions. Selon la décision de l'autorisation reçue par le récepteur et selon d'état de l'autorisation de la permission de l'API dans le système de permissions granulaires donnée pas l'utilisateur, ce BroadcastReceiver va bloquer la fonction qui accède à l'API ou la libère.

Sachant que cette figure représente un contrôle d'accès pour un seul API mais notre récepteur est capable de traiter les autorisations pour plusieurs APIs.

Il est noté que nous avons appliqué notre approche de réécriture sur des applications open source pour que nous puissions modifier leur code source.

Notre approche de réécriture consiste à ajouter un code sans modifier le code existant de l'application ce qui nous rend capables de contrôler toutes les applications.

3.4.2 Demande des autorisations en cours d'exécution

Pour accéder aux données privées de l'utilisateur comme la localisation, les photos, les contacts, etc. ou bien des informations du Smartphone tel que capteurs, wifi, internet, etc., Android met aux développeurs des fonctions qui sont déjà prédéfinies intitulées APIs. Ces APIs peuvent être utilisées facilement pour récupérer les données dont nous avons besoin, mais Android contrôle l'accès à ces APIs et exige la demande d'autorisation de la part de l'utilisateur de l'application. Si une application doit accéder à une fonctionnalité protégée par une autorisation, elle doit déclarer qu'elle demande l'autorisation avec un élément <uses-permission> dans le fichier Manifest de chaque application.

En cours d'exécution de l'application, si l'autorisation est accordée, l'application peut utiliser les fonctionnalités protégées. Sinon, ses tentatives d'accès à ces fonctionnalités échouent. Jusqu'à l'API 23, 58 permissions ont été définies par Android voici quelques exemples :

- android.permission.CALL_EMERGENCY_NUMBERS
- android.permission.READ_OWNER_DATA
- android.permission.READ_CALENDAR

Chaque API peut demander plus qu'une permission, alors Android crée un groupe de permissions. Par exemple dans le groupe de permission SMS nous trouvons cinq permissions de fonctionnalités différentes :

- android.permission.SEND_SMS
- android.permission.RECEIVE_SMS
- android.permission.READ_SMS
- android.permission.RECEIVE_WAP_PUSH
- android.permission.RECEIVE_MMS

Pour chaque autorisation déclarée dans le fichier Manifest, nous pouvons trouver plusieurs attributs, fonctions et classes prédéfinis par le système d'Android qui sont accordés. Notre principe de contrôle consiste à protéger ces attributs, fonctions et classes par la création d'un interpréteur avant chacun d'eux. Du coup, lorsque l'application reçoit la décision de la part du contrôleur, l'intercepteur met à jour le point de décision d'autorisation de l'interrupteur. Ce dernier va autoriser ou refuser l'accès à l'API.

3.4.3 L'interception des décisions de contrôle

Pour pouvoir recevoir des données entre différentes applications, Android permet aux développeurs de créer une classe qui implémente le BroadcastReceiver (récepteur de diffusion). Ces classes sont conçues pour recevoir des Intents (Intentions) et appliquer des comportements spécifiques à leur code.

Les BroadcastReceiver répondent simplement aux messages de diffusion provenant d'autres applications ou du système lui-même. Ces messages sont parfois appelés événements ou Intentions. Par exemple, les applications peuvent également initier des diffusions pour permettre à d'autres applications de savoir que certaines données ont été téléchargées sur l'appareil et qu'elles peuvent être utilisées. C'est donc le récepteur de diffusion qui interceptera cette communication et lancera l'action appropriée.

Lors de l'envoi d'une diffusion, le système achemine automatiquement les diffusions vers les applications qui se sont abonnées à ce type de diffusion.

D'une manière générale, les diffusions peuvent être utilisées comme un système de messagerie entre applications et en dehors du flux utilisateur normal. L'interface BroadcastReceiver possède une seule méthode onReceive() que la classe java doit implémenter, et qu'après exécution va être directement supprimée par le garbage collector . Malgré que la diffusion avec BroadcastReceiver soit la seule manière proposée par le système Android qui permet la communication directe avec les applications, cette

diffusion est aussi optimisée en temps d'exécution permettant d'envoyer des Intentions customisées. D'où, elle est la meilleure technique de communication à appliquer dans notre solution.

Selon notre cas, le monitor doit toujours envoyer une Intent qui contient la décision d'autorisation à l'application concernée par le contrôle. Le BroadcastReceiver qui reçoit cette Intent va appliquer les mises à jour sur les permissions aux APIs dans l'application. Comme nous avons déjà expliqué dans la section précédente, le contrôleur peut contrôler plusieurs applications et il peut retirer plusieurs permissions à la fois dans la même application. Pour cela, nous devons spécifier dans chaque Intent envoyée par le contrôleur, l'application destinatrice et la permission à contrôler.

Par la suite, nous aurons recours à utiliser des Intents paramétrées, le contrôleur va donc envoyer un tableau de données qui contient le nom de package de l'application à sécuriser, la permission à contrôler et le statut de l'autorisation.

La Figure 3.9 montre le rôle principal du BroadcastReceiver dans l'application à contrôler.

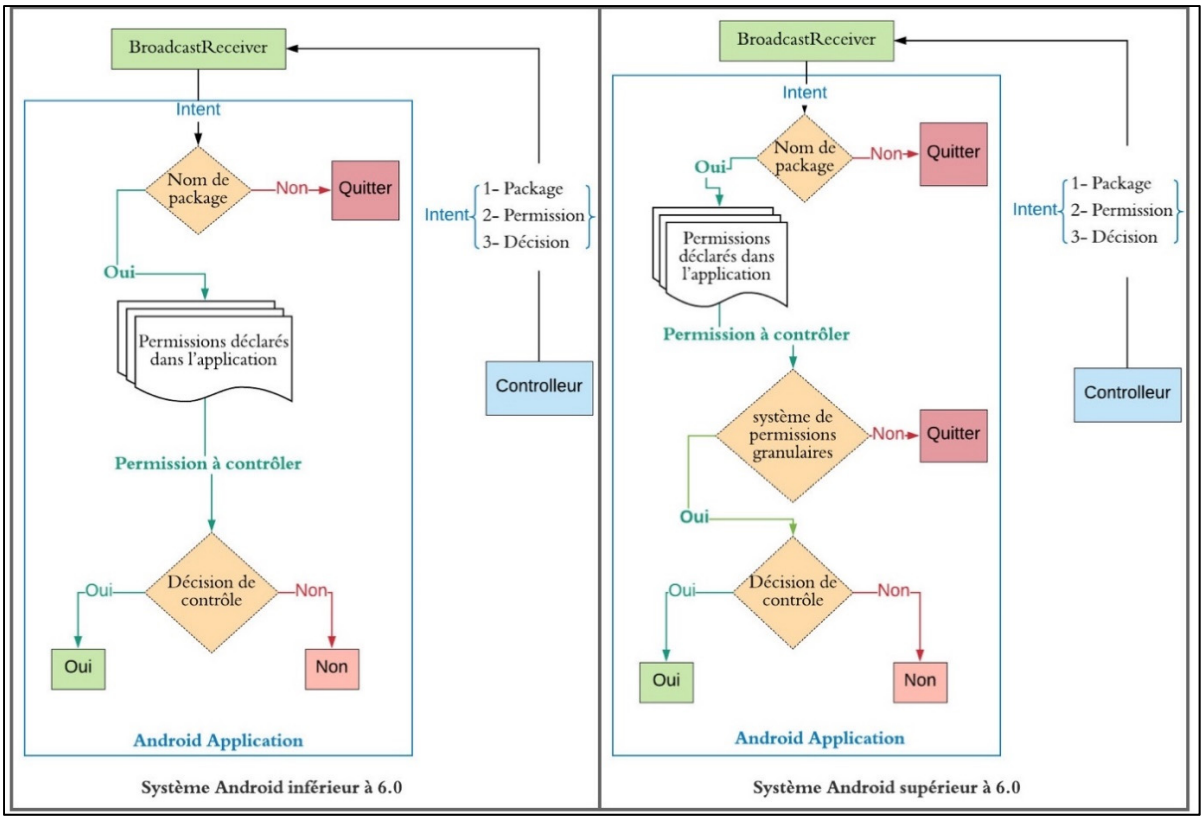


Figure 3.9 Architecture du rôle principal du BroadcastReciever
dans le contrôle des applications

Cette figure explique l'envoi d'une Intent paramétrée de la part du contrôleur vers l'application. Cette dernière reçoit l'Intent et analyse ses paramètres selon 5 étapes :

1. Si le nom de package envoyé par l'Intent correspond au nom de package de l'application, nous passons vers la deuxième étape, sinon, nous quittons, car l'application n'est pas concernée par ce contrôle.
2. Chercher la permission à contrôler dans la liste des permissions déclarées dans l'application.
3. Dans le cas où le système d'Android est supérieur à la version d'Android 6.0, nous passons à la quatrième étape sinon nous passons directement à la dernière étape.

4. Vérifier si l'utilisateur autorise l'accès à la permission concerné par le contrôle dans le système de permission granulaire, si oui nous passons à la dernière étape sinon, nous quittons l'application, car l'utilisateur n'autorise pas l'accès à cet API.
5. Après avoir identifié la permission à contrôler dans l'application, nous affectons la décision de contrôle à l'interrupteur de l'exécution de la fonction d'API.

Comme nous l'avons déjà cité dans la section précédente, le contrôleur peut contrôler plusieurs APIs dans la même application et que chaque API peut contenir plusieurs permissions.

Dans cet exemple, le contrôle est appliqué sur deux permissions qui sont la « CAMERA » et « ACCESS_FINE_LOCATION ».

Pour conclure, dans chaque application à contrôler, nous devons ajouter un BroadcastReceiver qui reçoit des Intents contenant le nom de package de l'application concernée par le contrôle, les permissions à contrôler et la décision du contrôle.

Le BroadcastReceiver est aussi responsable de vérifier si la présente application est concernée par le contrôle, si oui, il va effectuer la nouvelle valeur de décision de contrôle reliée à la permission.

Pour empêcher les fonctions d'APIs de s'exécuter, nous avons commencé par lire le code source et ajouter les contrôles manuellement avant que nous procédons à un enchainement automatique pour qu'il soit applicable sur toutes les applications.

3.5 Réécriture des applications

Pour connaitre quelles sont les fonctions qui permettent d'accéder à un API, il est nécessaire de lire le code source de chaque application pour comprendre la structure utilisée par le

développeur, le rôle de l'application, et les classes qui utilisent des fonctions qui demandent les APIs.
D'une autre façon, le développeur est le seul qui peut choisir où il veut mettre les appels aux APIs, par exemple, ça peut être dans la fonction main OnCreate(), OnResume() ou dans des actions sur des boutons par exemple. Donc, à chaque fois, nous sommes obligés de parcourir le code source et lire toutes les classes pour comprendre quelles sont les fonctions à contrôler.

Souvent, les développeurs ont recours à créer des fonctions et des variables avec des noms bien spécifiques qui permettent de comprendre les rôles des fonctions et nous pouvons facilement choisir les fonctions qui demandent des permissions. Cependant, si le développeur donne des noms non significatifs, il va compliquer ce processus et nous serons obligé de lire le code ligne par ligne et de tester les résultats des fonctions en utilisant des logs pour identifier avec précision les fonctions à contrôler.

Notre solution actuelle est une approche que les développeurs peuvent suivre afin d'insérer les contrôles aux APIs lors du développement de leurs applications et les contrôler à partir de notre contrôleur. Cependant, elle doit être aussi applicable sur des applications déjà existantes sur le marché. Donc pour que notre solution devienne un Framework souple à utiliser sur des applications existantes, on doit automatiser la réécriture des applications sans être dans le besoin de lire et de chercher dans le code source.

Une solution qui analyse le code source de l'application et injecte automatiquement les contrôles nécessaires sur les fonctions des API à contrôler est notre objective. L'AspectJ est une extension compatible avec notre Framework de réécriture (The AspectJ Team, 2003). Le tissage des aspects est réalisé sur les classes compilées.

Au moment de la compilation du code source, au moment de l'exécution, ou bien au moment du chargement des classes par la machine virtuelle, nous pouvons établir ce tissage. D'où, avec cette technique, nous serons capables de créer une seule classe en AspectJ qui contient

les classes, les fonctions et les variables dédiées aux APIs comme des Joint-points et que nous spécifions le contrôle à ajouter pour chacun des joint-points trouvés dans le code source de l'application à contrôler.

Pour préparer notre fichier d'AspectJ, nous avons commencé par grouper toutes les classes, les fonctions et les variables prédéfinies par le système d'Android pour chaque API. Le Tableau 3.2 contient des exemples d'APIs ainsi que leurs fonctions prédéfinies.

Tableau 3.2 Classes, fonctions et variables prédéfinies par Android pour différentes APIs

Nom d'API	Classes, fonctions et variables prédéfinies par Android
CALENDAR	o CalendarContract.Calendars o CalendarContract.Events o CalendarContract.Instances o CalendarContract.Attendees o CalendarContract.Reminders
CAMERA	o Camera o SurfaceView o MediaRecorder o MediaStore.ACTION_IMAGE_CAPTURE o MediaStore.ACTION_VIDEO_CAPTURE
CONTACTS	o CursorLoader
LOCATION	o LocationManager o requestLocationUpdates() o LocationListener
MICROPHONE	o MediaRecorder o MediaMuxer
SENSORS	o Sensor o SensorEvent o SensorManager o SensorEventListener
SMS	o Telephony.Sms.Conversations o Telephony.Sms.Draft o Telephony.Sms.Inbox o Telephony.Sms.Intents o Telephony.Sms.Outbox o Telephony.Sms.Sent
STORAGE	o getExternalFilesDir()

Puisque nous étions capables de regrouper toutes les classes, les fonctions et les variables des APIs à contrôlées, nous pouvons maintenant les pointées comme des joint-points dans notre fichier AspectJ et définir les fonctions de contrôle associées à chacun d'eux.

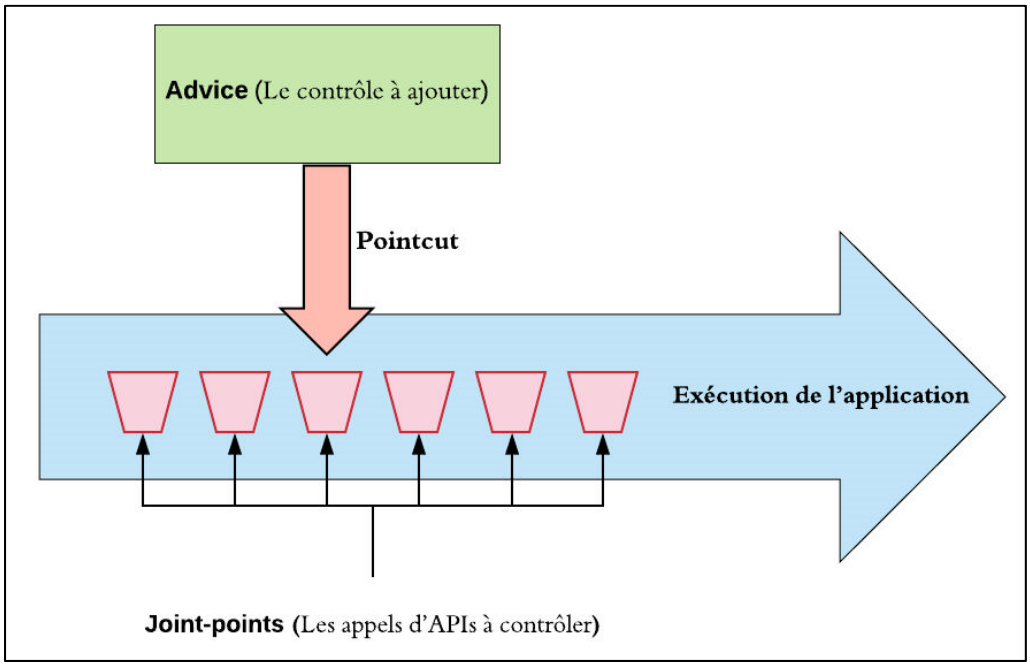


Figure 3.10 L'architecture d'injection des contrôles dans l'application avec AspectJ inspiré de (Gupta, 2015)

D'après cette Figure 3.10, nous pouvons voir que nos joint-points présentent les fonctions d'APIs à contrôler. Lors de la compilation de l'application, AspectJ cherche les joint-points et lorsqu'ils les trouvent, il fait un Pointcut et il ajoute l'Advice qui présente le code de contrôle à ajouter.

Avec cette technique, nous étions capables de contrôler toutes les API dans le même fichier AspectJ.

De cette façon, nous aurons juste besoin d'ajouter seulement le fichier AspectJ dans n'importe quelle application. En effet, lors de la compilation de l'application, le compilateur va chercher les joint-points dans le code source et injecter automatiquement les contrôles des APIs.

L'AspectJ, avec l'ajout des contrôles aux fonctions d'APIs nécessaires, procède à l'injection du code lors de la compilation de l'application, ce qui est équivalent à ce que nous faisons au début manuellement.
En conclusion, nous avons réalisé un Framework de réécriture de l'application automatique qui consiste à ajouter une seule classe dans l'application qui s'exécute automatiquement lors da compilation sans avoir besoin de modifier le code source originale.

3.6 Contrôleur centralisé des applications

Cette première partie consiste de réécrire les applications Android dans le but que ces dernières soient dans la mesure de communiquer avec le contrôleur d'application et recevoir les requêtes de contrôles. Pour cela, un Framework de réécriture a été développé consiste à ajouter une seule classe AspectJ qui injecte automatiquement des contrôles pour toutes les APIs.

La deuxième partie porte sur le contrôle des applications. Plus précisément, l'implémentation d'une interface utilisateur qui offre au contrôleur, d'un côté, une prise des décisions adéquates pour autoriser ou bloquer des applications à utiliser certaines permissions et d'un autre côté, une écriture des politiques textuelles offrant le contrôle centralisé des applications installées et capables de saisir des décisions obligatoires automatiquement dépendant du contexte actuel.

Pour cela, un langage de politique sera nécessaire pour comprendre les politiques textuelles données par l'utilisateur et les transformations en des contrôles automatiques. La Figure 3.11 montre en détail l'architecture de notre contrôleur.

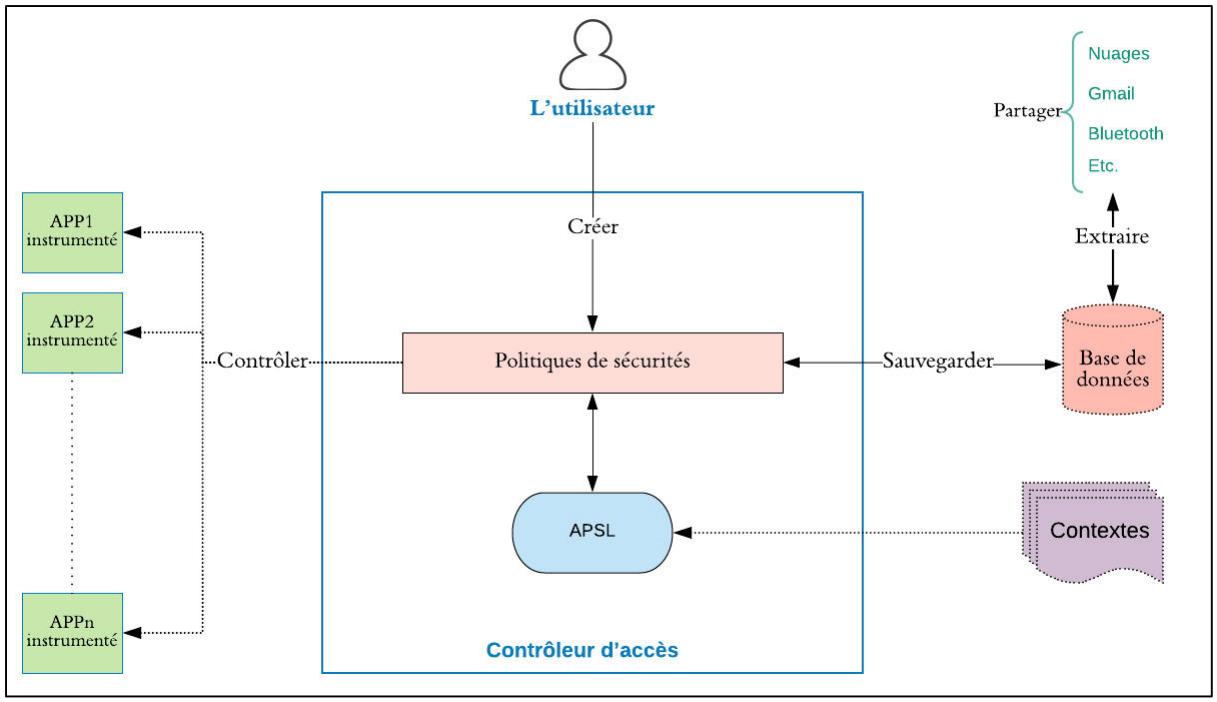


Figure 3.11 L'architecture de contrôleur d'accès

Le contrôleur offre à l'utilisateur une trousse d'outils pour qu'il soit capable de traduire ces scénarios dangereux à des politiques de sécurité. Du coup, un langage de contrôle d'accès des politiques doit être présent pour permettre à l'utilisateur de définir des règles de sécurité.

3.6.1 La création textuelle des politiques de sécurités

Notre contrôleur permet à l'utilisateur d'introduire les conditions qui contrôlent les scénarios dangereux. (Figure 3.12). De ce fait, il est capable de définir des politiques dépendantes de différents types de conditions. En outre, chacune des conditions créées peut avoir une combinaison avec d'autres conditions

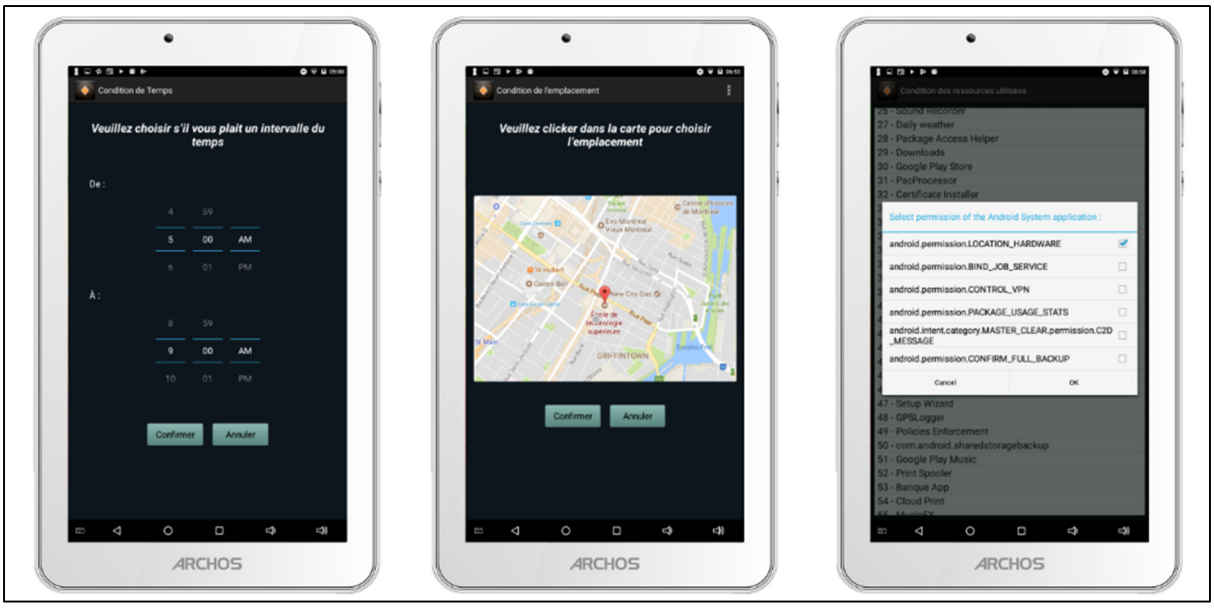


Figure 3.12 Exemples d'interfaces de création des conditions

Du coup, les politiques de contrôles d'accès se basent sur la combinaison des conditions en utilisant les deux opérateurs logiques "ET" et "OU" ainsi que la capabilité de spécifier la priorité d'exécution des conditions par l'ajout des parenthèses " (’’ et ")’’.

Chaque politique est caractérisée par un Nom qui donne une signification à son rôle, la liste des applications à contrôler, la règle de la sécurité à appliquer et son statut d'exécution. Ce dernier présente l'état de la politique s'il est en cours d'exécution ou en pause selon le choix de l'utilisateur. La règle contient le scénario de contrôle sous la forme de différentes conditions séparées par des parenthèses pour montrer la priorité entre eux ainsi que des opérateurs logiques. Le format d'une politique est bien expliqué dans la Figure 3.13.

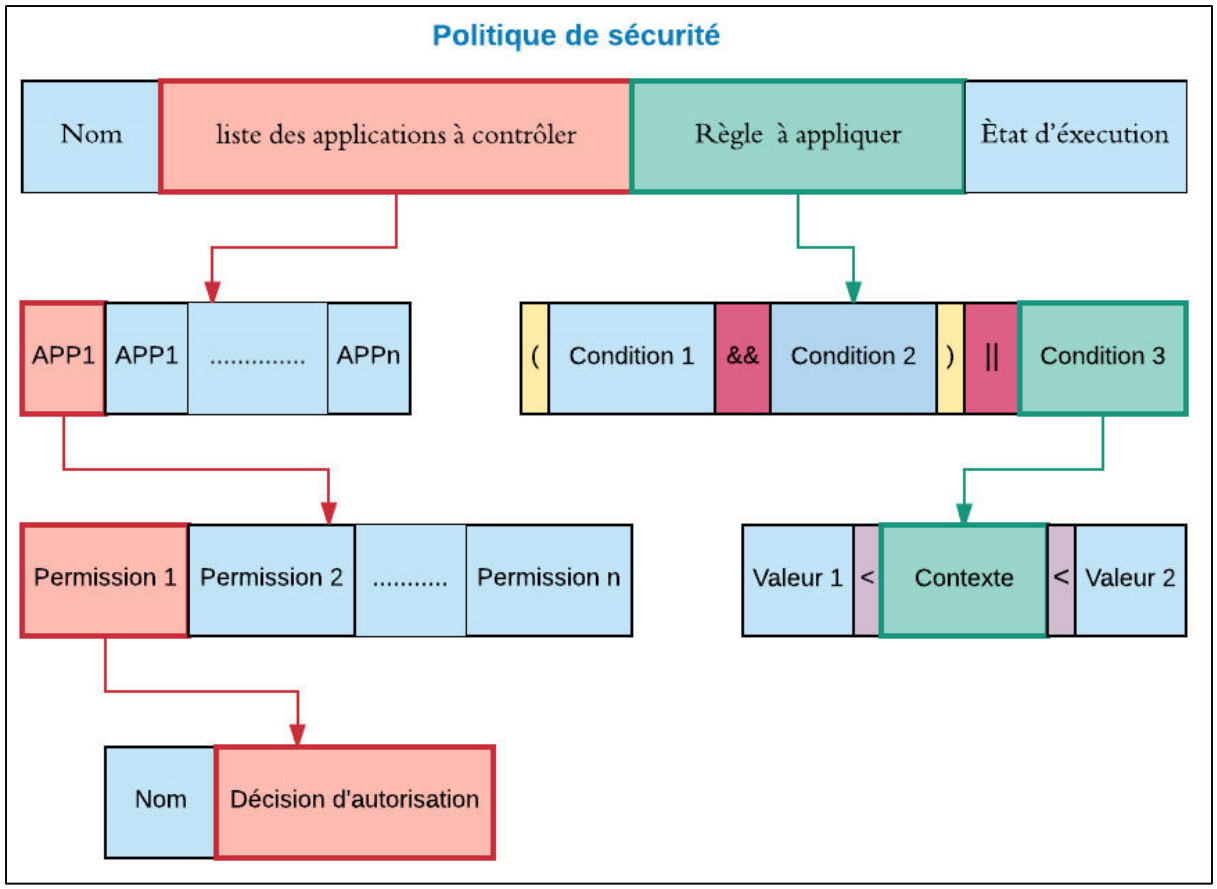


Figure 3.13 Le format principal d'une politique de sécurité

Pour montrer l'utilité de notre solution, nous avons choisi des exemples des scénarios dangereux et nous allons les traduire en des politiques de sécurité.

Scénario dangereux 1 : Si l'utilisateur utilise un réseau wifi public dans un café, Tim Hortons ou Starbucks pour connecter son smartphone sur Internet. Ensuite, il tente de s'authentifier pour consulter ces données dans son application bancaire BNC alors que le réseau public n'est pas sécurisé.

- **Solution :** Il faut empêcher l'application bancaire BNC de se connecter sur internet si l'utilisateur est localisé dans Tim Hortons ou Starbucks.
- **Politique sécuritaire** : si (GPS = TimHortons_location || GPS = Starbucks_location) alors, révoquer la permission (INTERNET dans [BNCBanque]).

Scénario dangereux 2 : Si l'utilisateur utilise son application bancaire BNC pour consulter ses données et déposer un chèque dans son compte, alors qu'une application TakeScreenshot est installée sur son smartphone. Cette application qui prend des captures d'écrans automatiques présente un risque sur les données bancaires qui sont des données de vie personnelle.

- **Solution :** Il faut empêcher l'application TakeScreenshot de s'exécuter lorsque l'utilisateur est en train d'utiliser son application bancaire BNC.
- **Politique sécuritaire :**
  si ([BNCApp] en cours d'exécution), arrêter [TakeScreenShot].

Scénario dangereux 3 : Si l'utilisateur est dans une réunion privée de travail chaque lundi de 9 :00 jusqu'à 10 :00 par Skype alors que beaucoup d'autres applications sont capables de lui enregistrer le discours et le partager en public comme les applications RecordVoice et RecordCall.

- **Solution :** Il faut empêcher les applications RecordVoice et RecordCall de s'exécuter lorsque l'utilisateur est en train de faire un appel sur Skype de 9 :00 jusqu'à 10 :00.
- **Politique sécuritaire :** si ((CALL_PHONE dans [Skype]) &&
  (9 :00<=Temps_actuel<=10 :00)), arrêter [RecordVoice && RecordCall].

Ces exemples de scénario peuvent être dangereux pour la plupart des utilisateurs par contre d'autres les trouvent inutiles. C'est pour cela, notre solution permet aux utilisateurs de traduire n'importe quels scénarios dangereux en des politiques de sécurité.

Nous remarquons toujours que le changement du contexte a une influence sur la sécurité des applications. Nous appliquons nos politiques sur des contextes qui sont les plus connus, les plus utilisés, qui touchent la vie privée de l'utilisateur et qui demandent son autorisation.

Sauf que, beaucoup d'autres contextes qui existent et parfois les données personnelles de l'utilisateur peuvent être leur même des contextes qui menacent la sécurité de sa vie privée. La Figure 3.14 montre un exemple de définition d'une politique de sécurité avec notre

contrôleur. Le scénario du contrôle consiste à bloquer l'exécution de la ressource Camera dans l'application Camera si l'utilisateur est dans une réunion du 10 :30 à 11 :30 ou du 13 :30 à 15 :30 sinon s'il est dans une réunion de 15 :30 à 15 :40 et que le Bluetooth est activé dans l'application BluetoothShareApplication.

Figure 3.14 Exemple de définition d'une politique de sécurité

L'interface graphique gérée par le contrôleur était développée sur mesure pour résoudre nos exigences. D'une façon générale, l'utilisateur sera capable de définir n'importe quels politiques de sécurité en quelques simples cliques. Nous avons choisi que notre solution soit ergonomique, personnalisée et conforme au design centré utilisateur pour avoir un service pratique et facile à utiliser. Nous avons pris aussi en considération que notre interface d'utilisateur doit réduire l'effort de recherche et limite l'entrée des données. C'est pour cela que nous avons éliminé les champs de texte dans l'interface utilisateur pour éviter au

maximum des entrées non contrôlées et nous étions basé seulement sur des composants graphiques actifs comme des listes de sélection, boutons, etc.

Le contrôle appliqué par les politiques de sécurité va être affiché sur les applications instrumentées *voir* Figure 3.15.

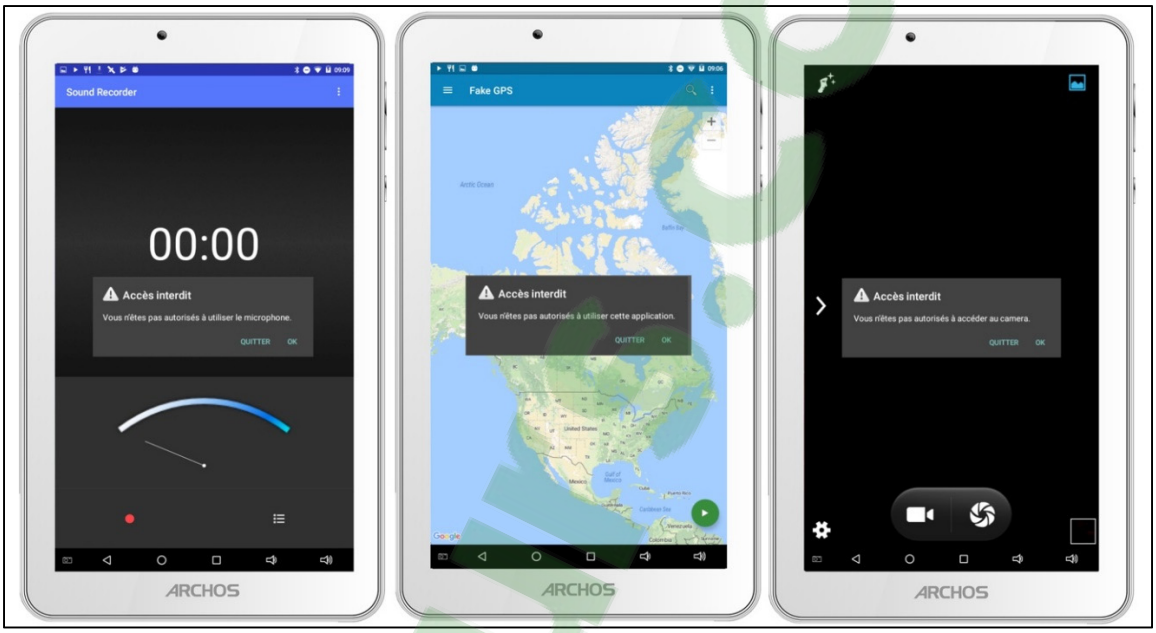


Figure 3.15 Exemples des contrôles appliqués sur des applications de l'utilisateur

3.6.2 Sauvegarde des politiques de sécurité

Pour que notre solution soit forte, nous avons pensé que l'utilisateur peut définir plusieurs politiques de sécurité et que ces politiques peuvent s'exécuter en même temps. Donc, il est important de garder la trace de ce que l'utilisateur a fait pour qu'il soit capable de gérer ces contrôles et les résoudre en cas de problèmes. Aussi, les politiques de sécurité doivent être enregistrées et rester en exécution en arrière-plan même si l'utilisateur a fermé le moniteur.

Dans ce cas, toutes les politiques créées par l'utilisateur doivent être sauvegardés dans une base de données. Avec la base de données, l'utilisateur sera capable d'importer, créer,

consulter, et modifier les politiques de sécurité rapidement ainsi que gérer efficacement et facilement une grande quantité d'informations.

La Figure 3.16 présente la liste des politiques de l'utilisateur, vous remarquez que l'utilisateur peut modifier, supprimer, mettre les politiques en exécution ou en pause.

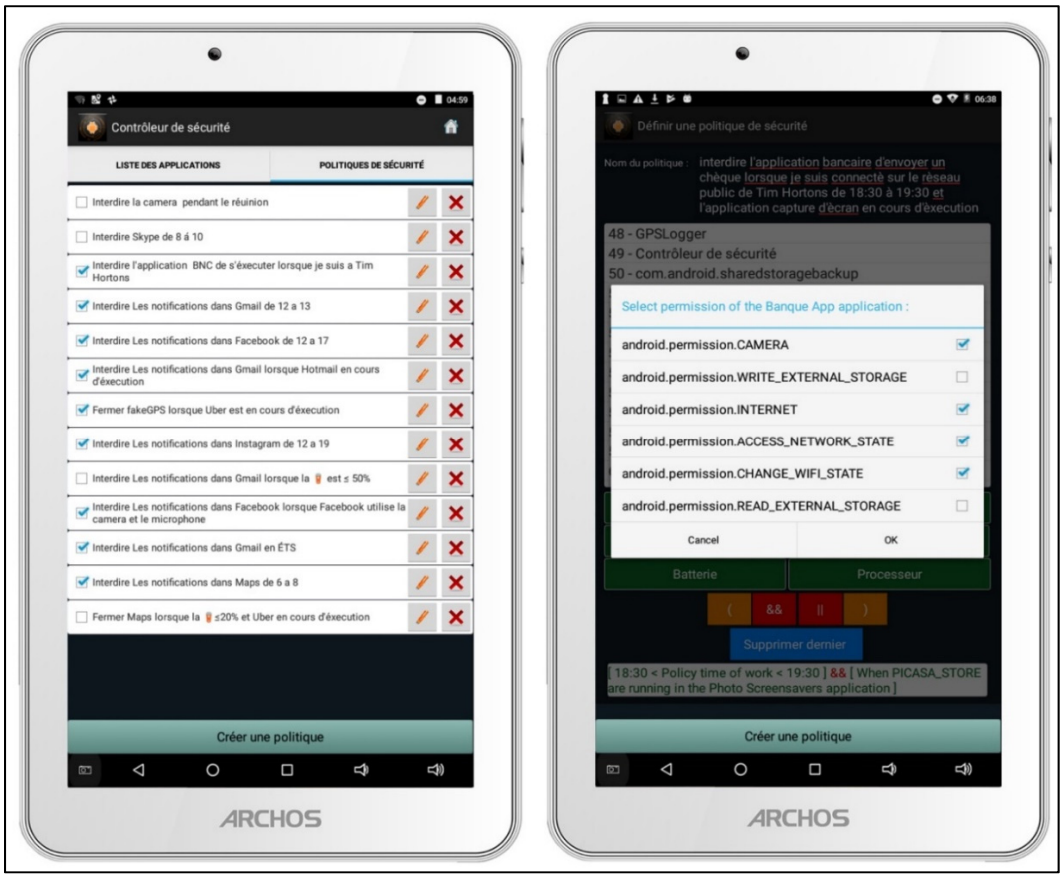


Figure 3.16 L'interface de gestion des politiques de sécurité

Lorsque les politiques de sécurité sont sauvegardées dans une base de données, l'utilisateur sera capable de les extraire et de les partager.

3.6.3 Partager des politiques

Notre solution permet à l'utilisateur d'extraire ses politiques déjà définies pour les partager avec d'autres utilisateurs du contrôleur ou les envoyer au serveur, une base de données sur le cloud ainsi qu'il peut les appliquer sur d'autres contrôleurs différents que notre contrôleur sur le smartphone Android. Sachant que notre langage de politiques est le seul compilé par le système d'Android et que la majorité des applications mobiles consomment des services de l'extérieur. Pour tout autre langage de politiques de l'état de l'art, il est nécessaire de trouver une solution de traduction et de communication.

Pour que notre langage des politiques de sécurité sur Android soit capable d'échanger des politiques avec d'autres langages sur d'autres systèmes installés sur Cloud, Gateway, iPhone, capteurs, etc., nous avons commencé par étudier les langages existants pour garder la même architecture et structure du langage.

Dans la section de revue de littérature, nous avons discuté des exemples de langages de politique, le XACML est le plus populaires entre eux. XACML est une norme OASIS qui exprime les politiques de contrôle d'accès et de sécurité basées sur le format XML.

Nous avons pensé au début à utiliser le langage XACML dans notre contrôleur et à ajouter le principe du contrôle selon un contexte précis, sauf que le système d'Android ne compile pas ce langage. C'est pour cela, nous avons créé notre langage d'une façon intelligente et de façon que les politiques seront traduites en langage java pour les exécuter et puis nous allons les extraire sur différents langages comme XML, JSON etc. De la même façon, si nous importons des politiques de l'extérieur, nous les convertissons en java pour les exécuter, nous les sauvegardons dans la base des données et nous pouvons les extraire en d'autres langages. Ce processus est détaillé dans la Figure 3.17.

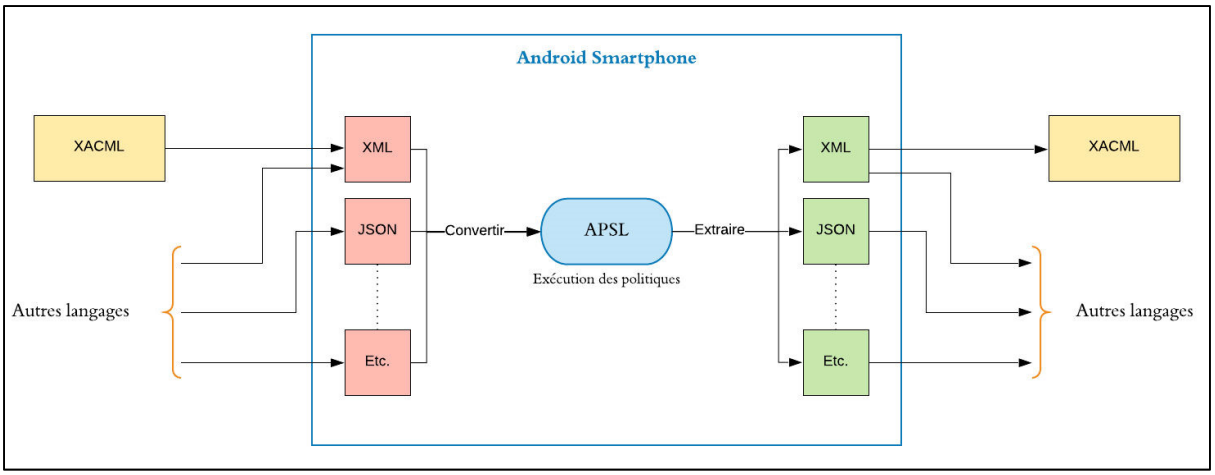


Figure 3.17 Importation et exportation des politiques de sécurité

Pour que notre langage soit compatible avec les autres langages, nous avons gardé la même structure de présentation, les objets et les attributs qui sont comme suit :

- **Policy−set** : présente un tableau qui regroupe la liste des politiques.
- **Policy** : présente l'objet de la politique qui contient les sous-objets « Target » et « Rule », ainsi que les attribues : « Combine » qui présente le rôle de la politique, l'attribut « AUTHOR-KEY-CN » présente l'identifiant de l'auteur de la politique et l'attribut « AUTHOR-KEY-FINGERPRINT » présente la clé empreinte digitale de l'auteur de la politique.
- **Target** : c'est l'objet qui contient la définition des applications ciblées par le contrôleur.
- **Rule** : c'est l'objet qui définit la règle de sécurité, l'attribut « effect » présente la décision du contrôle de donner ou retirer l'autorisation.
- **Condition** : contient les autorisations à retirer et les contextes.
- **Ressource-match** : contient différents attributs : « attr » qui peuvent être une application à bloquer ou une autorisation d'api, « subject-match » contient l'application à contrôler et « match » contient l'autorisation à retirer.

Parmi les valeurs que nous pouvons affecter aux attributs comme les noms des applications, les permissions, etc., nous avons défini des symboles qui nous permettent de simplifier les règles, par exemple :

- **ANY** : ça veut dire aucune, par exemple un contexte de type location, nous n'avons pas besoin d'une permission dans cette condition.
- **ALL** : ça veut dire que nous voulons contrôler toutes les permissions ou toutes les applications ça dépend des attributs utilisés.
- **APPS** : ça veut dire que nous voulons forcer l'arrêt d'une application.
- **API** : ça veut dire que nous allons appliquer le contrôle sur une permission d'API.

Pour transformer le OU et le ET en langage textuel, nous avons utilisé une technique facile.
Le cas où nous déclarons un objet qui contient deux sous objets c'est-à-dire qu'il y'a une relation ET entre les objets par exemple : <Contexte> <condition1/><condition2/><Contexte/> d'où le contexte = condition1 && condition 2.

Dans le cas où chaque objet contient un sous-objet donc ça sera une relation OU par exemple : <Contexte><condition1/><Contexte/> <Contexte><condition2/><Contexte/> d'où le contexte = condition1 || condition 2.

Le contrôleur permet à l'utilisateur de partager ces politiques de sécurité à travers l'envoi direct d'un e-mail joint d'un fichier qui contient les politiques à échanger sinon il est capable d'extraire le fichier dans la mémoire du smartphone et l'échanger manuellement par câble ou Bluetooth, ou de télécharger les politiques directement sur un serveur à travers des services web.

Notre solution permet actuellement d'extraire les politiques en différents langages pour communiquer avec les autres langages de politiques comme XML et JSON.

Nous supposons le scénario de contrôle suivant :

- **Scénario** : Interdiction de lancer l'application TakeScreenShot qui permet de prendre des captures d'écrans automatiques lorsque l'utilisateur ouvre la camera dans son application bancaire de BNCBanque pour envoyer un Chèque.
- **Politique** : si (CAMERA dans [BNCApp]), alors arrêtez l'application [TakeScreenShot].

La Figure 3.18, présente l'extraction de la politique de sécurité suivante en langage XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<policy combine="deny-overrides" id="1" AUTHOR-KEY-CN="Mahdi" AUTHOR-KEY-FINGERPRINT="Mahdi">
   <target>
      <subject>
         <subject-match attr="id_ScreenShot" match="com.apps.TakeScreenShot" />
         <subject-match attr="id_BNC" match="com.apps.BNCbanque" />
      </subject>
   </target>
   <rule effect="deny">
      <condition>
         <ressources>
            <ressource>
               <ressource-match attr="APPS" subject-match="id_ScreenShot" match="ALL" />
            </ressource>
         </ressources>
         <contexts>
            <context>
               <context-match attr="UsedRessources" subject-match="id_BNC" match="android.permission.CAMERA" />
            </context>
         </contexts>
      </condition>
   </rule>
</policy>
```

Figure 3.18 Un exemple d'une politique de sécurité sous le format XML

La Figure 3.19 présente l'extraction de la même politique de sécurité en langage JSON :


```
{
    "@combine": "deny-overrides",
    "@id": "1",
    "@AUTHOR-KEY-CN": "Mahdi",
    "@AUTHOR-KEY-FINGERPRINT": "Mahdi",
    "target": {
        "subject": [
            {
                "@attr": "id_ScreenShot",
                "@match": "com.apps.TakeScreenShot"
            },
            {
                "@attr": "id_BNC",
                "@match": "com.apps.BNCbanque"
            }
        ]
    },
    "rule": {
        "@effect": "deny",
        "condition": {
            "ressources": {
                "ressource": {
                    "ressource-match": {
                        "@attr": "APPS",
                        "@subject-match": "id_ScreenShot",
                        "@match": "ALL"
                    }
                }
            },
            "contexts": {
                "context": {
                    "context-match": {
                        "@attr": "UsedRessources",
                        "@subject-match": "id_BNC",
                        "@match": "android.permission.CAMERA"
                    }
                }
            }
        }
    }
}
```


Figure 3.19 Un exemple d'une politique de sécurité sous le format JSON

Notre solution permet de présenter la politique de sécurité en n'importe quel langage souhaité et c'est ce qui rend notre contrôleur capable de communiquer avec tous les autres langages de politiques existantes. Ainsi, l'utilisateur peut échanger des politiques avec d'autres utilisateurs, télécharger une politique de l'internet par exemple sous n'importe quel langage de politique, l'importer dans son contrôleur, la modifier et l'exécuter.

3.7 Conclusion

Dans ce chapitre, nous avons exposé notre solution proposée basée sur les travaux utilisés afin de sécuriser le système d'exploitation Android. Tout d'abord, nous avons commencé par définir notre langage de spécification des politiques de sécurité, ses composants, et son rôle important afin d'appliquer des politiques de sécurité flexibles et déclaratives.

En deuxième lieu, nous avons présenté l'architecture de notre solution proposée et l'utilité du langage APSL dans notre solution, puis nous avons bien détaillé les deux principaux axes de notre solution, soient, l'approche de réécriture et le contrôleur centralisés des applications Android. Ensuite, nous avons détaillé chacun de ces axes, notre solution de récriture en se basant sur le technique d'injection des aspects afin d'instrumenter des applications Android prêtes au contrôle souhaité appliqué par l'utilisateur, puis, nous avons exposé le contrôleur d'accès qui se base sur APSL afin de contrôler ces applications instrumentées.

Finalement, nous avons présenté des exemples des scénarios dangereux. Les politiques de contrôle qui sécurisent ces scénarios aussi que des exemples de partages de ses politiques de sécurité avec d'autres utilisateurs de notre contrôleur ou d'autres contrôleurs l'extérieur tel que les nuages.

Dans le chapitre suivant, nous présenterons l'aboutissement de notre Framework de réécriture, les résultats issus du contrôleur d'accès des applications ainsi que les informations détaillées de la performance de notre langage de politique afin d'évaluer notre système de sécurisation des applications Android élaboré.

CHAPITRE 4

TEST ET EXPÉRIMENTATIONS

4.1 Introduction

Le but de nos expériences est de présenter les résultats issus du contrôleur d'accès des applications ainsi que les informations détaillées de la performance de notre langage de politique afin d'évaluer notre système de sécurisation des applications Android.

Les expériences ont été faites sur un Samsung Galaxy S5 doté du système d'exploitation Android 4.4 kitkat, avec 2G de RAM, un processeur Qualcomm à 2.5 GHz et une mémoire interne de 16G.

Pour évaluer notre travail, nous avons effectué les expérimentations sur les deux parties de la solution qui sont la réécriture des applications et le contrôleur des applications. Comme nous l'avons déjà détaillé dans le troisième chapitre, la technique de réécriture des applications consiste à injecter des fonctions de contrôle avant chaque appel d'API et un écouteur qui reçoit la décision du contrôle de la part du contrôleur. Pour cela, dans cette partie, nous allons expérimenter notre Framework de réécriture sur plusieurs applications avec différents types et nombres d'APIs pour mesurer le temps d'exécution nécessaire pour appliquer le contrôle. Les métriques considérées durant cette étude sont les tailles des APKs des applications avant et après la réécriture, la taille du code injecté ainsi que le pourcentage du code injecté par rapport les applications originales.

Pour évaluer la deuxième partie et pour identifier les limites du contrôleur, nous avons commencé nos expériences par la définition des échantillons des politiques de plus en plus complexes avec différents contextes, des permissions normales et d'autres dangereuses. Les métriques considérées sont le temps d'exécution pour la prise de décision du contrôle de chaque politique aussi que sa taille.

4.2 Réécriture des applications

Afin que notre contrôleur soit capable de superviser l'accès aux permissions des applications concernées, un BroadcastReceiver doit être implémenté dans chacune d'elle. En effet, ce BroadcastReceiver serait toujours en attente de recevoir les ordres envoyés par le contrôleur qu'il doit les exécuter dès la réception des notifications. Pour chaque application, un seul BroadcastReceiver doit être implémenté dans le but de mettre à jour automatiquement le droit d'accès par permission.

Au cours de cette partie, nous allons présenter le temps de traitement nécessaire pour réécrire l'application ainsi que la nouvelle taille après l'ajout du récepteur.

Un ensemble de 109 applications a été adopté pour ce mécanisme. Ces applications ont été choisies selon leurs nombres de permissions, leurs groupes, leur type d'API (normal et dangereux) et suivant différents contextes.

Le Tableau 4.1 présente dix-huit applications comme un échantillon choisi pour montrer le calcul effectué pour le temps d'interception du contrôle et la taille ajoutée.

Tableau 4.1 La liste des APIs existantes dans un échantillon de dix-huit applications

	CONTACTS	CAMERA	LOCATION	INTERNET	WIFI	PHONE	STORAGE	MICROPHONE	SMS	CALENDAR
Contact Identicons	✓									
Camera		✓								
GPS tracker			✓	✓						
Show web view				✓	✓					
Contact Search	✓									
Contacts Widget	✓					✓				
Beta Updater for WhatsApp				✓						
Contact loader	✓									
Photo Manager				✓	✓		✓			
Wi-Fi setup				✓	✓					
Time tracker							✓			
Calendar Trigger				✓		✓	✓			✓
Calendar Color										✓
Calendar Import-Export				✓		✓	✓			✓
CamTimer		✓					✓			
OpenCamera		✓	✓				✓	✓		
Microphone								✓		
SMS backup	✓			✓	✓	✓	✓		✓	✓

Comme représenté par le tableau 4.1, chacune de ces applications utilise un ou plusieurs APIs. Comme détaillé dans le chapitre précèdent, nous avons concentré notre choix sur les applications qui demandent des permissions dangereuses qui sont les APIs de CONTACTS, CAMERA, LOCATION, PHONE, STORAGE, SENSORS, MICROPHONE, SMS et CALENDAR. Nous avons aussi couvert d'autres permissions classées par le système d'Android comme des permissions normales qui sont : INTERNET et WIFI.

Dans le Tableau 4.2, nous allons spécifier les permissions contrôlées pour chaque API utilisé dans ces applications.

Tableau 4.2 Liste des permissions contrôlées pour chaque API utilisé dans l'échantillon

API	Permissions
CONTACTS	• android.permission.READ_CONTACTS • android.permission.WRITE_CONTACTS • android.permission.PICK_CONTACT • android.permission.GET_ACCOUNTS
CAMERA	• android.permission.CAMERA
LOCATION	• android.permission.ACCESS_FINE_LOCATION
INTERNET	• android.permission.INTERNET • android.permission.ACCESS_NETWORK_STATE
WIFI	• android.permission.CHANGE_WIFI_STATE
PHONE	• android.permission.CALL_PHONE • android.permission.READ_PHONE_STATE
STORAGE	• android.permission.READ_EXTERNAL_STORAGE • android.permission.WRITE_EXTERNAL_STORAGE
SENSORS	• android.permission.VIBRATE • android.permission.WAKE_LOCK
CALENDAR	• android.permission.READ_CALENDAR • android.permission.WRITE_CALENDAR
MICROPHONE	• android.permission.RECORD_AUDIO
SMS	• android.permission.READ_SMS • android.permission.WRITE_SMS • android.permission.RECEIVE_SMS

4.2.1 Temps d'exécution du contrôle des autorisations

Dans cette partie, nous allons présenter le temps de traitement nécessaire pour mettre à jour les décisions du contrôle des autorisations dans chaque application. Le Tableau 4.3 illustre l'échantillon des applications choisies et leur temps d'exécution de mise à jour d'autorisations.

Tableau 4.3 Temps d'exécution de la mise à jour des autorisations

Applications	Temps d'exécution (ms)
Microphone (1 API)	8
Contact Identicons (1 API)	10
Camera (1 API)	12
Contact Search (1 API)	14
Calendar Color (1 API)	16
Beta Updater for WhatsApp (1 API)	17
Contact loader (1 API)	19
GPS trakcer (2 API)	21
CamTimer (2 API)	22
wifi setup (2 API)	24
Show web view (2 API)	26
Time tracker (2 API)	30
Contacts Widget (2 API)	33
Photo Manager (3 API)	46
OpenCamera (4 API)	48
Calendar Trigger (4 API)	50
Calendar Import Export (5 API)	55
SMS backup (7 API)	87

Nous pouvons bien remarquer d'après le tableau que les résultats du temps d'exécution du contrôle des autorisations obtenues sont trop petits. Ceci est un bon avantage puisque le temps d'exécution de l'application après la réécriture est presque le même.

Le temps de mise à jour des autorisations diffère d'une application à une autre selon son type et le nombre d'API à contrôler. Nous avons classé les applications en ordre croissant selon leur nombre d'API utilisés ensuite selon leur temps d'exécution.

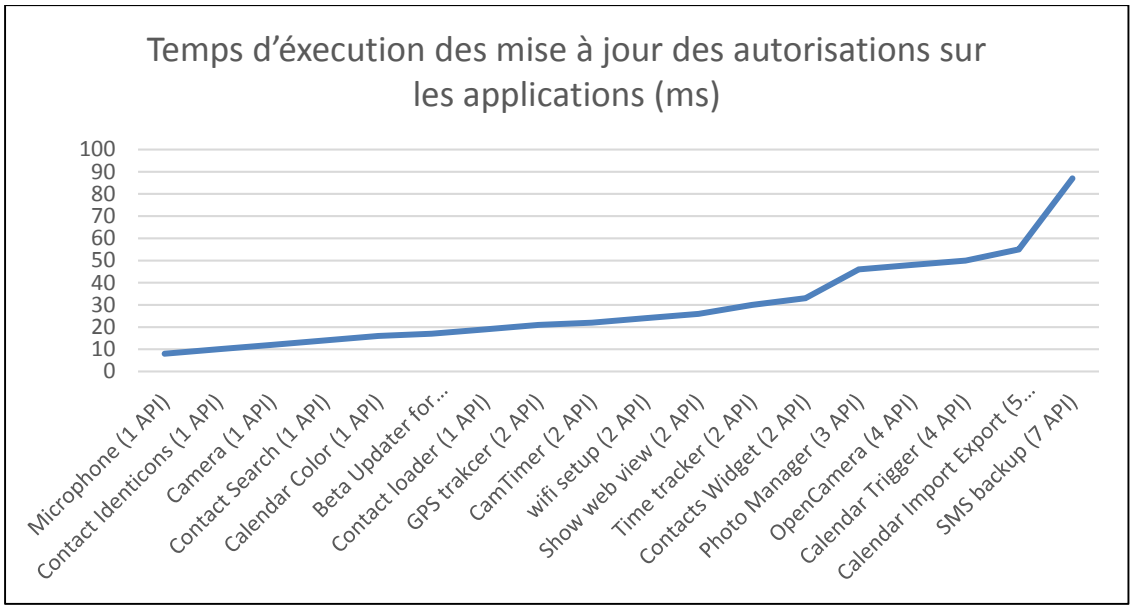


Figure 4.1 Temps d'exécution de la mise à jour des autorisations pour chaque application

Nous pouvons bien remarquer à partir de la Figure 4.1 que la progression du temps d'exécution des mises à jour des autorisations augmente suivant le nombre des APIs à contrôler par application.

4.2.2 Taille des applications

La contrainte la plus connue dans les périphériques mobiles est la limitation des ressources. Il est alors nécessaire d'effectuer la mesure de l'effet de l'instrumentation des applications sur le système installé. Nous devons bien tenir compte du fait que le mécanisme de réécriture ne garde pas la même taille de l'application à cause des mises à jour requis par le contrôleur et le traitement pour révoquer les permissions.

Comme noté dans le chapitre précèdent, l'injection du code source pour toutes les applications a été effectué automatiquement avec un seul fichier AspectJ qui s'exécute lors de la compilation.

Nous avons trouvé que c'était nécessaire de mesurer la taille des applications instrumentées

ainsi que le code rajouté. Dans le but d'éclaircir la notion de la variation des tailles, nous avons réécrit un ensemble de 109 applications en utilisant notre Framework, tout en gardant un œil sur leurs tailles originales et instrumentés.

Le Tableau 4.4 explique la différence de tailles des dix-huit applications, la taille de l'APK original de l'application, et celui instrumenté. Nous présentons aussi les pourcentages des tailles rajoutés aux applications instrumentés par rapport à leurs tailles d'origine. En effet, ce pourcentage représente la taille du code ajouté lors du contrôle des appels aux APIs.

Pour les 109 applications utilisées, les résultats du tableau 4.4 montre que la taille moyenne ajoutée est de 705 octets, qui est à l'ordre de 0.063% de la taille moyenne des applications originales.

Tableau 4.4 La taille des applications avant et après la réécriture

Application	Originale (octets)	Instrumenté (octets)	Taille rajoutée (octets)	Pourcentage rajouté
Contact Identicons	246904	247965	603	0.24%
GPS trakcer	22420823	22421668	845	0.0037%
Show web view	483839	484460	621	0.12%
Contact Search	368610	369278	668	0.18%
Contacts Widget	227386	228034	648	0.28%
Beta Updater for WhatsApp	321673	322448	775	0.24%
Contact loader	2701541	2702019	478	0.01%
Photo Manager	366100	366668	568	0.15%
wifi setup	2177239	2177747	508	0.023%
Time tracker	1477841	1478711	870	0.058%
Calender Trigger	119863	120732	869	0.72%
Calender Color	629361	630232	871	0.13%
Calender Import Export	45823	46772	949	2.07%
CamTimer	1580753	1581393	640	0.04%
OpenCamera	226585	227420	835	0.36%
Microphone	1905254	1905910	656	0.034%
SMS backup	26071	26984	913	3.5%

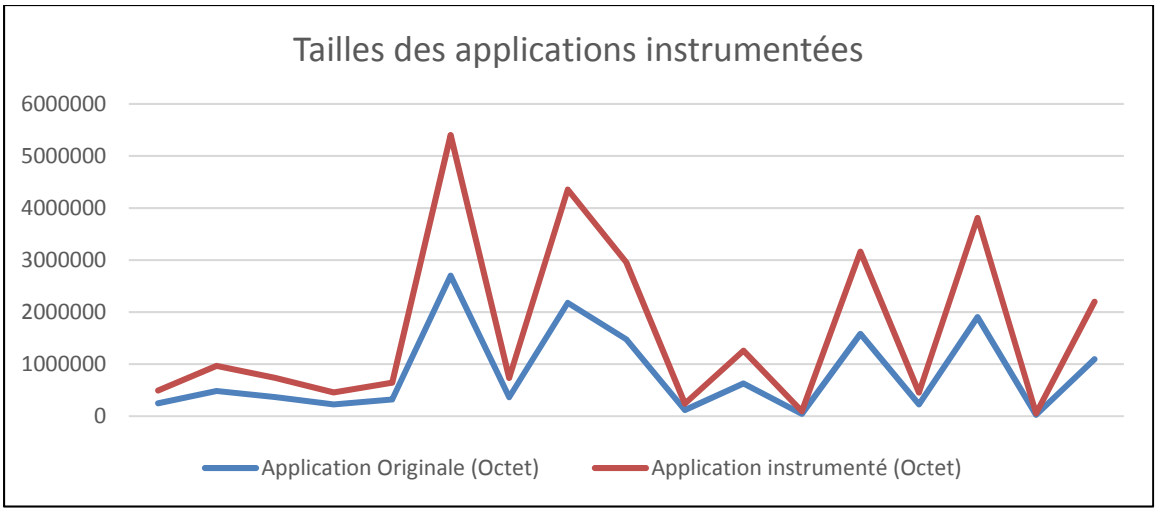


Figure 4.2 Taille des applications avant et après la réécriture

Comme c'est indiqué dans la Figure 4.2, le taille des applications instrumentées augmente par rapport à la taille des applications originales. Cependant, cette augmentation est presque proportionnelles à la taille originale pour toutes les applications comme c'est indiqué dans la Figure 4.3. Le code ajouté moyen est mesuré à 705 octets, celui-ci est trop bas.

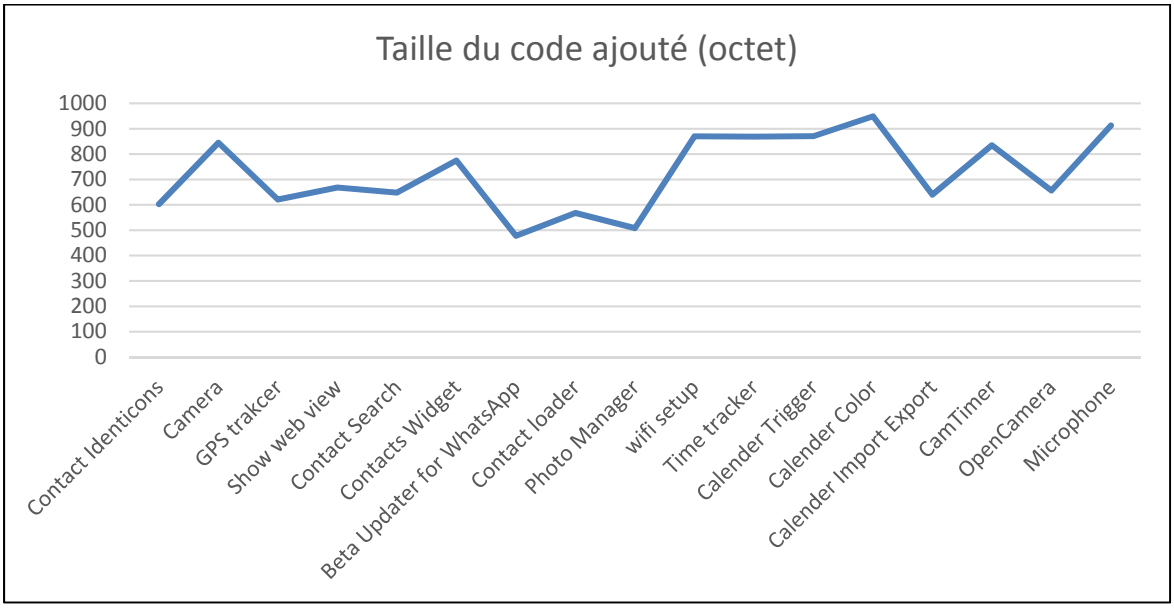


Figure 4.3 Taille (en octet) du code ajouté par après la réécriture des applications

4.3 Contrôleur d'applications

Dans cette étude, nous avons développé un contrôleur d'applications qui se base essentiellement sur notre langage de contrôle d'accès APSL et qui offre à l'utilisateur l'occasion de définir un ensemble de politiques dans le but de bien contrôler les applications déjà installées dans notre plateforme Android. De ce fait, nous avons réalisé des tests de fonctionnalité pour vérifier l'analyse et l'implémentation des politiques de sécurité par notre langage de contrôle d'accès.

La réalisation des tests est faite sur un ensemble d'applications instrumentées par notre solution de réécriture. Au cours de notre étude, nous avons choisi d'implémenter un ensemble de politiques de sécurité dans un contrôleur qui génère des logs déjà analysés.

L'objectif de cette recherche est de vérifier que les permissions nécessaires pour les applications ont bel et bien été contrôlées en conformité avec les politiques de sécurité.

Notre langage de contrôle d'accès a introduit des politiques de sécurité qui dépendent des permissions, des applications exécutées, des APIs en cours d'exécutions et différentes autres contraintes comme le temps, la date, la localisation etc.

La politique de sécurité est une traduction d'une règle de sécurité textuelle écrite par l'utilisateur contre un scénario dangereux en un contrôle sécurisé. Plus précisément, la politique présente une règle ou un ensemble de règles tout en se basant sur un ensemble de conditions. Ces dernières permettent essentiellement à l'utilisateur de mettre en place les scénarios de contrôle désirés. Du coup, le contrôleur doit être capable de transformer tous les scénarios dangereux qui menacent la vie privée de l'utilisateur en des contrôles de sécurité. Plus précisément, la politique présente une règle ou un ensemble de règles en se basant sur un ensemble des conditions.

Les politiques de contrôles d'accès se basent sur une combinaison de plusieurs conditions en utilisant les deux opérateurs logiques "ET" et "OU" ainsi que la capacité de spécifier la priorité d'exécution des conditions par l'ajout des parenthèses " (" et ")".

Chaque politique est caractérisée par un nom qui donne une signification à son rôle, une liste des applications à contrôler, une règle de la sécurité à appliquer et son statut d'exécution. Ce dernier présente l'état de la politique s'il est en cours d'exécution ou en pause selon le choix de l'utilisateur. La règle contient le scénario du contrôle sous la forme de différentes conditions séparées par des parenthèses pour montrer la priorité entre eux ainsi que des opérateurs logiques.

Voici cinq scénarios et leur écriture sous la forme des politiques de sécurité choisies parmi l'ensemble des scénarios utilisés lors des tests. Ces scénarios seront classés en ordre croissant selon leur niveau de complexité.

- **Scénario 1** :
  Interdiction de lancer l'application TakeScreenShot qui permet de prendre des captures d'écrans automatiques lorsque l'utilisateur ouvre la camera dans son application bancaire de BNCBanque pour envoyer un Chèque.
  - **Politique 1** :
  Si (CAMERA dans [BNCApp]), alors arrêtez l'application [TakeScreenShot].

- **Scénario 2** :
  Interdiction de l'application RecordAudioMedia d'enregistrer lorsque l'utilisateur est en train de faire un appel téléphonique via l'application PhoneCall.
  - **Politique 2** :
  Si ((CALL_PHONE && ANSWER_PHONE_CALLS) dans [PhoneCall]), alors arrêter l'application [RecordAudioMedia].

- **Scénario 3** :

  Interdiction de l'application FakeGPS de changer la localisation de l'utilisateur lorsqu'il est en train d'utiliser une ou plusieurs de ces applications BNCBanque, Uber et GoogleMap.

  - **Politique 3** :

  Si [BNCBanque || UBER || GoogleMap], alors retirez les permissions (ACCESS_COARSE_LOCATION && ACCESS_FINE_LOCATION) dans Fake GPS.

- **Scénario 4** :

  Interdiction de l'application BNCBanque d'accéder à Internet ou d'utiliser la camera lorsque l'utilisateur est à TimHortons sachant que sa longitude = 45.491318 et sa latitude = -73.727987.

  - **Politique 4** :

  Si (GPS = [45.491318,-73.72798]), alors retirez les permissions (INTERNET && CAMERA) dans [BNCBanque]

- **Scénario 5** :

  Interdiction des applications Facebook, Instagram et Gmail d'accéder à Internet, utiliser la camera ou localiser l'utilisateur. De plus, interdiction à l'application Message de recevoir des SMS et à l'application RecordAudio d'enregistrer la voix lorsque l'utilisateur est en réunion à l'ÈTS de 8h à 9h.

  - **Politique 5** :

  Si (8<Temps<9 && Location = [45.491318,-73.727987]), alors retirez les permissions ((INTERNET && CAMERA && ACCESS_FINE_LOCATION) dans [Facebook && Instagram && Gmail]) && ((RECEIVE_SMS || RECEIVE_MMS) dans [Message]) && (RECORD_AUDIO dans [RecordAudio]).

Pour valider la performance du notre langage de contrôle d'accès des politiques, les résultats de calcul du temps d'exécution de décision de politique et le taille des politiques sont importants.

4.3.1 Temps d'exécution de décision de politique

Dans cette partie, nous allons présenter le temps de traitement nécessaire pour exécuter des politiques de sécurité définies par notre langage afin obtenir leurs points de décision. Comme déjà expliqué dans le précédant chapitre que le point de décision d'une politique présente un résultat booléen pour savoir si la politique est valide ou non. Pour calculer le temps d'exécution, nous avons utilisé un test préliminaire qui consiste à donner des données aléatoires comme des entrées afin de montrer la performance de notre langage.

Pour les 5 scénarios détaillés, nous avons calculé le temps d'exécution de décision pour chacune des politiques :

Pour la première et la deuxième politiques, les résultats des tests obtenus étaient fixes à cause que le contexte ne varie pas avec les changements de valeurs d'entrées. Donc le temps d'exécution pour la première politique est 116 ms et pour la deuxième politique est 234 ms.

Pour la troisième politique, la différence par rapport aux deux politiques précédentes est que le contexte est lié à 3 applications différentes qui sont en cours d'exécution, mais il reste toujours fixe. Le temps d'exécution pour toute la politique est 307 ms.

Pour la quatrième politique, notre contexte est la localisation donc les résultats étaient plus ou moins proches, mais ils varient selon le changement des valeurs des GPS. Dans ce cas-là le temps moyen d'exécution de toute la politique est 314 ms.

Pour la cinquième politique, deux contextes différents étaient utilisés : le temps et la localisation. Les valeurs d'entrés aléatoires diffèrent d'un test à l'autre, mais sont plus ou moins équivalents. Le temps moyen d'exécution pour notifier chaque application a était aussi calculé. Pour l'application Skype le temps d'exécution est 549 ms, 592 ms pour l'application Messages, 634 ms pour l'application Instagram, 758 ms pour l'application Gmail et 814 pour

Facebook. Aussi que le temps d'exécution moyen pour toute la politique est 818 ms. Le Tableau 4.5 illustre le temps d'exécution par milliseconde pour chaque politique.

Tableau 4.5 Temps d'exécution des politiques de sécurité en milliseconde

Politiques	**Temps d'exécution en millisecondes**
Politique 1	116
Politique 2	234
Politique 3	307
Politique 4	314
Politique 5	818

Les règles des politiques de sécurité définies par l'utilisateur peuvent contenir plusieurs conditions, différents contextes, plusieurs opérateurs logiques et parenthèses pour spécifier la priorité entre les conditions à titre d'exemple la cinquième politique dans la liste des scénarios. Ainsi, plusieurs contextes peuvent être détectés à la fois et plusieurs politiques en cours d'exécution. Dans ce cas, la détermination du point de décision de la politique devient plus complexe. Les politiques de sécurité de l'échantillon des scénarios que nous avons choisis sont triées dans un ordre croissant de la plus simple à la plus complexe.

La Figure 4.4 présente la différence en temps d'exécution selon la complexité pour chacune des politiques.

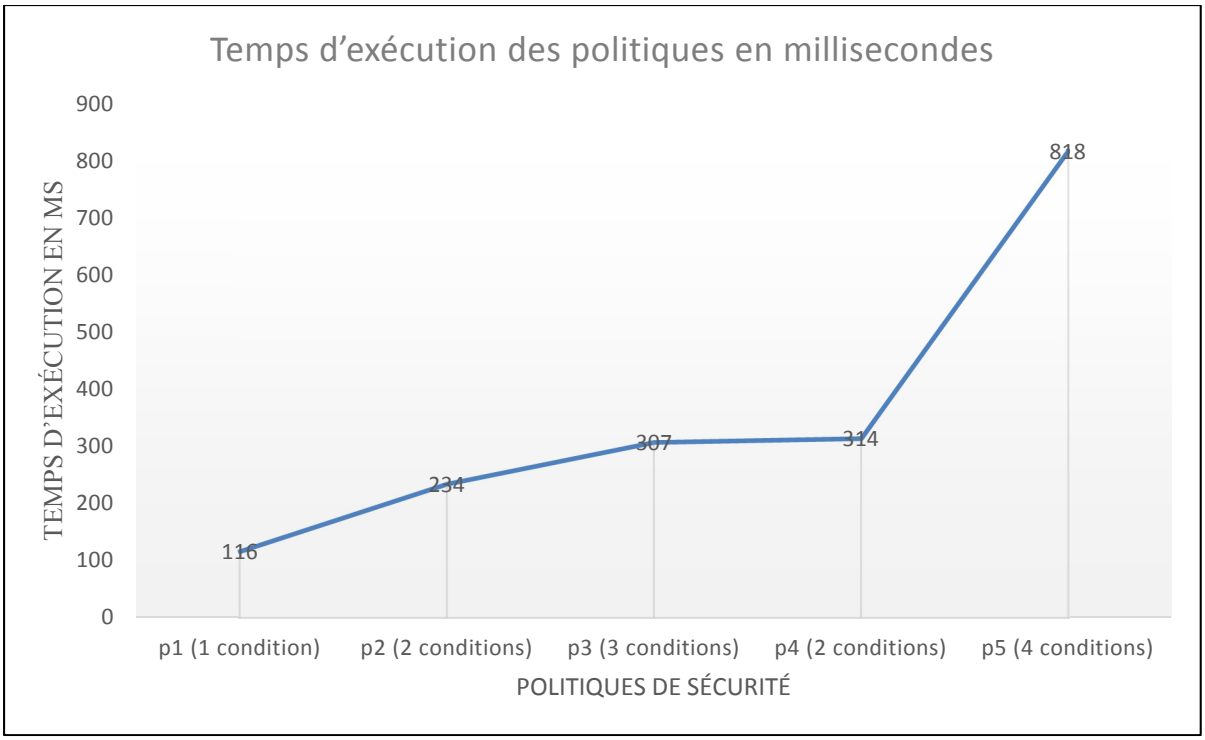


Figure 4.4 Temps d'exécution pour chacune des politiques utilisées

Nous remarquons que le temps d'exécution est différent par chaque politique. Tellement que la politique devient de plus en plus complexe le temps d'exécution devient de plus en plus grand.

4.3.2 Taille des politiques

Comme c'est déjà expliqué dans le chapitre président, les politiques définies par l'utilisateur vont être sauvegardées dans la mémoire de son smartphone. L'utilisateur peut extraire la liste des politiques dans un fichier sous le format XML, JSON etc. De cette manière, il sera capable de les envoyer directement vers Cloud ou d'autres systèmes. Il sera aussi capable de partager ces fichiers avec d'autres utilisateurs qui peuvent importer ces politiques dans leur moniteur.

La Figure 4.5 présente la cinquième politique générée par notre langage des politiques en langage XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<policy combine="deny-overrides" id="5" AUTHOR-KEY-CN="KeyUser" AUTHOR-KEY-FINGERPRINT="FingerPrintUser">
   <target>
      <subject>
         <subject-match attr="id_Facebook" match="com.apps.Facebook" />
         <subject-match attr="id_Instagram" match="com.apps.Instagram" />
         <subject-match attr="id_Gmail" match="com.apps.Gmail" />
         <subject-match attr="id_Message" match="com.apps.Message" />
         <subject-match attr="id_RecordAudio" match="com.apps.recordAudio" />
      </subject>
   </target>
   <rule effect="deny">
      <condition>
         <ressources>
            <ressource>
               <ressource-match attr="API" subject-match="id_Facebook" match="android.permission.CAMERA" />
               <ressource-match attr="API" subject-match="id_Facebook" match="android.permission.INTERNET" />
               <ressource-match attr="API" subject-match="id_Facebook" match="android.permission.ACCESS_FINE_LOCATION" />
               <ressource-match attr="API" subject-match="id_Instagram" match="android.permission.CAMERA" />
               <ressource-match attr="API" subject-match="id_Instagram" match="android.permission.INTERNET" />
               <ressource-match attr="API" subject-match="id_Instagram" match="android.permission.ACCESS_FINE_LOCATION" />
               <ressource-match attr="API" subject-match="id_Gmail" match="android.permission.CAMERA" />
               <ressource-match attr="API" subject-match="id_Gmail" match="android.permission.INTERNET" />
               <ressource-match attr="API" subject-match="id_Gmail" match="android.permission.ACCESS_FINE_LOCATION" />
               <ressource-match attr="API" subject-match="id_Message" match="android.permission.RECEIVE_SMS" />
               <ressource-match attr="API" subject-match="id_Message" match="android.permission.RECEIVE_MMS" />
               <ressource-match attr="API" subject-match="id_RecordAudio" match="android.permission.RECORD_AUDIO" />
            </ressource>
         </ressources>
         <contexts>
            <context>
               <context-match attr="TIME" subject-match="ANY" match="ANY">
                  <target attr="after" match="8" />
                  <target attr="before" match="9" />
               </context-match>
               <context-match attr="LOCATION" subject-match="ANY" match="ANY">
                  <target attr="longitude" match="45.491318" />
                  <target attr="latitude" match="-73.727987" />
               </context-match>
            </context>
         </contexts>
      </condition>
   </rule>
</policy>
```

Figure 4.5 Exemple d'un fichier XML généré à partir d'une politique

Pour des contraintes liées à la taille de la mémoire du smartphone et à l'envoi des politiques vers les nuages, il est important de mesurer les tailles des politiques créées par notre langage. Nos tests pour calculer les tailles des politiques étaient effectués sur une liste de 109 applications. Nous avons pris le même échantillon des cinq politiques que nous avons mesuré leur temps d'exécution dans la section précédente pour faire notre simulation. Les résultats de calcul les tailles des politiques vont être illustré dans le Tableau 4.6.

Tableau 4.6 Taille en octet pour chaque politiques

Politiques	Taille en octets
Politique 1	771
Politique 2	911
Politique 3	940
Politique 4	1139
Politique 5	2399

Comme nous pouvons remarquer, la taille de la politique devient de plus en plus élevée si la politique est plus en plus complexe. La taille totale du fichier qui contient toutes les politiques est mesuré à 5772 octets ce qui est équivalant à 5 kilooctets. La Figure 4.6 suivante présente la progression des tailles des politiques selon leurs complexités.

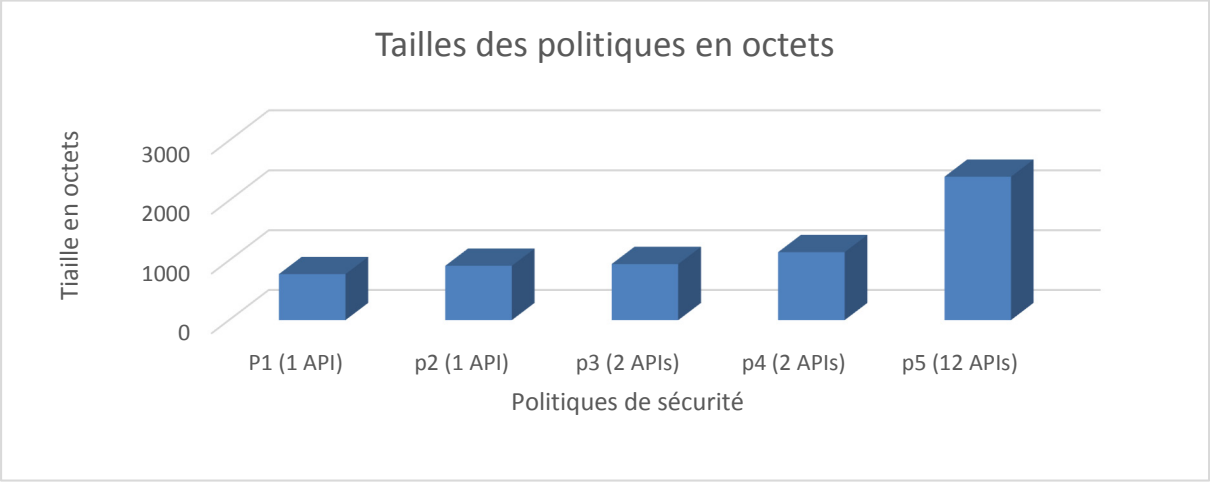


Figure 4.6 Tailles des cinq politiques de l'échantillon
selon leur niveau de complexité

Nous remarquons que les politiques qui dérivent d'un même nombre des contextes et des permissions à révoquer sont proches de la taille même si le nombre des applications à contrôler est différent. Nous prenons à titre d'exemple la deuxième et la troisième politique, elles ont des tailles comparables malgré que la troisième politique contrôle trois applications alors que la deuxième politique contrôle seulement une seule application.

4.4 Discussion

Notre approche qui vise à surveiller l'accès aux méthodes API, contient deux cardes essentielles à noter, le Framework de réécriture et le contrôleur des applications basé sur un langage de spécification des politiques de sécurité que nous avons déjà développé. Durant les expériences élaborées, nous avons pu bien remarquer l'importante efficacité de ces deux axes de notre solution.

Tout d'abord, concernant le Framework de réécriture, des techniques existantes instrumentent les applications manuellement, d'une façon que nous devons parcourir le code source des applications classe par classe et chercher dans les fonctions et les sous fonctions pour trouver les appels des APIs cibles. Cette démarche manuelle, demande un long temps pour parcourir le code et injecter les fonctions de contrôle dans les applications concernées par la sécurité. Par contre, notre solution de réécriture permet d'instrumenter les applications rapidement avec un temps moyen de 23 millisecondes par application. De plus, le code à insérer est trop petit et n'affecte pas trop la taille originale de l'application. La taille moyenne de ce code est environ 705 octets par application. En outre, notre technique est capable de contrôler toutes les APIs dans l'application sans exception qu'elles soient des APIs de protection normale ou dangereuses.

En second lieu, notre solution surpasse le système de permissions granulaire implémenté par le système Android pour permettre aux utilisateurs d'accorder des autorisations aux applications pendant leur exécution. En effet, ce dernier fonctionne seulement à partir du niveau d'API 23 et permet de contrôler seulement les API dangereuses d'une façon manuelle. Par contre, notre contrôleur est fonctionnel avec toutes les versions du système Android et avec tous les niveaux d'APIs et il est capable de contrôler tous les types des APIs de protection normale ou les APIs dangereuses avec les deux manières : manuellement, avec l'interaction de l'utilisateur pendant l'exécution de l'application ou automatiquement en créant des politiques de sécurité qui se déclenchent automatiquement dépendamment du contexte.

Finalement, les approches existantes telles que XACML sont capables de créer des règles de sécurité qui nécessite de customiser le système d'exploitation Android. Par contre notre solution n'exige pas le changement du système Android. En effet, nous avons créé notre premier langage de spécification des politiques de sécurité native qui est compilable sur Android. Ce langage permet à l'utilisateur de créer une infinité de politiques de sécurité qui sont dépendantes du contexte avec n'importe quel niveau de complexité. Le temps d'exécution et les tailles des politiques dépendent de la complexité de la politique. De plus, notre solution permet d'importer et exporter des politiques avec n'importe quel d'autres langages des politiques comme XACML, UMA etc. Cette fonctionnalité permet de partager les politiques avec le monde extérieur tel que d'autres utilisateurs ou des services Cloud.

Il est important de mentionner que la complexité des politiques de sécurités peut créer des conflits dans le partage des ressources. Considérant l'exemple suivant :

- P1 : une première politique de sécurité qui interdit l'utilisation du GPS dans l'application bancaire de 8h00 à 10h00.
- P2 : une deuxième politique qui demande l'autorisation pour accéder aux données GPS dans l'application bancaire dans le ça où l'utilisateur ouvre la caméra pour prendre une photo d'un chèque afin de l'envoyer à la banque.
- P3 : une troisième politique qui interdit d'ouvrir la camera et accéder à l'internet dans l'application bancaire si l'utilisateur est localisé à Tim Hortons ou Starbucks.

Dans cette situation, le conflit des politiques de sécurité reste un chalenge à étudier.

CONCLUSION

Android est devenu le meneur des plateformes mobiles en dominant le marché avec une part estimé à 85,1% en 2017 (Auffray, Chiffres clés : les OS pour smartphones, 2017).Cette position de leadership s'est manifestée par l'augmentation significative des applications Android disponibles sur le marché, mais a causé en contrepartie l'augmentation des taux d'attaques qui cibles cette plateforme. En effet, environ 40% des applications malveillantes existent sur le store d'Android.

Cette foule importante d'attaques continuelles ciblant Android, demeure un énorme risque qui affecte la confidentialité de plusieurs utilisateurs, ainsi, l'économie des différentes organisations utilisant Android dans leurs infrastructures TI. Pour cela, il est assez important de garantir une bonne sécurité à cette plateforme.

De nombreuses applications Android ont besoin d'accéder aux informations privées des utilisateurs pour des fonctionnalités légitimes, notamment que ces applications impliquent souvent le partage d'informations privées entre utilisateurs à des emplacements dispersés. Ce partage d'informations peut causer des risques de divulgation des informations confidentielles des utilisateurs.

Nous avons commencé notre rapport par présenter un aperçu de la plateforme Android, les composants des applications, les logiciels malveillants et les modèles de sécurité. Ensuite, nous avons effectué une revue critique de la littérature de plusieurs travaux récents sur la détection, l'évaluation et l'atténuation des failles de confidentialité dans la plateforme et les applications Android.

Beaucoup d'études étaient faites par de nombreux chercheurs ont discuté la sécurité et la confidentialité sous différents aspects. Elles ont fourni des différentes techniques qui ont été mises en œuvre pour offrir une protection contre les divers logiciels malveillants et la divulgation des données privées de l'utilisateur.

Les travaux que nous avons examinés sont principalement axés sur la sécurité des applications Android en proposant différents modèles et méthodes. Ils sont divisés en trois catégories principales : détection, évaluation et atténuation. De nombreux chercheurs ont proposé des méthodes pour détecter et analyser les fuites de confidentialité, soit de manière statique ou dynamique. Alors que d'autres recherches ont été menées pour atténuer les fuites de confidentialité dans les applications Android en proposant des mécanismes de sécurité nécessitant une modification du système, d'autres ont utilisé la réécriture des applications.

Tous ces travaux examinés ont proposé des solutions utiles pour limiter la divulgation de confidentialité dans les applications Android. Cependant, cet examen a également souligné certaines limites que nous avons prises en compte dans la solution proposée.

Par conséquent, nous proposons un cadre de réécriture des applications pour appliquer efficacement des contrôles à chaque application individuellement sans modifier le système d'Android. En outre, ce cadre applique des stratégies de sécurité inter-applications dépendantes au contexte afin de contrôler les fuites de confidentialité entre les différentes applications Android susceptibles de partager les mêmes ressources. En conséquence, les applications modifiées seront forcées de communiquer avec notre moniteur d'applications centralisées qui permet de définir des politiques de contrôles sécurisés basées sur notre langage de spécification des politiques afin de permettre aux utilisateurs d'interpréter et d'appliquer leurs règles de sécurité complexes sur leurs applications Android. Ce langage de spécification des politiques donne aux utilisateurs la possibilité d'ajouter ou de modifier une politique de sécurité via une interface utilisateur.

Cependant, les contraintes de performances et les ressources limitées comme le CPU, batterie, mémoire, etc., sont les plus gros problèmes de solutions de sécurité. À ce sujet, nous devons s'investir dans d'autres recherches et plus d'enquêtes afin de surmonter les problèmes de nos solutions proposées.

ANNEXE I

RÉSULTATS DE TESTS DE L'ÉCHANTILLON DES CINQ POLITIQUES PROPOSÉES SOUS LA FORME DU LANGUAGE XML

```
<?xml version="1.0" encoding="UTF-8"?>
<policy combine="deny-overrides" id="1" AUTHOR-KEY-CN="Mahdi" AUTHOR-KEY-FINGERPRINT="Mahdi">
   <target>
      <subject>
         <subject-match attr="id_ScreenShot" match="com.apps.TakeScreenShot" />
         <subject-match attr="id_BNC" match="com.apps.BNCbanque" />
      </subject>
   </target>
   <rule effect="deny">
      <condition>
         <ressources>
            <ressource>
               <ressource-match attr="APPS" subject-match="id_ScreenShot" match="ALL" />
            </ressource>
         </ressources>
         <contexts>
            <context>
               <context-match attr="UsedRessources" subject-match="id_BNC" match="android.permission.CAMERA" />
            </context>
         </contexts>
      </condition>
   </rule>
</policy>
```

Figure I.1 La présentation XML de la première politique

```
<?xml version="1.0" encoding="UTF-8"?>
<policy combine="deny-overrides" id="2" AUTHOR-KEY-CN="Mahdi" AUTHOR-KEY-FINGERPRINT="Mahdi">
   <target>
      <subject>
         <subject-match attr="id_PhoneCall" match="com.apps.PhoneCall" />
         <subject-match attr="id_RecordAudio" match="com.apps.RecordAudio" />
      </subject>
   </target>
   <rule effect="deny">
      <condition>
         <ressources>
            <ressource>
               <ressource-match attr="APPS" subject-match="id_RecordAudio" match="ALL" />
            </ressource>
         </ressources>
         <contexts>
            <context>
               <context-match attr="API" subject-match="id_PhoneCall" match="android.permission.CALL_PHONE" />
            </context>
            <context>
               <context-match attr="API" subject-match="id_PhoneCall" match="android.permission.ANSWER_PHONE_CALLS" />
            </context>
         </contexts>
      </condition>
   </rule>
</policy>
```

Figure I.2 La présentation XML de la deuxième politique

```
<?xml version="1.0" encoding="UTF-8"?>
<policy combine="deny-overrides" id="3" AUTHOR-KEY-CN="Mahdi" AUTHOR-KEY-FINGERPRINT="Mahdi">
   <target>
      <subject>
         <subject-match attr="id_FakeGPS" match="com.apps.FakeGPS" />
         <subject-match attr="id_BNC" match="com.apps.BNCbanque" />
         <subject-match attr="id_Uber" match="com.apps.Uber" />
         <subject-match attr="id_GoogleMaps" match="com.apps.GoogleMaps" />
      </subject>
   </target>
   <rule effect="deny">
      <condition>
         <ressources>
            <ressource>
               <ressource-match attr="APPS" subject-match="id_FakeGPS" match="android.permission.ACCESS_FINE_LOCATION" />
               <ressource-match attr="APPS" subject-match="id_FakeGPS" match="android.permission.ACCESS_COARSE_LOCATION" />
            </ressource>
         </ressources>
         <contexts>
            <context>
               <context-match attr="UsedRessources" subject-match="id_BNC" match="ALL" />
            </context>
            <context>
               <context-match attr="UsedRessources" subject-match="id_Uber" match="ALL" />
            </context>
            <context>
               <context-match attr="UsedRessources" subject-match="id_GoogleMaps" match="ALL" />
            </context>
         </contexts>
      </condition>
   </rule>
</policy>
```

Figure I.3 La présentation XML de la troisième politique

```
<?xml version="1.0" encoding="UTF-8"?>
<policy combine="deny-overrides" id="4" AUTHOR-KEY-CN="Mahdi" AUTHOR-KEY-FINGERPRINT="Mahdi">
   <target>
      <subject>
         <subject-match attr="id_BNC" match="com.apps.BNCbanque" />
      </subject>
   </target>
   <rule effect="deny">
      <condition>
         <ressources>
            <ressource>
               <ressource-match attr="API" subject-match="id_BNC" match="android.permission.CAMERA" />
               <ressource-match attr="API" subject-match="id_BNC" match="android.permission.INTERNET" />
            </ressource>
         </ressources>
         <contexts>
            <context>
               <context-match attr="LOCATION" subject-match="ANY" match="ANY">
                  <target attr="longitude" match="45.491318" />
                  <target attr="latitude" match="-73.727987" />
               </context-match>
            </context>
         </contexts>
      </condition>
   </rule>
</policy>
```

Figure I.4 La présentation XML de la quatrième politique

```
<?xml version="1.0" encoding="UTF-8"?>
<policy combine="deny-overrides" id="5" AUTHOR-KEY-CN="Mahdi" AUTHOR-KEY-FINGERPRINT="Mahdi">
   <target>
      <subject>
         <subject-match attr="id_Facebook" match="com.apps.Facebook" />
         <subject-match attr="id_Instagram" match="com.apps.Instagram" />
         <subject-match attr="id_Gmail" match="com.apps.Gmail" />
         <subject-match attr="id_Message" match="com.apps.Message" />
         <subject-match attr="id_RecordAudio" match="com.apps.recordAudio" />
      </subject>
   </target>
   <rule effect="deny">
      <condition>
         <ressources>
            <ressource>
               <ressource-match attr="API" subject-match="id_Facebook" match="android.permission.CAMERA" />
               <ressource-match attr="API" subject-match="id_Facebook" match="android.permission.INTERNET" />
               <ressource-match attr="API" subject-match="id_Facebook" match="android.permission.ACCESS_FINE_LOCATION" />
               <ressource-match attr="API" subject-match="id_Instagram" match="android.permission.CAMERA" />
               <ressource-match attr="API" subject-match="id_Instagram" match="android.permission.INTERNET" />
               <ressource-match attr="API" subject-match="id_Instagram" match="android.permission.ACCESS_FINE_LOCATION" />
               <ressource-match attr="API" subject-match="id_Gmail" match="android.permission.CAMERA" />
               <ressource-match attr="API" subject-match="id_Gmail" match="android.permission.INTERNET" />
               <ressource-match attr="API" subject-match="id_Gmail" match="android.permission.ACCESS_FINE_LOCATION" />
               <ressource-match attr="API" subject-match="id_Message" match="android.permission.RECEIVE_SMS" />
               <ressource-match attr="API" subject-match="id_Message" match="android.permission.RECEIVE_MMS" />
               <ressource-match attr="API" subject-match="id_RecordAudio" match="android.permission.RECORD_AUDIO" />
            </ressource>
         </ressources>
         <contexts>
            <context>
               <context-match attr="TIME" subject-match="ANY" match="ANY">
                  <target attr="after" match="8" />
                  <target attr="before" match="9" />
               </context-match>
               <context-match attr="LOCATION" subject-match="ANY" match="ANY">
                  <target attr="longitude" match="45.491318" />
                  <target attr="latitude" match="-73.727987" />
               </context-match>
            </context>
         </contexts>
      </condition>
   </rule>
</policy>
```

Figure I.5 La présentation XML de la cinquième politique

ANNEXE II

RÉSULTATS DE TESTS DE L'ÉCHANTILLON DES CINQ POLITIQUES PROPOSÉES SOUS LA FORME DU LANGUAGE JSON


```
{
    "@combine": "deny-overrides",
    "@id": "1",
    "@AUTHOR-KEY-CN": "Mahdi",
    "@AUTHOR-KEY-FINGERPRINT": "Mahdi",
    "target": {
        "subject": [
            {
                "@attr": "id_ScreenShot",
                "@match": "com.apps.TakeScreenShot"
            },
            {
                "@attr": "id_BNC",
                "@match": "com.apps.BNCbanque"
            }
        ]
    },
    "rule": {
        "@effect": "deny",
        "condition": {
            "ressources": {
                "ressource": {
                    "ressource-match": {
                        "@attr": "APPS",
                        "@subject-match": "id_ScreenShot",
                        "@match": "ALL"
                    }
                }
            },
            "contexts": {
                "context": {
                    "context-match": {
                        "@attr": "UsedRessources",
                        "@subject-match": "id_BNC",
                        "@match": "android.permission.CAMERA"
                    }
                }
            }
        }
    }
}
```


Figure II.1 La présentation de la première politique en JSON

```
{
    "@combine": "deny-overrides",
    "@id": "2",
    "@AUTHOR-KEY-CN": "Mahdi",
    "@AUTHOR-KEY-FINGERPRINT": "Mahdi",
    "target": {
        "subject": [
            {
                "@attr": "id_PhoneCall",
                "@match": "com.apps.PhoneCall"
            },
            {
                "@attr": "id_RecordAudio",
                "@match": "com.apps.RecordAudio"
            }
        ]
    },
    "rule": {
        "@effect": "deny",
        "condition": {
            "ressources": {
                "ressource": {
                    "ressource-match": {
                        "@attr": "APPS",
                        "@subject-match": "id_RecordAudio",
                        "@match": "ALL"
                    }
                }
            },
            "contexts": [
                {
                    "context-match": {
                        "@attr": "API",
                        "@subject-match": "id_PhoneCall",
                        "@match": "android.permission.CALL_PHONE"
                    }
                },
                {
                    "context-match": {
                        "@attr": "API",
                        "@subject-match": "id_PhoneCall",
                        "@match": "android.permission.ANSWER_PHONE_CALLS"
                    }
                }
            ]
        }
    }
}
```


Figure II.2 La présentation de la deuxième politique en JSON

```
{
    "@combine": "deny-overrides",
    "@id": "3",
    "@AUTHOR-KEY-CN": "Mahdi",
    "@AUTHOR-KEY-FINGERPRINT": "Mahdi",
    "target": {
        "subject": [
            {
                "@attr": "id_FakeGPS",
                "@match": "com.apps.FakeGPS"
            },
            {
                "@attr": "id_BNC",
                "@match": "com.apps.BNCbanque"
            },
            {
                "@attr": "id_Uber",
                "@match": "com.apps.Uber"
            },
            {
                "@attr": "id_GoogleMaps",
                "@match": "com.apps.GoogleMaps"
            }
        ]
    },
    "rule": {
        "@effect": "deny",
        "condition": {
            "ressources": {
                "ressource": [
                    {
                        "@attr": "APPS",
                        "@subject-match": "id_FakeGPS",
                        "@match": "android.permission.ACCESS_FINE_LOCATION"
                    },
                    {
                        "@attr": "APPS",
                        "@subject-match": "id_FakeGPS",
                        "@match": "android.permission.ACCESS_COARSE_LOCATION"
                    }
                ]
            },
            "contexts": [
                {
                    "context-match": {
                        "@attr": "UsedRessources",
                        "@subject-match": "id_BNC",
                        "@match": "ALL"
                    }
                },
                {
                    "context-match": {
                        "@attr": "UsedRessources",
                        "@subject-match": "id_Uber",
                        "@match": "ALL"
                    }
                },
                {
                    "context-match": {
                        "@attr": "UsedRessources",
                        "@subject-match": "id_GoogleMaps",
                        "@match": "ALL"
                    }
                }
            ]
        }
```

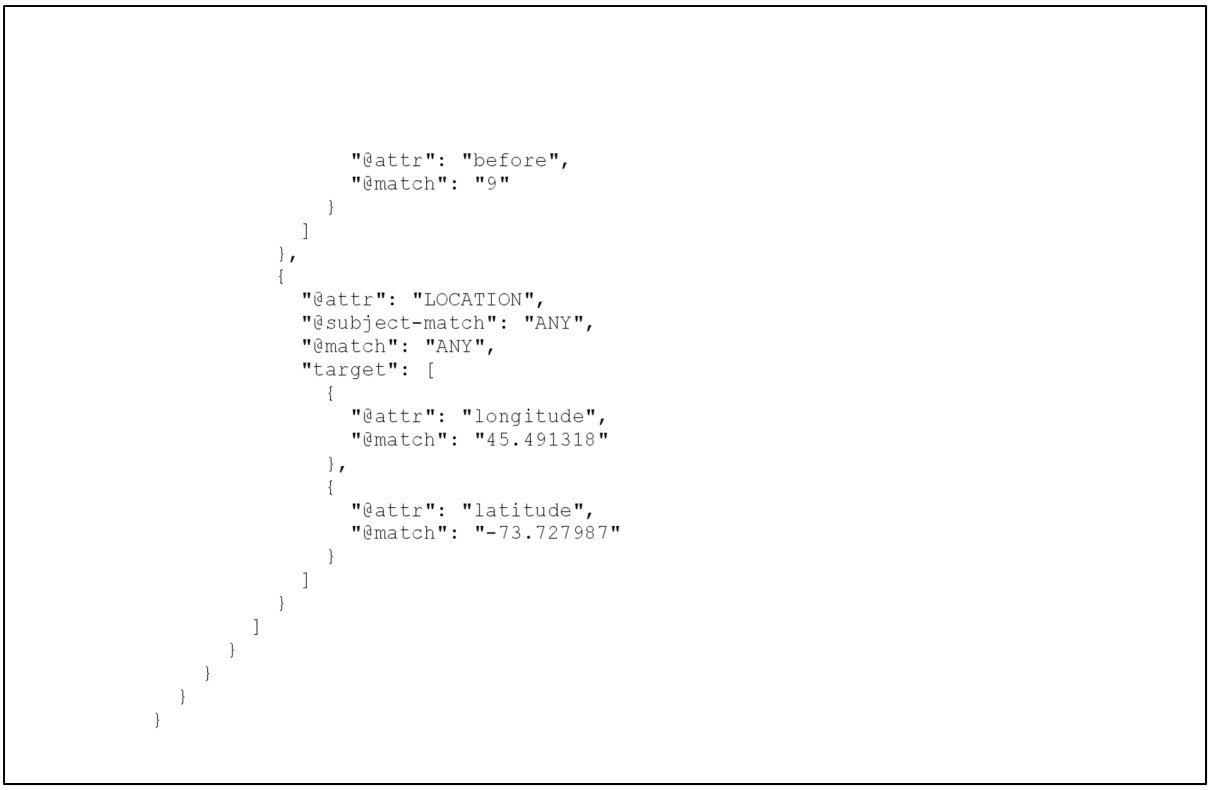

Figure II.3 La présentation de la troisième politique en JSON

```
{
    "@combine": "deny-overrides",
    "@id": "4",
    "@AUTHOR-KEY-CN": "Mahdi",
    "@AUTHOR-KEY-FINGERPRINT": "Mahdi",
    "target": {
        "subject": {
            "subject-match": {
                "@attr": "id_BNC",
                "@match": "com.apps.BNCbanque"
            }
        }
    },
    "rule": {
        "@effect": "deny",
        "condition": {
            "ressources": {
                "ressource": [
                    {
                        "@attr": "API",
                        "@subject-match": "id_BNC",
                        "@match": "android.permission.CAMERA"
                    },
                    {
                        "@attr": "API",
                        "@subject-match": "id_BNC",
                        "@match": "android.permission.INTERNET"
                    }
                ]
            },
            "contexts": {
                "context": {
                    "context-match": {
                        "@attr": "LOCATION",
                        "@subject-match": "ANY",
                        "@match": "ANY",
                        "target": [
                            {
                                "@attr": "longitude",
                                "@match": "45.491318"
                            },
                            {
                                "@attr": "latitude",
                                "@match": "-73.727987"
                            }
                        ]
                    }
                }
            }
```


Figure II.4 La présentation de la quatrième politique en JSON

```
{
  "@combine": "deny-overrides",
  "@id": "5",
  "@AUTHOR-KEY-CN": "Mahdi",
  "@AUTHOR-KEY-FINGERPRINT": "Mahdi",
  "target": {
    "subject": [
      {
        "@attr": "id_Facebook",
        "@match": "com.apps.Facebook"
      },
      {
        "@attr": "id_Instagram",
        "@match": "com.apps.Instagram"
      },
      {
        "@attr": "id_Gmail",
        "@match": "com.apps.Gmail"
      },
      {
        "@attr": "id_Message",
        "@match": "com.apps.Message"
      },
      {
        "@attr": "id_RecordAudio",
        "@match": "com.apps.recordAudio"
      }
    ]
  },
  "rule": {
    "@effect": "deny",
    "condition": {
      "ressources": {
        "ressource": [
          {
            "@attr": "API",
            "@subject-match": "id_Facebook",
            "@match": "android.permission.CAMERA"
          },
          {
            "@attr": "API",
            "@subject-match": "id_Facebook",
            "@match": "android.permission.INTERNET"
          },
          {
            "@attr": "API",
            "@subject-match": "id_Facebook",
            "@match": "android.permission.ACCESS_FINE_LOCATION"
          },
          {
            "@attr": "API",
            "@subject-match": "id_Instagram",
            "@match": "android.permission.CAMERA"
          },
```

```
      {
        "@attr": "API",
        "@subject-match": "id_Instagram",
        "@match": "android.permission.INTERNET"
      },
      {
        "@attr": "API",
        "@subject-match": "id_Instagram",
        "@match": "android.permission.ACCESS_FINE_LOCATION"
      },
      {
        "@attr": "API",
        "@subject-match": "id_Gmail",
        "@match": "android.permission.CAMERA"
      },
      {
        "@attr": "API",
        "@subject-match": "id_Gmail",
        "@match": "android.permission.INTERNET"
      },
      {
        "@attr": "API",
        "@subject-match": "id_Gmail",
        "@match": "android.permission.ACCESS_FINE_LOCATION"
      },
      {
        "@attr": "API",
        "@subject-match": "id_Message",
        "@match": "android.permission.RECEIVE_SMS"
      },
      {
        "@attr": "API",
        "@subject-match": "id_Message",
        "@match": "android.permission.RECEIVE_MMS"
      },
      {
        "@attr": "API",
        "@subject-match": "id_RecordAudio",
        "@match": "android.permission.RECORD_AUDIO"
      }
    ]
  },
  "contexts": {
    "context": [
      {
        "@attr": "TIME",
        "@subject-match": "ANY",
        "@match": "ANY",
        "target": [
          {
            "@attr": "after",
            "@match": "8"
          },
          {
```



```
                    "@attr": "before",
                    "@match": "9"
                  }
                ]
              },
              {
                "@attr": "LOCATION",
                "@subject-match": "ANY",
                "@match": "ANY",
                "target": [
                  {
                    "@attr": "longitude",
                    "@match": "45.491318"
                  },
                  {
                    "@attr": "latitude",
                    "@match": "-73.727987"
                  }
                ]
              }
            ]
          }
        }
      }
    }
  }
```

Figure II.5 La présentation de la cinquième politique en JSON

BIBLIOGRAPHIE

Align Minds. (2015, October 24). *Broadcast Announcements in Android*. Récupéré sur Align Minds: http://www.alignminds.com/blog/broadcast-announcements-android/

Alkurdi, M. Z. (2014). *Malware detection for android applications using simhash algorithm.* Gaza, Palastine : The Islamic University of Gaza.

Apktool. (2016, Oct 18). *A tool for reverse engineering Android apk files*. Récupéré sur ibotpeaches: https://ibotpeaches.github.io/Apktool

Arena, V., Catania, V., & Torre, G. L. (2013). SecureDroid: An Android security framework extension for context-aware policy enforcement. *Privacy and Security in Mobile Systems (PRISMS), 2013 International Conference on*, 1-8.

Auffray, C. (2017, Décembre 01). *Chiffres clés : les OS pour smartphones*. Récupéré sur zdnet: http://www.zdnet.fr/actualites/chiffres-cles-les-os-pour-smartphones-39790245.htm

Auffray, C. (2017, Novembre 30). *Chiffres clés : les ventes de mobiles et de smartphones*. Récupéré sur zdnet: http://www.zdnet.fr/actualites/chiffres-cles-les-ventes-de-mobiles-et-de-smartphones-39789928.htm

Belkaab, O. (2017, mars 14). *40 % des applications Android risquent de mettre en danger vos données*. Récupéré sur frandroid: http://www.frandroid.com/android/applications/securite-applications/417730_40-des-applications-android-risquent-de-mettre-en-danger-vos-donnees

Boudar, O. (2016). *Solution centralisée de contrôle d'accès basée réécriture d'applications pour la plateforme Android.* Montreal, Canada: école de technologie supérieure université du québec.

Chilowicz, M. (2012, 09 30). *IPC sous Android*. Récupéré sur igm univmlv: http://igm.univmlv.fr/~chilowi/teaching/subjects/java/android/ipc/index.html.

Cooper, V. N., Shahriar, H., & Haddad, H. M. (2014). A Survey of Android Malware Characterisitics and Mitigation Techniques. *Information Technology: New Generations (ITNG), 2014 11th International Conference on* (pp. 327-332). Las Vegas, NV, USA: IEEE.

Crussell, J., Gibler, C., & Chen, H. (2012). Attack of the Clones: Detecting Cloned Applications on Android Markets. *European Symposium on Research in Computer (ESORICS)*, 37-54.

Davis, B., & Chen, H. (2013). Retroskeleton: Retrofitting android apps. *Proceeding of the 11th annual international conference on Mobile systems* (pp. 181-192). New York, NY, USA: ACM.

Davis, B., Sanders, B., Khodaverdian, A., & Chen, H. (2012). I-arm-droid: A rewriting framework for in-app reference monitors for android applications. Mobile Security Technologies, 2012. *Mobile Security Technologies*.

El-Harake, K., Falcone, Y., Jerad, W., Langet, M., & Mamlouk, M. (2014). Blocking Advertisements on Android Devices using Monitoring Techniques. *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, 239-253.

El-Serngawy, M., & Talhi, C. (2015). CaptureMe: Attacking the User Credential in Mobile Banking Applications. *2015 IEEE Trustcom/BigDataSE/ISPA*. Helsinki, Finland: IEEE.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. N. (2010). TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 393-407.

Enck, W., Octeau, D., McDaniel, P., & Chaudhuri, S. (2011). A Study of Android Application Security. *SEC'11 Proceedings of the 20th USENIX conference on Security* (pp. 21-21). San Francisco, CA: USENIX.

Espiau, F. (2017, Mars 2). *Créez des applications pour Android*. Récupéré sur Open Classrooms: https://openclassrooms.com/courses/creez-des-applications-pour-android/la-communication-entre-composants

Espiau, F. (2017, janvier 12). *L'architecture d'Android*. Récupéré sur openclassrooms: https://openclassrooms.com/courses/creez-des-applications-pour-android/l-architecture-d-android

Felt, A. P., Finifter, M., Chin, E., Hanna, S., & Wagner, D. (2011). A survey of mobile malware in the wild. *SPSM '11 Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 3-14.

Feth, D., & Pretschner, A. (2012). Flexible data-driven security for android In Software Security and Reliability (SERE). *IEEE Sixth International Conference*, 41-50.

Gandhi, V. (2014, July 15). *And The Mice Will "Play"...: App Stores And The Illusion Of Control Part II*. Récupéré sur zscaler: https://www.zscaler.com/blogs/research/and-mice-will-play-app-stores-and-illusion-control-part-ii

Gibler, C., Crussell, J., Erickson, J., & Chen, H. (June 13-15, 2012). AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. *5th International Conference, TRUST 2012,* Vienna, Austria.

Google. (2017, 12 01). *Requesting Permissions*. Récupéré sur Developer Android: https://developer.android.com/guide/topics/permissions/requesting.html#perm-groups

Google. (2016, 01 31). *Android 6.0 Changes*. Récupéré sur Developer Android: https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html

Google. (2016, March). *Application Fundamentals*. Récupéré sur developer: http://developer.android.com/guide/components/fundamentals.html

Gruver, B. (2017, 12 31). *Smali*. Récupéré sur github: https://github.com/JesusFreke/smali

Gupta, L. (2015, 01 30). *Spring AOP – AspectJ Annotation Config Example*. Récupéré sur How to do in java: https://howtodoinjava.com/spring/spring-aop/spring-aop-aspectj-example-tutorial-using-annotation-config/

Hornyack, P., Han, S., Jung, J., Schechter, S., & Wetherall, D. (2011). These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. *Proceedings of the 18th ACM conference* (pp. 639-652). Chicago, Illinois, USA: Computer and communications security.

Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., & Millstein, T. (2012). Dr. android and mr. hide: fine-grained permissions in android applications. *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 3-14.

Jiang, X., & Zhou, Y. (2013). *Android Malware.* Chicago: Springer.

Karmakar, S. (2012). *A Policy Enforcement Framework for Android (.* Bombay Mumbai: Indian Institute of Technology.

Ketfi, C. (2017, janvier 03). *Android a été le logiciel le plus vulnérable aux attaques en 2016*. Récupéré sur Frandroid: http://www.frandroid.com/android/402067_android-a-ete-le-logiciel-le-plus-vulnerable-aux-attaques-en-2016

Kim, J., Yoon, Y., Yi, K., & Shin, J. (2012). SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications. *Proceedings of the Mobile Security Technologies Workshop (MoST'12)*.

Lu, L., Li, Z., Wu, Z., Lee, W., & Jiang, G. (2012). CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. *Proceedings of the 2012 ACM conference* (pp. 229-240). Raleigh, North Carolina, USA: Computer and communications security.

Maler, E. (2014, decembre 28). *UMA FAQ*. Récupéré sur kantarainitiative: https://kantarainitiative.org/confluence/display/uma/UMA+FAQ

Mann, C., & Darmstadt, G. (2012). A Framework for Static Detection of Privacy Leaks in Android Applications. *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)* (pp. 1457-1462). Trento, Italy: ACM Symposium on Applied Computing.

Mann, C., & Starostin, A. (2012). A Framework for Static Detection of Privacy Leaks in Android Applications. *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*, 1457-1462.

Montemagno, J. (2015, 09 21). *Requesting Runtime Permissions in Android Marshmallow*. Récupéré sur Xamarin: https://blog.xamarin.com/requesting-runtime-permissions-in-android-marshmallow/

Nauman, M., Khan, S., & Zhang, X. (2010). Apex: extending Android permission model and enforcement with user-defined runtime constraints. *ASIACCS '10 Proceedings of the 5th ACM Symposium on Information* (pp. 328-332). Beijing, China: Computer and Communications Security.

Parducci, B., Lockhart, H., & Levinson, R. (2017, July 17). *OASIS eXtensible Access Control Markup Language (XACML) TC*. Récupéré sur oasis-open: https://www.oasis-open.org/committees/xacml

Pearce, P., Felt, A. P., Nunez, G., & Wagner, D. (2012). AdDroid: privilege separation for applications and advertisers in Android. *Proceedings of the 7th ACM Symposium on Information* (pp. 71-72). Seoul, Korea: Computer and Communications Security.

Pétrod, J.-L. (2017, août 23). *Android, l'OS le plus vulnérable en 2016 : 3 fois plus de failles qu'iOS !* Récupéré sur Phonandroid: http://www.phonandroid.com/android-os-plus-vulnerable-2016-3-plus-failles-ios.html

Picard, T. (2012, avril 20). *Des Virus débarquent sur les mobiles sous Androïd*. Récupéré sur Nrblog: http://www.nrblog.fr/pepito/2012/04/20/des-virus-debarquent-sur-les-mobiles-sous-android/

Rasthofer, S., Arzt, S., & Lovat, E. (2014). DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on* (pp. 40-49). Fribourg, Switzerland: IEEE.

Rasthofer, S., Arzt, S., Lovat, E., & l. (2014). DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, 40-49.

Rasthofer, S., Asrar, I., Huber, S., & Bodden, E. (2015). *An investigation of the Android/BadAccents malware which exploits a new Android tapjacking attack.* Darmstadt: Fraunhofer.

Saini, G. (2015). *Permission based android malware detection using supervised learning techniques.* Jalandhar: Department of Computer Science and Engineering.

Says, Y. (2017, November 12). *How to manage app permissions in Android Marshmallow?* Récupéré sur Gadget guide online: http://gadgetguideonline.com/android/android-marshmallow-guide/how-to-manage-app-permissions-in-android-marshmallow/

Schenk, C. (2007, July 2). *AOP with AspectJ*. Récupéré sur Christian Schenk: http://www.christianschenk.org/blog/aop-with-aspectj/

Schiefer, L. (2014, February 3). *Android, Best Practices, Security*. Récupéré sur hiqes: http://hiqes.com/android-security-part-1/

Scriptol. (2017, 12 31). *Dalvik, la machine virtuelle d'Android*. Récupéré sur scriptol: https://www.scriptol.fr/programmation/dalvik.php

Shekhar, S., Dietz, M., & Wallach, D. S. (2012). AdSplit: Separating smartphone advertising from applications. *Computer Science*, 1202-1217.

Styp-Rekowsky, P. v., Gerling, S., Backes, M., & Hammer, C. (2013). Idea: callee-site rewriting of sealed system libraries. *Engineering Secure Software and Systems* (pp. 33-41). Berlin Heidelberg: Springer.

The AspectJ Team. (2003). *Introduction to AspectJ*. Récupéré sur eclipse: https://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

Tutorial-all. (2017). *Application components*. Récupéré sur Tutorial-all: http://tutorial-all.org/11-android/126-4-application-components

Wu, C., Zhou, Y., Patel, K., Liang, Z., & Jiang, X. (2014). AirBag: Boosting Smartphone Resistance to Malware Infection. *Proceedings of the 21th Annual Network.* Distributed System Security Symposium (NDSS'14).

Xu, R., Saıdi, H., & Anderson, R. (2012). Aurasium: Practical Policy Enforcement for Android Applications. *USENIX Security Symposium* (pp. 539-552). Bellevue, WA: USENIX.

Yang, W., Li, J., Zhang, Y., Li, Y., Shu, J., & Gu, D. (2014). APKLancet: tumor payload diagnosis and purification for android applications. *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 483-494.

Yang, Z., Yang, M., Zhang, Y., Gu, G., Gu, G., & Wang, X. S. (2013). AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. *Proceedings*

of the 2013 ACM SIGSAC conference on Computer & communications security(CCS '13), 1043-1054 .

Zhang, M., & Yin, H. (2014). Efficient, context-aware privacy leakage confinement for android applications without firmware modding. *Proceedings of the 9th ACM symposium on Information, computer and communications security (ASIA CCS '14)* , 259-270.

Zhang, X., Ahlawat, A., & Du, W. (2013). AFrame: isolating advertisements from mobile applications in Android. *Proceedings of the 29th Annual Computer Security Applications Conference* (pp. 9-18). New Orleans, Louisiana, USA: ACM.

Zhou, W., Zhang, X., & Jiang, X. (2013). AppInk: watermarking android apps for repackaging deterrence. *Proceedings of the 8th ACM SIGSAC symposium on Information,* (pp. Hangzhou, China). Hangzhou, China: computer and communications security.

Zhou, W., Zhou, Y., Jiang, X., & Ning, P. (2012). Detecting repackaged smartphone applications in third-party android marketplaces. In(pp.). . *Proceedings of the second ACM conference* (pp. 317-326). San Antonio, Texas, USA: Data and Application Security and Privacy.

Zhou, Y., & Jiang, X. (2012.). Dissecting Android Malware: Characterization and Evolution. *IEEE Symposium on Security and Privacy*, 95–109.

Zhou, Y., Zhang, X., Jiang, X., & Freeh, V. W. (2011). Taming information-stealing smartphone applications (on android). *Springer Berlin Heidelberg.*, 93-107. Récupéré sur In Trust and Trustworthy Computing.

Zhou, Y., Zhang, X., Jiang, X., & Freeh, V. W. (2011). Taming Information-Stealing Smartphone Applications (on Android). *International Conference on Trust and Trustworthy Computing* (pp. 93-107). Berlin, Heidelberg: Springer.