Contrôle d'accès dynamique et architecture de sécurité pour la protection des applications sous Android

Guillaume Averlant

▶ To cite this version:

Guillaume Averlant. Contrôle d'accès dynamique et architecture de sécurité pour la protection des applications sous Android. Cryptographie et sécurité [cs.CR]. INSA de Toulouse, 2019. Français. NNT : 2019ISAT0026 . tel-02440787v2

HAL Id: tel-02440787
https://theses.hal.science/tel-02440787v2

Submitted on 4 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :
l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le *02/10/2019* **par :**
Guillaume Averlant

Contrôle d'accès dynamique et architecture de sécurité pour la protection des applications sous Android

---

JURY

Maryline Laurent	Professeur des universités	Rapporteur
Jean-Louis Lanet	Professeur des universités	Rapporteur
Jean-François Lalande	Maître de conférences	Examinateur
Louis Rilling	Ingénieur de recherche	Examinateur
Mohamed Kaâniche	Directeur de recherche	Examinateur
Vincent Nicomette	Professeur des universités	Directeur de thèse
Éric Alata	Maître de conférences	Directeur de thèse

École doctorale et spécialité :
EDMITT : Informatique et Télécommunications
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes
Directeur(s) de Thèse :
Vincent Nicomette et *Éric Alata*
Rapporteurs :
Maryline Laurent et *Jean-Louis Lanet*

Remerciements

Les travaux présentés dans ce manuscrit ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS). Je remercie Jean Arlat et Liviu Nicu qui ont assuré la direction du LAAS depuis mon arrivée.

Je remercie aussi Mohamed Kaâniche, responsable de l'équipe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF), équipe qui m'a chaleureusement accueilli pour la réalisation de mes travaux de recherche. En plus de diriger cette équipe, Mohamed Kaâniche est très impliqué dans tous les travaux de recherche menées en son sein, et m'a ainsi fourni à de nombreuses reprises une aide précieuse.

Je tiens ensuite à exprimer mon entière gratitude envers mes deux directeurs de thèse, Vincent Nicomette dont l'énergie et la compassion m'a permis de resté motivé même dans les temps difficiles, et Éric Alata qui a toujours su apporter un point de vue pertinent sur mes travaux de recherche. Ils m'ont tout deux grandement épaulé tout au long de cette thèse, et ce malgré leur charges de travail importantes, c'est aussi pour cela que je tiens à les remercier.

J'adresse également mes sincères remerciements aux membres du jury qui ont accepté de juger mon travail. Je leur suis très reconnaissant pour l'intérêt qu'ils ont porté à mes travaux :

- Maryline Laurent, Professeur des universités à Télécom SudParis.
- Jean-Louis Lanet, Professeur des universités à l'INRIA-Rennes.
- Jean-François Lalande, Maître de conférences à Supelec Rennes.
- Louis Rilling, Ingénieur de recherche à la DGA-MI.
- Mohamed Kaâniche, Directeur de recherche au LAAS-CNRS.
- Vincent Nicomette, Professeur des universités à l'INSA de Toulouse.
- Éric Alata, Maître de conférences à l'INSA de Toulouse.

Je remercie tout particulièrement Maryline Laurent et Jean-Louis Lanet qui ont accepté de rapporter ma thèse, et dont les différents retours ont été très enrichissants.

Le bon déroulement de cette thèse est également à mettre au crédit de l'équipe TSF dans son ensemble qui a constitué un cadre de recherche bienveillant et enrichissant. Je remercie tout particulièrement mes collègues doctorants et amis qui ont grandement participé à cette agréable ambiance de travail : Matthieu, Jonathan, Aliénor, Clément, Rémi, Daniel, Malcolm, Christophe, Bilel, Jean, PF, Romain, Yuxiao, Luca, Raul, Mohamed, Syrius, Florent et également leurs prédécesseurs qui se reconnaîtront.

Mes derniers remerciements sont adressés à mes parents et mes deux frères qui m'ont toujours poussé à aller de l'avant, à être curieux et à m'épanouir. Leur soutien indéfectible a été primordial à la réalisation de ce grand voyage, dans lequel il aurait été facile de se perdre autrement.

The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka !', but 'That's funny ...'

Isaac Asimov

Table des matières

Introduction **1**

1 Contexte général, mobilité et écosystème Android **5**
1.1 Smartphone et mobilité 5
1.1.1 Matériel embarqué 6
1.2 L'écosystème Android 7
1.2.1 Panorama des couches logicielles 8
1.2.2 Modèle de sécurité existant 12
1.3 Motivations des attaquants et problématique 16
1.4 Conclusion 17

2 État de l'art **19**
2.1 Terminologie de la sécurité informatique 20
2.1.1 Sûreté de fonctionnement 20
2.1.2 Sécurité-immunité 21
2.1.3 Politiques de sécurité 23
2.2 Mécanismes de contrôle d'accès pour Android 27
2.2.1 Mécanismes de niveau système 27
2.2.2 Mécanismes de niveau OS 29
2.2.3 Mécanismes de niveau framework 32
2.2.4 Mécanismes de niveau applicatif 34
2.3 Politiques d'autorisation pour Android 35
2.3.1 Considérations relatives à l'environnement Android 35
2.3.2 Politiques d'autorisation de la littérature 37
2.4 Conclusion et démarche scientifique 40

3 Politique de sécurité pour Android sensible au contexte **43**
3.1 Présentation, objectifs et modèle d'attaque 44
3.1.1 Objectifs de la solution 44
3.1.2 Objets considérés par la politique 44
3.1.3 Types de règles de contrôle d'accès 46
3.1.4 Contexte nécessaire à l'application des règles 47
3.1.5 Modèle et scénarios d'attaque 47
3.2 Format, exemples et distribution 48
3.2.1 Format des règles de sécurité : le *Secure Manifest* 48
3.2.2 Exemples d'utilisation 50
3.2.3 Distribution des *Secure Manifest* 52
3.3 Les conflits et leur gestion 53
3.3.1 Définition et objectifs de la gestion des conflits 53
3.3.2 Aspects pratiques 54

3.3.3 Aspects ergonomiques . . . 55
3.3.4 Améliorations possibles . . . 56
3.4 Encourager l'utilisateur à paramétrer la politique . . . 57
3.5 Conclusion . . . 59

4 Architecture de sécurité multi-niveau 61
4.1 Besoins et contraintes techniques de notre politique . . . 61
4.1.1 Contrôle d'accès et contexte . . . 61
4.1.2 Protections contre le modèle d'attaque envisagé . . . 62
4.1.3 Un moniteur de référence multi-niveau . . . 63
4.2 Présentation de l'architecture . . . 64
4.2.1 Aperçu des composants et caractéristiques . . . 64
4.2.2 Détails des composants . . . 65
4.3 Mise en œuvre de la politique . . . 67
4.3.1 Récupération des *Secure Manifest* . . . 67
4.3.2 Génération, activation et mise en application des règles . . . 68
4.3.3 Gestion des conflits et expérience utilisateur . . . 74
4.4 Contrôle de l'intégrité de notre solution . . . 76
4.4.1 Périmètre de la protection . . . 76
4.4.2 Protection contre les attaques logicielles . . . 77
4.4.3 Protection contre les attaques matérielles . . . 79
4.4.4 Protection au démarrage et chaîne de confiance . . . 80
4.5 Conclusion . . . 81

5 Prototype et performances 83
5.1 Plateformes d'expérimentation . . . 84
5.1.1 La plateforme de développement *96Boards hikey* . . . 84
5.1.2 L'émulateur Android . . . 85
5.2 Implémentation . . . 85
5.2.1 AOSP, base logicielle d'Android . . . 86
5.2.2 Implémentation du *PDA* . . . 87
5.2.3 Implémentation du *PHS* . . . 90
5.2.4 Implémentation du *SCIH* . . . 93
5.2.5 Implémentation de l'*IH* . . . 96
5.2.6 Limitations du prototype . . . 100
5.3 Mesures de performances . . . 101
5.3.1 Performances des actions atomiques de notre prototype . . . 102
5.3.2 Microbenchmarks . . . 105
5.3.3 Macrobenchmarks . . . 106
5.4 Validation par scénario . . . 107
5.4.1 Scénario n°1 . . . 108
5.4.2 Scénario n°2 . . . 110
5.4.3 Mise en relation avec les attaques récentes . . . 113
5.5 Conclusion . . . 113

Conclusion **115**

A Exemples de code **119**

A.1 Modification des droits en écriture d'une page de la MMU pour l'architecture *ARMv8* 119
A.2 Introspection de la mémoire du driver du binder 120
A.3 Makefile de l'*IH* 122
A.4 Point d'entrée de l'*IH* 123

Bibliographie **125**

Liste des figures

1.1 Niveaux de privilèges de l'architecture ARMv8-A 7
1.2 Composition du framework Android 9
1.3 Fonctionnement du Binder 14

3.1 Visualisation de l'utilisation des ressources par une application . . . 58

4.1 Aperçu de l'architecture 64
4.2 Récupération du Secure Manifest 68
4.3 Structures de données et activation des règles 69
4.4 Processus d'activation des règles (1/2) 70
4.5 Processus d'activation des règles (2/2) 71
4.6 Mise en application des règles aux *intents* 72
4.7 Mise en application des règles aux appels systèmes 73
4.8 Gestion des conflits 75
4.9 Virtualisation de la MMU par un hyperviseur 78

5.1 La plateforme de développement *96Boards hikey* 85
5.2 *PDA* : écran d'accueil et liste des *Secure Manifests* 88
5.3 *PDA* : configuration des *Secure Manifests* et gestion des conflits . . . 89
5.4 Diagramme de classe UML simplifié du *PHS* 90
5.5 Introspection réalisée par le SCIH sur les transactions binder 94
5.6 Diagramme de classe UML des structures internes du Binder 95
5.7 Processus de démarrage et insertion de l'IH 98
5.8 Coût de l'application des règles à l'interception d'*intents* 102
5.9 Coût de la vérification périodique de la fermeture des applications . 103
5.10 Scénario n°1 109
5.11 Scénario n°2 - Lancement des deux applications 111
5.12 Scénario n°2 - Mise en œuvre des règles du *Secure Manifest* 112

Liste des tableaux

2.1 Synthèse des politiques d'autorisation de l'état de l'art 40

3.1 Ressources considérées 45

5.1 Surcoût de l'interception des appels système 104
5.2 Surcoût pour l'exécution d'un *intent* 105
5.3 Surcoût pour l'exécution d'une transaction binder 106
5.4 Résultats des macrobenchmarks 107

Liste des abréviations

AAF	Android Application Framework, page 8
ABAC	Attribute-Based Access Control, page 25
ADB	Android Debug Bridge, page 84
AIDL	Android Interface Definition Language, page 14
AMS	Activity Manager Service, page 14
AOSP	Android Open Source Project, page 7
API	Application Programming Interface, page 8
ART	Android Runtime, page 9
BLE	Bluetooth Low Energy, page 17
BN	Binder Node, page 13
BYOD	Bring Your Own Device, page 1
CAAC	Context-Aware Access Control, page 25
CFI	Control-flow Integrity, page 78
DAC	Discretionary Access Control, page 24
Dex	Dalvik Executable, page 9
DMA	Direct Memory Access, page 79
DVM	Dalvik VM, page 34
EL	Exception Level, page 6
GID	Group ID, page 12
HAL	Hardware Abstraction Layer, page 11
HIDL	HAL Interface Definition Language, page 11
ICC	Inter Component Communication, page 13
IH	Integrity Hypervisor, page 64
IO-MMU	Input–Output Memory Management Unit, page 28
IPC	Inter-Process Communication, page 8
JNI	Java Native Interface, page 10
LMK	Low Memory Killer, page 8
LSM	Linux Security Module, page 30

LTS	Long Term Support, page 8
MAC	Mandatory Access Control, page 24
MMU	Memory Management Unit, page 28
NDK	Native Development Kit, page 9
NFC	Near-Field Communication, page 6
PCB	Printed Circuit Board, page 62
PDA	Policy Definition App, page 64
PHS	Policy Handler Service, page 64
RBAC	Role-Based Access Control, page 25
RPC	Remote Procedure Call, page 13
SCIH	System Call Interception Handler, page 64
SDK	Software Development Kit, page 9
SoC	System on Chip, page 6
TOCTOU	time of check to time of use, page 45
UART	Universal Asynchronous Receiver Transmitter, page 84
UEFI	Unified Extensible Firmware Interface, page 80
UID	User ID, page 12
VMI	Virtual Machine Introspection, page 29
VPN	Virtual Private Network, page 46
XACML	eXtensible Access Control Markup Language, page 26

Introduction

Étant poussée par les changements technologiques des microprocesseurs, cette dernière décennie a vu croître l'utilisation des smartphones au quotidien. Aujourd'hui, leur usage est ancré dans les habitudes de consommation du numérique de tout à chacun, et remplace de plus en plus souvent l'utilisation des ordinateurs « classiques », de manière partielle voire totale. Cette transformation des moyens de consommation du numérique s'est aussi accompagnée d'une évolution des contenus consommés. Le smartphone est donc désormais utilisé comme une plateforme privilégiée pour les interactions sociales, mais aussi avec les différents services numériques ayant fleuri depuis l'avènement du « cloud computing ». Du fait de ces multiples changements, les smartphones ont désormais une place prépondérante au quotidien et sont de plus en plus utilisés pour prendre en charge de multiples activités de la vie de tous les jours. De plus, cet état de fait est renforcé par la composante intrinsèque de connectivité des smartphones, assurée par de multiples périphériques de communications, qui leur permet d'interagir avec un environnement de plus en plus connecté. Outre son utilisation personnelle, l'usage du smartphone dans le milieu de l'entreprise n'est pas non plus en reste, notamment avec l'apparition du concept de *Bring Your Own Device* (BYOD) qui permet à un employé d'utiliser son appareil personnel pour les besoins professionnels.

Ainsi, l'ensemble des fonctionnalités offertes par un smartphone oblige celuici à devoir gérer une grande quantité de données personnelles ou professionnelles sensibles. On peut considérer comme exemple les données relatives aux applications des réseaux sociaux, aux emails, ainsi qu'aux applications bancaires. Cette position particulière fait du smartphone une cible très attractive pour un grand nombre d'attaquants qui souhaitent mettre en péril la sécurité des données y étant stockées. Cependant ils peuvent aussi se servir du smartphone pour cibler d'autres objets du quotidien (ordinateurs, objets connectés...) via les moyens de communication offerts par celui-ci.

Dans cette thèse, nous nous intéressons plus particulièrement à l'environnement Android, qui est le système d'exploitation mobile le plus utilisé à l'heure actuelle avec 85% de part du marché mondial en 2018 [IDC 2018]. De multiples mesures de sécurité ont déjà été implémentées dans ce système d'exploitation. Cependant, sont-elles suffisantes pour permettre à l'utilisateur du smartphone d'avoir un contrôle adéquat concernant l'accès aux différentes données du téléphone, ainsi qu'aux ressources matérielles de connectivité, par les applications installées ? C'est à cette question que nous souhaitons répondre dans cette thèse, en offrant à l'utilisateur la possibilité de :

- Mieux contrôler les données manipulées par les smartphones Android.
- Bénéficier d'une gestion avancée des contrôles d'accès à ces données.

Pour atteindre cet objectif, cette thèse propose trois contributions essentielles. La première contribution consiste à étendre le modèle de contrôle d'accès utilisé ha-

bituellement dans les systèmes Android de façon à permettre à ses utilisateurs une gestion plus fine et plus dynamique des privilèges des applications qui s'exécutent sur leur smartphone. Ce modèle de contrôle d'accès permet notamment d'autoriser ou d'interdire l'exécution de certaines applications, ou l'accès à certaines ressources en fonction du contexte courant d'exécution du smartphone. Autrement dit, il nous semble fondamental aujourd'hui, par exemple, de pouvoir autoriser ou interdire l'exécution d'une application lorsqu'une autre application, considérée plus critique est en cours d'utilisation, ou de pouvoir interdire à une application l'accès à une ressource particulière lorsqu'une application plus critique accède au même moment à la même ressource. Ceci nécessite de pouvoir formaliser les règles d'accès correspondantes et de pouvoir les associer aux différentes applications. Nous définissons ainsi dans cette thèse le concept de « Secure Manifest » qui vient s'ajouter au *manifeste* habituel des applications Android, et dans lequel, de nouvelles règles de sécurité peuvent être définies.

La définition de cette politique de contrôle d'accès est fondamentale mais elle ne peut être efficace que si des mécanismes de contrôle d'accès sont bel et bien implantés dans les couches logicielles du smartphone. La seconde contribution de cette thèse consiste donc en la proposition d'une architecture de sécurité permettant de mettre en œuvre ces contrôles d'accès sur un smartphone Android. L'approche adoptée dans cette thèse est d'insérer ces mécanismes de contrôle d'accès à la fois dans le framework Android, le noyau Linux et dans un hyperviseur *bare-metal* spécifiquement développé, constitutifs d'une architecture dite « multi-niveau ».

Enfin, nous avons montré la pertinence de cette approche par le développement d'un prototype d'une partie de cette architecture, associée à des tests de performance. C'est la troisième contribution de cette thèse.

Le manuscrit est organisé comme suit. Le premier chapitre présente le contexte général de nos travaux, puis expose en détail l'écosystème Android puisque nos travaux ont été réalisés dans ce cadre. Ce système est assez complexe et une explication de son architecture est indispensable à la bonne compréhension des travaux de cette thèse. Nous dressons donc un panorama des différentes couches logicielles qui le composent, puis nous présentons le modèle de sécurité actuel de ce système.

Le second chapitre est consacré à la discussion concernant les travaux connexes de l'état de l'art. Après une introduction à la terminologie relative à la sécurité informatique (et notamment aux définitions se rapportant aux politiques de sécurité), nous présentons les mécanismes de contrôle d'accès existant aujourd'hui dans le système Android, en les classant en fonction des couches logicielles dans lesquelles ils sont implantés. Nous présentons également les politiques d'autorisation qui sont à notre connaissance utilisées dans ce système.

Le troisième chapitre est consacré à la première contribution de cette thèse et présente une politique de sécurité pour Android sensible au contexte. Après avoir précisé les objectifs de cette politique, nous révélons les objets et les règles qui la composent, la structure des « Secure Manifest », ainsi que le modèle d'attaque sous-jacent. Nous présentons également quelques exemples d'utilisation de ces règles

au travers de quelques cas d'études simples, mais pertinents pour illustrer l'utilité de cette nouvelle politique. Pour chacun de ces cas d'étude, le « Secure Manifest » associé est détaillé. Nous présentons ensuite la résolution de conflits qu'impose l'application de cette politique de contrôle d'accès, et nous terminons par l'exposition d'un outil encourageant l'utilisateur à paramétrer la politique en l'informant sur l'historique de consommation de ressources de ses applications installées.

Le quatrième chapitre est consacré à la seconde contribution de nos travaux et présente en détail l'architecture de sécurité multi-niveau que nous proposons pour le système Android afin d'y implanter de façon efficace les mécanismes de contrôle d'accès associés à cette politique. L'architecture globale de notre solution est introduite, avec l'ensemble des éléments logiciels qui la composent. Nous présentons ensuite comment notre politique est mise en œuvre grâce à cette architecture, par la récupération des « Secure Manifest » associés aux applications, par la mise en place dans le système des règles de contrôle d'accès à partir du contenu de ces « Secure Manifest » (à la fois dans le framework Android et dans le noyau Linux), par l'application de ces règles en fonction des opérations qui s'exécutent dans le système, et par la gestion des conflits. Le chapitre se termine par la présentation d'un hyperviseur de sécurité *bare-metal*, dont le but est de vérifier l'intégrité des mécanismes de contrôle d'accès implantés dans les couches logicielles supérieures.

Le cinquième chapitre de ce manuscrit présente un prototype, constituant la troisième contribution de cette thèse, implémentant certains mécanismes de notre architecture de sécurité exposé dans le quatrième chapitre. Les différents composants matériels et logiciels utilisés pour ce prototype sont présentés ainsi que les détails d'implantation des mécanismes de contrôle d'accès des différents couches logicielles. Les mécanismes de contrôle utilisés par l'hyperviseur de sécurité sont également exposés en détail. Nous terminons ce chapitre par quelques tests fonctionnels et quelques tests de performance. Ils ont pour but de démontrer la pertinence de cette approche. Ils révèlent que notre architecture de sécurité dégrade très peu les performances du système et permet donc de réaliser des contrôles efficaces sans pour autant constituer un goulot d'étranglement dans le système. Les tests fonctionnels illustrent, au travers d'exemples simples, comment notre solution peut bloquer plusieurs attaques utilisées aujourd'hui dans le monde des smartphones Android.

Enfin, le dernier chapitre de cette thèse vient résumer les contributions de nos travaux et présente quelques perspectives pour des travaux futurs.

Chapitre 1

Contexte général, mobilité et écosystème Android

Sommaire

1.1 Smartphone et mobilité **5**
1.1.1 Matériel embarqué 6
1.2 L'écosystème Android **7**
1.2.1 Panorama des couches logicielles 8
1.2.2 Modèle de sécurité existant 12
1.3 Motivations des attaquants et problématique **16**
1.4 Conclusion **17**

Ce premier chapitre est dédié à la présentation du contexte relatif au smartphone et du système d'exploitation Android que nous avons étudié dans cette thèse. Une première partie explicite les notions de smartphone et mobilité, tant du point de vue de leur écosystème associé que du matériel embarqué dans les smartphones actuels. Ensuite, une deuxième partie présente les détails du système d'exploitation Android requis pour la compréhension de cette thèse. Enfin une dernière partie offre un aperçu du modèle d'attaque et des problématiques considérées dans cette thèse.

1.1 Smartphone et mobilité

Le terme *smartphone* désigne un téléphone mobile disposant d'un nombre important de fonctionnalités que l'on retrouve habituellement sur un ordinateur. Ainsi, en plus d'être capable d'émettre et de recevoir des appels ou des messages textuels, il permet à son utilisateur d'exécuter un certain nombre de programmes sous la forme d'applications. Son interface utilisateur est composée d'un écran tactile accompagné d'un ensemble de boutons capacitifs ou physiques. Sa conception à mi-chemin entre le téléphone mobile et l'ordinateur lui permet en outre de posséder un format adapté à la mobilité, tout en étant autonome sur une ou plusieurs journée(s) grâce à sa batterie intégrée. De plus, un smartphone est équipé d'un grand nombre de capteurs et de périphériques de communication lui permettant d'échanger avec un environnement de plus en plus connecté. Enfin, son format réduit ainsi que les contraintes énergétiques nécessaires à sa mobilité lui imposent des restrictions concernant la taille et la consommation des composants matériels qu'il embarque.

1.1.1 Matériel embarqué

La très grande majorité des smartphones s'appuient sur une base matérielle ayant pour fondation un processeur d'architecture ARM. Plus précisément, le processeur prend place au sein d'un *System on Chip* (SoC) qui regroupe les autres unités principales du système ainsi que les contrôleurs de bus permettant d'accéder aux périphériques du smartphone. Le choix d'utiliser ce type de processeurs s'appuie sur leur meilleure efficacité énergétique en comparaison avec les autres architectures existantes. La dernière itération de l'architecture ARM en date pour les profils applicatifs [1], nommée *ARMv8-A*, possède les caractéristiques suivantes :

- Première architecture ARM 64 bits.
- Jeu d'instructions de taille fixe (32 bits).
- Mode de compatibilité avec l'architecture 32 bits précédente.
- 31 registres généraux de 64 bits.
- 32 registres de 128 bits utilisés pour les calculs SIMD.
- 4 niveaux de privilèges permettant la prise en charge des extensions d'aide à la virtualisation, ainsi que les extensions de sécurité.

Plus précisément, la figure 1.1 présente ces quatre niveaux de privilèges, appelés *Exception Level* (EL) dans la nomenclature proposée par ARM. Les niveaux EL0 et EL1 sont destinés, de manière similaire aux ring 3 et ring 0 de l'architecture Intel64, respectivement aux applications et aux systèmes d'exploitation. EL2 est quant à lui réservé à l'hébergement d'hyperviseurs dans le cadre des extensions de virtualisation de l'architecture. Enfin, le niveau EL3 appartient aux extensions de sécurité, aussi appelées *TrustZone*. Ces extensions ont pour but de définir deux *modes de sécurité* hermétiques, au sein du processeur ainsi que pour le reste du système, appelées *normal world* et *secure world*. De nombreux mécanismes d'isolation sont ensuite mis en place pour que les ressources appartenant au *secure world* soient rendues inaccessibles depuis le *normal world*. De plus, le niveau de privilège EL3 est le seul niveau de privilège capable d'effectuer la transition du *normal world* au *secure world* et inversement au sein du processeur.

Plus de détails sur l'organisation interne des systèmes à base de processeurs ARMv8-A, ainsi que des précisions supplémentaires sur cette architecture sont fournies dans l'article [Averlant 2017a].

Outre le processeur et les autres composants intégrés au SoC tels que le GPU, le reste du matériel est essentiellement composé de périphériques d'interface utilisateur, de périphériques de communication et de multiples capteurs. Parmi les périphériques de communication présents sur un smartphone, on retrouve généralement des périphériques permettant d'accéder aux réseaux de téléphonie mobile et Wi-Fi, ainsi qu'aux technologies Bluetooth et *Near-Field Communication* (NFC). Ces nombreux moyens de communication représentent des vecteurs d'attaque supplémentaires à considérer dans le cadre de notre modèle de sécurité. Concernant les capteurs embarqués, nous pouvons les répartir en quatre catégories :

1. Profil architectural le plus adapté pour les smartphones

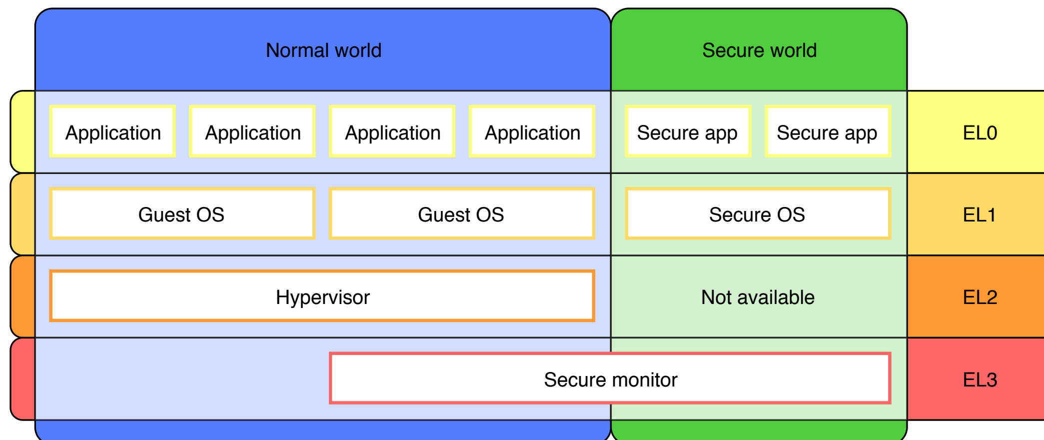


FIGURE 1.1 – Niveaux de privilèges de l'architecture ARMv8-A

- Les capteurs photographiques et audio.
- Les capteurs de mouvement (accéléromètre, gyroscope).
- Les capteurs de position (GPS, champ magnétique, proximité).
- Les capteurs environnementaux (température, lumière, pression, humidité).

L'ensemble des données produites par ces capteurs représente, là aussi, une menace pour la vie privée des utilisateurs de smartphone.

1.2 L'écosystème Android

Le marché des systèmes d'exploitation dédiés aux smartphones est globalement divisé entre deux acteurs : android développé par Google, et iOS par Apple. Nous avons choisi dans cette thèse de nous concentrer sur le système d'exploitation Android, dont le code a l'avantage d'être disponible sous licence open source contrairement à son rival. Il est en effet possible de le consulter depuis les dépôts Git de l'*Android Open Source Project* (AOSP) maintenus par Google depuis 2007. Toutefois, certaines parties mineures du système d'exploitation tel qu'il est distribué dans les smartphones restent sous licence propriétaire, comme les applications Google ou les modifications cosmétiques apportées par les différents constructeurs.

Depuis son rachat par Google en 2005, le système d'exploitation Android n'a cessé d'évoluer, tant en terme de fonctionnalités que de mesures de sécurité. Dans cette section, nous allons effectuer un tour d'horizon de l'écosystème Android en sa dernière version en date, Android 9.0 « Pie ». Pour se faire, nous allons premièrement présenter l'architecture et l'organisation des différentes couches logicielles le composant. Ensuite nous étudierons le modèle de sécurité mis en place au sein de cet OS. Enfin, nous exposerons le modèle d'attaque considéré pour cette thèse, et introduirons notre problématique.

1.2.1 Panorama des couches logicielles

1.2.1.1 Le noyau Linux

Premièrement, le système d'exploitation Android se base sur un noyau Linux légèrement modifié [2] qui provient exclusivement des branches *Long Term Support* (LTS) du noyau. Ces branches ont d'ailleurs obtenu un support étendu à 6 ans, depuis la sortie d'Android 8 « Oreo » et la mise en place du projet *Treble*, pour permettre aux constructeurs d'effectuer des mises à jour de sécurité pour toute la durée de vie du smartphone. Parmi les spécificités de la version Android du noyau, on retrouve notamment :

- **Binder** : un système de *Communication Inter Processus* (IPC) propre à Android dont le fonctionnement est décrit dans la section 1.2.2.
- **ION** : un driver d'allocation de mémoire physique.
- **Ashmem** : un gestionnaire de mémoire partagée.
- **Low Memory Killer** (LMK) : un driver utilisé pour stopper automatiquement les processus les moins « prioritaires » lors d'un manque de mémoire.
- **Paranoid networking** : un composant utilisé pour restreindre l'accès des processus aux sockets.

D'autres modifications mineures sont apportées au noyau telles que l'ajout ou la personnalisation de divers drivers [3], ainsi que l'ajout d'un régulateur de fréquence CPU et d'un planificateur de tâches optimisé. De plus, ces spécificités sont unifiées au fur et à mesure dans le noyau générique, ce qui facilite la maintenance et l'évolution de la version dédiée à Android.

1.2.1.2 Le framework Android

S'exécutant sous le contrôle du noyau Linux, le framework Android, aussi appelé *Android Application Framework* (AAF), représente la couche logicielle assurant le support d'exécution des applications Android. Celle-ci peut s'apparenter à un *middleware* fournissant aux développeurs la capacité de créer facilement des applications en s'appuyant sur un ensemble d'*interfaces de programmation* (API) et de services de haut niveau. Ainsi, la conception du framework n'est pas indifférente à l'adoption massive qu'a connue le système d'exploitation Android. Cette riche composition est notamment étudiée pour mettre à profit le plus possible la réutilisation de code afin d'économiser au maximum la mémoire, et ainsi prendre en compte une contrainte importante des smartphones. La figure 1.2 résume la composition de ce framework que l'on peut décrire en trois parties.

Premièrement, la couche *fonctionnalités* est consacrée à fournir les interfaces d'accès à l'ensemble des fonctionnalités offertes par le middleware. Ces interfaces

2. Environ 30000 lignes de différences par rapport au noyau générique
3. USB, système de fichiers (f2fs, sdcardfs), supports de SoC ...

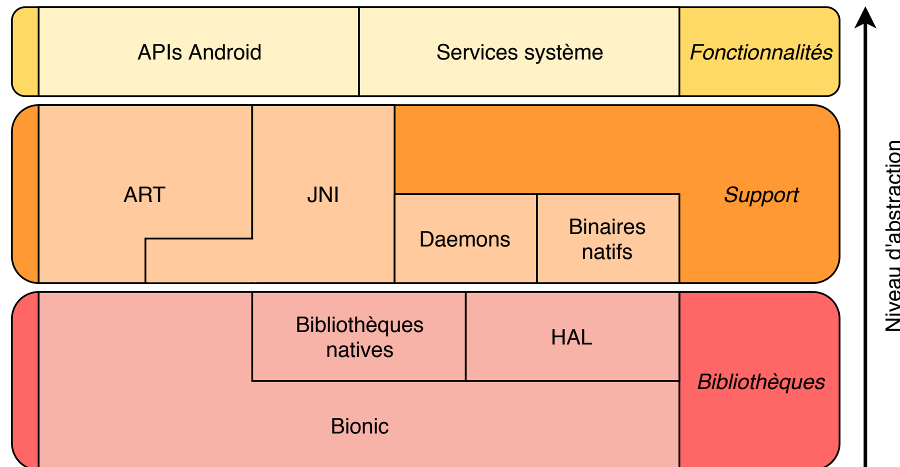


FIGURE 1.2 – Composition du framework Android

sont définies par le biais d'APIs qui sont de deux types : celles appartenant au *software development kit* (SDK) utilisables depuis les parties d'applications écrites en Java ; et celles définies dans le *native development kit* (NDK) pouvant être exploitées depuis du code natif (C/C++). De plus, les applications peuvent aussi faire appel à de nombreux services système, pouvant être définis comme un ensemble de « serveurs » responsables de la gestion des ressources, physiques ou logicielles, du smartphone. Par exemple, le service *Wi-Fi* s'occupe de la gestion du périphérique du même nom, et offre la capacité d'effectuer des connexions à n'importe quel réseau sans fil à proximité. Autre exemple, le service *PackageManager* permet de se renseigner sur certains détails concernant les applications installées. La plupart des services système sont écrits en Java, mais il existe aussi certains services système natifs. En outre, ces services sont le plus souvent consultés à travers les méthodes des APIs Android, et non pas directement même si cela reste possible.

Ensuite vient la couche *support* qui constitue le ciment liant les différents composants du framework entre eux. En effet, c'est elle qui assure le lien entre les APIs et services de haut niveau mis en place dans la couche fonctionnalités, et les ressources de plus bas niveau d'abstraction, en fournissant les mécanismes d'adaptation nécessaires à leur accès. Le plus important de ces mécanismes est l'environnement d'exécution Java, nommé *Android Runtime* (ART), qui est spécifique à Android. Celui-ci est chargé d'effectuer la compilation du bytecode *Dalvik EXecutable* (Dex), issu de la pré-compilation du code Java des applications, en code machine. Ce processus est effectué grâce à une combinaison de compilation anticipée (AOT), de compilation juste-à-temps (JIT) et de compilation guidée par profil. Le produit de cette compilation est un fichier ELF contenant à la fois le code machine correspondant aux méthodes compilées, ainsi que le bytecode Dex à des fins de debogage. De plus, l'ART embarque un interpréteur Dex utilisé pour exécuter le code des applications fraîchement installées, ou bien les sections de code n'ayant pas été compilées

car rarement utilisées. En outre, il assure le support d'exécution du bytecode ainsi compilé ou interprété de manière similaire à n'importe quelle machine virtuelle Java. On peut citer par exemple la fonctionnalité de ramasse miettes utilisé pour collecter les zones de mémoire n'étant plus utilisées par chaque application. En collaboration avec l'environnement d'exécution ART, la *Java Native Interface* (JNI) permet d'intégrer du code natif (C/C++) au code Java. Plus précisément, cette interface rend possible le chargement du code machine contenu dans des bibliothèques partagées ELF, et permet son interaction avec le bytecode exécuté dans l'environnement ART.

L'autre facette de cette couche support concerne les outils nécessaires au démarrage du système et à son maintien dans un état fonctionnel. Ces outils appartiennent à deux catégories. Tout d'abord la catégorie des binaires natifs qui regroupe le programme *init*, un shell dérivé du Korn Shell [4], certaines commandes Unix fournies par l'implémentation *Toybox*, et quelques autres binaires spécifiques à Android permettant d'interagir avec certains éléments du framework. Pour être plus précis concernant le binaire init, l'implémentation d'Android est quelque peu différente de son équivalent Unix. Celui-ci reste bien évidemment le premier binaire appelé par le noyau Linux au démarrage, et père de tous processus utilisateur, mais il est aussi chargé de la gestion des *propriétés système*. Ces dernières peuvent être décrites comme des registres stockant l'état et la configuration du système. Init est d'ailleurs capable de déclencher certaines actions lors de la modification de ces propriétés. Enfin, il est aussi responsable du démarrage de la deuxième catégorie d'outils : les *daemons*. Ceux-ci sont des programmes natifs, fonctionnant en arrière plan, et qui assurent certaines tâches de bas niveau. La plupart des daemons possèdent une ou plusieurs interfaces de communication (socket ou binder) utilisées le plus souvent depuis les services système ou les binaires natifs. Parmi ces daemons, on retrouve notamment *ServiceManager* et *Zygote*. *ServiceManager* est le daemon auprès duquel tous les services système doivent se déclarer lors de leur lancement. Ainsi, il fonctionne comme un annuaire de services auprès duquel les applications peuvent se renseigner pour découvrir l'ensemble des services leur étant disponibles. *Zygote* constitue quant à lui la base de tous les processus de haut niveau (i.e. des applications). Plus précisément, ce daemon a été étudié pour réduire le temps de lancement de ces processus et optimiser leur consommation mémoire. Ainsi, à son démarrage, il charge et initialise l'ensemble des composants nécessaires à l'exécution des processus de haut niveau (bibliothèques, descripteurs de fichiers, environnement ART...). Puis, lorsque l'un de ces processus doit être lancé, Zygote fork son processus et adapte les attributs et les privilèges du processus ainsi créé avant de donner la main au code du logiciel devant être lancé. La seule exception à cette règle concerne le premier processus créé de cette manière : le meta service système, nommé *System Server*, qui est chargé du lancement des autres services système dans des threads.

La dernière partie de ce framework est constituée des éléments de plus bas niveau d'abstraction s'assurant de la liaison entre les composants du framework et le noyau

4. MirBSD Korn Shell (mksh)

Linux. La base de cette couche est constituée de la bibliothèque *Bionic* qui est une implémentation de libC propre à Android. Outre les considérations de licences qui auraient empêché l'utilisation d'autres libC, celle-ci a été conçue de manière à être la plus simple possible. Par exemple, les wrappers des appels systèmes sont étudiés pour être les plus optimisés possibles, et il n'y a pas de support des IPC System V [5]. De plus, Bionic offre certaines fonctionnalités supplémentaires telles que :

- Le support partiel de l'API Pthread.
- La consultation des propriétés système.
- L'implémentation des UID/GID.
- Un résolveur DNS intégré.

Sur cette base s'appuie un ensemble de bibliothèques natives qui fournissent les fonctionnalités nécessaires au support d'exécution de la couche fonctionnalités. Parmi ces bibliothèques, certaines appartiennent à la couche d'abstraction matérielle, ou *Hardware Abstraction Layer* (HAL) en anglais. Cette couche constitue une interface qui standardise l'accès aux différentes ressources matérielles gérées depuis les pilotes présents dans le noyau Linux. L'implémentation de ces interfaces est réalisée par les fournisseurs de ce matériel. Ainsi, les daemons et services système en charge de chaque type de matériel n'ont pas besoin de connaître leurs spécificités en respectant cette interface. De plus, avec la mise en place du projet *Treble*, ces interfaces sont désormais versionnées et décrites dans un langage spécifique nommé *HAL interface definition language* (HIDL). En outre, les versions successives d'Android devront supporter un certain nombre de versions de ces interfaces pour permettre une meilleure compatibilité ascendante, et de ce fait faciliter le développement de nouvelles versions d'Android pour chaque appareil. Par ailleurs, cette évolution permet également de charger chacune de ces bibliothèques HAL dans des services spécifiques, servant de wrappers, qui doivent ensuite s'enregistrer auprès du daemon nommé *HWServiceManager*.

1.2.1.3 Les applications

Au delà du middleware que représente le framework, s'exécutent les applications Android. Celles-ci sont distribuées sous la forme d'archives zip contenant divers éléments : le code des classes Java compilé, les bibliothèques natives embarquées, les ressources utilisées (graphique, xml ...), et enfin des meta-données. Parmi ces meta-données, on retrouve principalement un fichier xml nommé `AndroidManifest.xml`, et plus communément appelé *manifeste*. Celui-ci décrit notamment l'ensemble des *composants* de l'application, implémentés dans des classes Java dédiées, et qui peuvent être de quatre types. Le premier, nommé *activité*, représente un ensemble d'éléments graphiques regroupés au sein d'un affichage (ou vue). Par exemple, une application de messagerie possède usuellement au moins deux activités : une pour l'affichage des contacts, et une pour la lecture des derniers messages d'un contact

5. Message queue, Semaphore et Mémoire partagée

et la rédaction de nouveaux messages. De plus, la navigation entre les activités s'effectue à l'aide d'une pile, appelée *back stack*, qui sera dépilée lors de l'appui sur la touche « retour ». Le *service* est le deuxième type de composant d'une application, qui est utilisé pour effectuer diverses tâches en arrière plan. Nous retrouvons ensuite le *fournisseur de contenu*, ou *content provider* en anglais, qui est utilisé pour le partage d'informations entre applications. À travers ce composant, il est en effet possible d'exposer certaines données qui peuvent ensuite être accédées via des requêtes similaires au langage SQL. Le dernier composant, nommé *récepteur de transmissions* ou *broadcast receiver* en anglais, sert à réceptionner les évènements asynchrones du système dans le but de déclencher certains traitements en conséquence. Par exemple, une application e-mail peut s'abonner à l'évènement `NETWORK_STATE_CHANGED_ACTION` pour consulter automatiquement les comptes enregistrés lorsqu'une connexion internet est établie.

Par ailleurs, comme la description de ces composants le laisse présager, leur comportement est dicté par des évènements asynchrones. Et plus précisément, les activités et services doivent respecter un cycle de vie, et réagir en conséquence de son évolution. Ainsi, certaines méthodes de classes sont responsables d'effectuer les traitements appropriés lors du changement de l'état du composant en question. Pour une activité, les quatre états possibles à prendre en compte sont : sa création, sa mise au premier plan, son passage en arrière plan [6], et sa destruction. Pour un service, celui-ci peut fonctionner sous deux modes : le *mode standard*, et le *mode lié*. Dans le mode lié, une référence au service est détenu par la ou les applications souhaitant l'utiliser. Ainsi un service est considéré comme actif s'il est démarré en mode normal ou lié, et considéré comme inactif lorsqu'il est arrêté et que plus aucune référence n'est possédée par des applications. En outre, les composants applicatifs peuvent aussi être tués indépendamment des actions de l'utilisateur, par le système LMK du noyau Linux. Cependant, l'occurrence de ce type d'évènement dépend fortement de l'état de ces composants. En effet, une « note » est attribuée à chaque processus applicatif en fonction de l'état des composants qu'il héberge, et les composants actifs ont ainsi de moindres chances d'être la cible du LMK.

1.2.2 Modèle de sécurité existant

De la même manière que l'architecture d'Android repose sur le noyau Linux, ce dernier est aussi utilisé à son avantage pour le modèle de sécurité du système. Le premier cas d'utilisation réside dans les mécanismes d'isolation employés pour assurer le sandboxing des applications. Celui-ci s'appuie sur les mécanismes de contrôle d'accès mis en place à travers les utilisateurs et groupes Linux ; ces derniers étant représentés par des identifiants appelés respectivement *User ID* (UID) et *Group ID* (GID). De plus, chaque fichier se voit attribuer un utilisateur propriétaire et des groupes d'appartenance auquel il est possible d'associer des droits d'accès. Le système Android utilise habilement ce mécanisme en associant un UID à chaque application installée. De plus, un répertoire privé est assigné à chaque application,

6. Les activités n'effectuent plus de traitements lorsqu'elles sont en arrière plan

et ces dernières sont les seules à posséder les droits d’accès en lecture ou en écriture sur ce répertoire. Ainsi, lorsque le daemon zygote lance une application, il assigne au processus d’accueil l’UID de l’application démarrée. De cette façon, toutes les applications installées se retrouvent isolées les unes des autres, tant au niveau processus (en s’exécutant dans des processus différents) qu’au niveau fichier (en possédant des répertoires privés). Les applications système ont le droit au même traitement, et ne diffèrent des applications utilisateur que de part leur emplacement d’installation : les applications système sont installées sur la partition système, étant uniquement accessible en lecture seule, tandis que les applications des utilisateurs le sont sur la partition data. De manière similaire, les daemons et binaires natifs se voient assigner des UIDs, définis de manière statique, limitant ainsi leurs capacités dans le but d’empêcher de potentielles attaques.

1.2.2.1 Moyens de communication

Étant donnée cette isolation forte mise-en-place par le noyau, il est indispensable de posséder des mécanismes de communication pour que les applications soient en mesure d’accéder aux fonctionnalités offertes par les services système et les autres applications installées. De plus, ces moyens de communication devront être mis-en-œuvre au moins en partie par le noyau, car la seule surface de communication existante du point de vue d’un processus ainsi isolé est celle des appels systèmes. Ces derniers sont dans un premier temps utilisés par les applications, via la bibliothèque Bionic, pour accéder aux fonctionnalités du noyau, et aux IPC natifs proposés par celui-ci (fichiers partagés, sockets, ...). De plus, ceux-ci sont utilisés dans un second temps pour faire transiter les messages des IPC de plus haut niveau, comme ceux du Binder.

Le Binder est le mécanisme de communication privilégié des composants d’Android et spécifique à cette plateforme. Il est d’ailleurs référencé sous le nom d’*Inter Component Communication* (ICC), en lieu et place d’IPC, car il a pour but de faire communiquer les composants des applications entre eux, ainsi qu’avec les services système. En effet, ce moyen d’échange puise sa source dans la volonté de réutilisation inscrite dans la conception d’Android en offrant la possibilité aux applications de partager leurs fonctionnalités. Pour ce faire, le Binder implémente un modèle client-serveur où chaque composant, aussi appelé *Binder Node* (BN), est capable d’effectuer des *Remote Procedure Calls* (RPC) ciblant les autres composants du système exposant des services. De plus, les BN clients ne connaissent qu’un seul BN serveur par défaut : celui appartenant au ServiceManager. Ce dernier est ainsi utilisé comme un annuaire répertoriant l’ensemble des services système : lorsqu’il est interrogé par un BN client, il renvoie une référence vers le BN serveur du service recherché.

Plus précisément, le fonctionnement du Binder repose sur un module du noyau Linux assurant la communication inter-processus. Il est accessible depuis l’espace utilisateur à partir de la bibliothèque *libbinder* qui communique avec le *character device* associé au module (`/dev/binder`) via des appels systèmes *ioctl*. En outre,

chaque serveur décrit l'ensemble des fonctions exposées aux clients via un fichier d'interface *Android Interface Definition Language* (AIDL). À partir de ce fichier d'interface, sont générées les classes *proxy* et *stub*, utilisées respectivement par le client et le serveur, et qui sont responsables de la sérialisation des données transportées par les transactions binder. Ainsi, le client utilise les fonctions du serveur de manière transparente, et la complexité inhérente à la communication lui est masquée, comme le montre la figure 1.3.

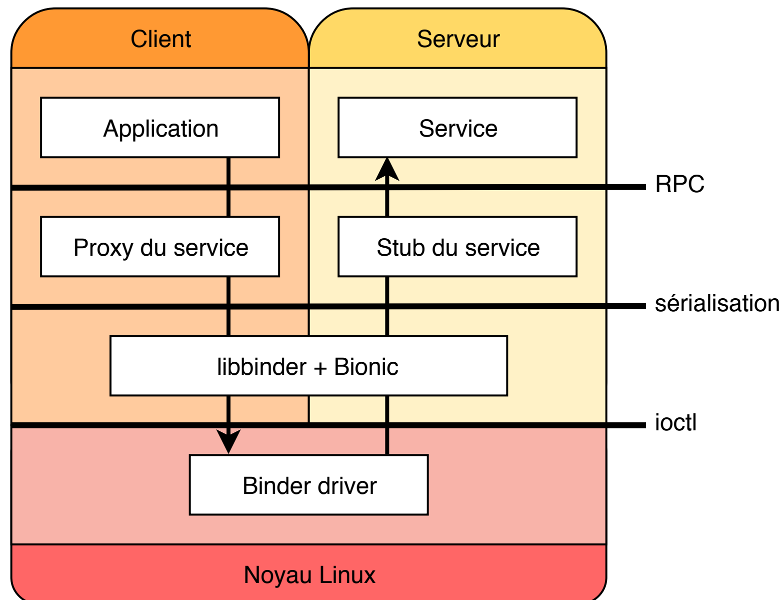


Figure 1.3 – Fonctionnement du Binder

Au delà de l'envoi de données sérialisables, les transactions binder peuvent aussi contenir des références à des descripteurs de fichiers. Ces descripteurs de fichiers peuvent notamment correspondre à des zones de mémoire partagées créées par le driver ashmem.

Par ailleurs, les transactions binder sont utilisées pour transporter des messages de plus haut niveau d'abstraction, appelés *Intent*, permettant de décrire une action à effectuer par un autre composant applicatif[7]. La cible de cette action peut être mentionnée de manière explicite, ou bien implicite en décrivant le type d'action à effectuer (comme envoyer un SMS par exemple). Le composant approprié sera ainsi démarré s'il ne l'était pas, et interrogé pour effectuer l'action demandée. La gestion des intents, et de leurs résolutions, est effectuée par le service système *Activity Manager Service* (AMS). Plus précisément, ce service est responsable du démarrage et de la gestion du cycle de vie des composants applicatifs ciblés par les intents. Il joue ainsi le rôle, d'une part de médiateur des intents, et d'autre part de gestionnaire des composants applicatifs. Pour plus de détails sur le fonctionnement interne du Binder, le lecteur est invité à consulter [Latini 2014].

7. D'une même application ou d'une autre application.

1.2.2.2 Permissions des applications

L'autre point de contrôle des capacités d'une application réside dans le système de permissions. En effet, l'accès aux ressources exposées par les services système et les applications est protégé par des droits d'accès. Il existe ainsi un certain nombre de permissions natives, définies dans l'API Android, et restreignant de manière assez fine l'accès aux ressources matérielles et logicielles fournies par l'API. Il est cependant possible pour les développeurs d'applications d'enrichir celles-ci en créant des permissions personnalisées. Ainsi, ces permissions sont définies selon plusieurs niveaux :

- Les permissions normales pouvant être accordées aux applications sans avoir besoin d'une confirmation de la part de l'utilisateur.
- Les permissions dangereuses, donnant accès à des données sensibles de l'utilisateur, ou permettant d'effectuer des actions critiques sur l'appareil.
- Les permissions protégées par signature, dont l'utilisation est réservée aux applications signées par la même clé cryptographique que celle associée à la permission. Ce type de permissions est le plus souvent utilisé par les applications système qui sont signées par une même *platform key*.
- Les permissions spéciales, relatives à des actions particulièrement sensibles, et requérant une approbation approfondie [8] de l'utilisateur.

Les permissions requises par une application doivent être déclarées dans le manifeste de celle-ci. L'utilisateur est ensuite amené à accorder ces permissions à l'installation de l'application, ou bien à leur exécution depuis Android 6.0, lorsque l'application en question nécessite celle-ci pour une de ses fonctionnalités.

1.2.2.3 Points d'application du modèle de sécurité

Ce système de permissions est ensuite contrôlé dans différentes parties de l'écosystème Android. Premièrement, certaines permissions sont associées à des groupes Linux prédéfinis. Celles-ci sont le plus souvent amenées à restreindre la capacité de créer ou d'accéder à certains objets du noyau, tels que les sockets ou les fichiers. Par exemple, les applications possédant la permission `INTERNET` sont assignées au groupe `inet`. Le respect de ces permissions est donc assuré directement par les droits d'accès Linux, qui peuvent être assistés par d'autres composants du noyau comme le *paranoid networking*. Concernant les permissions relatives à des ressources accessibles via le binder, celles-ci sont directement contrôlées par le logiciel responsable de la ressource, que ce soit un service système, un daemon ou une application. En effet, comme il est impossible pour une application source de falsifier son identité à travers ce mécanisme de communication, il est aisé de vérifier les permissions détenues par celles-ci en interrogeant le service système *PackageManager*. De plus,

8. Un affichage contenant un ensemble d'explications relatives à la permission est présenté à l'utilisateur qui doit expressément donner son consentement.

des vérifications supplémentaires sont effectuées dans AMS afin de vérifier si les permissions nécessaires à l'exécution d'un intent sont respectées.

La sécurité des applications est aussi assurée lors de leur distribution et leur installation. Chaque application est en effet associée à un identifiant unique, nommé *Application ID*, renseigné dans le manifeste de celle-ci, et doit être signée par une ou plusieurs signatures cryptographique. Celles-ci sont ensuite vérifiées lors de leur installation et mise à jour. Cela permet ainsi au système de s'assurer que le contenu des applications n'a pas été altéré pendant la phase de distribution, ainsi que de vérifier que les mises à jour sont issues du ou des même(s) développeur(s). Les applications système sont aussi concernées par le système de signature. Ces dernières sont en effet signées par un ensemble de clés générées par la personne en charge de la création de l'image système, c'est-à-dire le constructeur dans la plupart des cas. Le rôle de la signature ne s'arrête pas seulement à ces tâches. En effet, deux applications signées par une même clé cryptographique peuvent bénéficier de traitements particuliers. Notamment, elles ont la capacité de faire exception à la règle d'isolation en s'associant à un même UID et en partageant un même processus. De nombreuses applications système utilisent ce mécanisme pour pouvoir partager des ressources facilement. Concernant les applications des utilisateurs, la pratique est plus rare, d'autant plus qu'il est nécessaire de définir un paramètre spécifique dans le manifeste de chaque application concernée dès leur première installation.

Pour plus de détails sur le modèle de sécurité mis en œuvre par le système d'exploitation Android, le lecteur est invité à consulter [Mayrhofer 2019].

1.3 Motivations des attaquants et problématique

Après avoir passé en revue l'écosystème Android et son modèle de permissions, nous pouvons légitimement nous poser la question de son adéquation avec les menaces auxquelles il fait face. La position particulière du smartphone, issue d'une part de sa grande capacité de connectivité, et d'autre part des nombreuses données qu'il héberge ou génère, en fait une cible de choix pour de nombreux attaquants.

Dans nos travaux, l'attaquant considéré peut avoir différents buts. Il peut vouloir chercher à compromettre la sécurité du smartphone en question ; c'est-à-dire de compromettre l'intégrité, la confidentialité, ou la disponibilité du smartphone ou des objets avec lesquels il est connecté. Le but d'un attaquant peut se manifester de différentes façons. Il peut, par exemple, s'attaquer à l'intégrité ou la disponibilité du smartphone pour générer de l'argent en installant un adware, un malware crypto-mineur ou un ransomware. Il peut aussi en prendre le contrôle pour espionner son utilisateur, ou intégrer l'appareil au sein d'un botnet. De plus, il peut chercher à compromettre l'intégrité physique du smartphone pour nuire à son utilisateur. Du point de vue de la confidentialité, il peut aussi vouloir chercher à voler les données personnelles ou professionnelles présentes sur le smartphone, qu'elles soient stockées sur le système de fichiers de l'appareil ou générées à partir des différents périphériques de celui-ci. Les attaquants simplement curieux sont plus concentrés sur des

violations de la confidentialité. On peut citer comme exemple les bibliothèques de publicité qui vont mettre en œuvre des techniques de fingerprinting ou de tracking, en se servant des données personnelles des utilisateurs ou des identifiants du smartphone pour mieux effectuer leurs ciblages publicitaires.

Dans cette thèse, nous avons relevé un manque dans les capacités d'expression du modèle de permission d'Android vis-à-vis de ces menaces. Plus particulièrement, il est impossible de définir un contrôle d'accès dynamique des ressources et actions protégées par les permissions existantes. En effet, une fois qu'une permission est accordée à l'application, les droits d'accès associés peuvent être utilisés quel que soit le contexte d'exécution du smartphone. Or, il existe certaines attaques reposant sur l'utilisation concurrente de ressources système mises à disposition des applications. Par exemple, [Sivakumaran 2018] et [Naveed 2014] ont démontré la possibilité de détourner des communications *Bluetooth Low Energy* (BLE) et Bluetooth, établies entre une application et un appareil externe, à partir d'une application malveillante hébergée sur le même smartphone. Il est important d'envisager que ce type d'attaque soit étendu à d'autres types de périphériques. Par ailleurs, les applications exécutées sur un smartphone peuvent être de différents niveaux de criticité. Un utilisateur n'a cependant pas la capacité de classifier les applications installées selon le niveau de confiance qu'il leur accorde. Ce problème est d'autant plus important du fait de certaines permissions qui, comme mentionné dans la section 1.2.2, sont particulièrement sensibles et accordent à leur détenteur des capacités supplémentaires de contrôle sur l'appareil. Là encore, il est impossible avec l'implémentation actuelle de restreindre l'usage de ces permissions lorsque les applications les plus critiques, comme les applications bancaires, sont démarrées. Par exemple, l'attaque *Cloak and Dagger* [Fratantonio 2017] repose sur le détournement de deux de ces permissions étant utilisées pour intercepter et modifier de manière furtive les interactions de l'utilisateur avec une application cible.

1.4 Conclusion

Dans ce premier chapitre, nous avons présenté le contexte relatif à l'environnement matériel et logiciel étudié dans cette thèse, celui-ci étant nécessaire à la bonne compréhension du reste du manuscrit. Suite à l'étude de ce contexte, nous avons également présenté la problématique de recherche abordée au cours de cette thèse, et qui est issue de faiblesses que nous avons pu mettre en lumière au cours de ce chapitre. Pour compléter cette analyse, le chapitre 2 est dédié à l'étude des politiques de contrôle d'accès pour Android, et des moyens d'isolation pouvant être utilisés pour les mettre en œuvre.

Chapitre 2

État de l'art

Sommaire

2.1 Terminologie de la sécurité informatique 20
2.1.1 Sûreté de fonctionnement 20
2.1.2 Sécurité-immunité 21
2.1.3 Politiques de sécurité 23
2.2 Mécanismes de contrôle d'accès pour Android 27
2.2.1 Mécanismes de niveau système 27
2.2.2 Mécanismes de niveau OS 29
2.2.3 Mécanismes de niveau framework 32
2.2.4 Mécanismes de niveau applicatif 34
2.3 Politiques d'autorisation pour Android 35
2.3.1 Considérations relatives à l'environnement Android 35
2.3.2 Politiques d'autorisation de la littérature 37
2.4 Conclusion et démarche scientifique 40

Dans ce chapitre dédié à l'état de l'art, nous allons discuter des travaux de recherches existants ayant porté sur la conception de politiques de sécurité adaptées à l'environnement Android, ainsi que des moyens techniques utilisés pour les mettre en œuvre. En effet, ce système d'exploitation mobile possède des spécificités nécessitant l'adaptation des politiques de sécurité de référence, ayant été conçues pour des systèmes d'information dit « classiques », en comparaison avec les systèmes dit « appifiés » dont Android est un des représentants notables [Acar 2016]. Par ailleurs, au delà des challenges supplémentaires qui sont imputés à leur conception, se pose la question de leur mise en œuvre. Ainsi, ce chapitre s'articule en quatre parties. La première présente les termes nécessaires à la bonne compréhension des travaux évoqués dans ce chapitre. Une deuxième partie se concentre sur l'état de l'art des mécanismes de contrôle d'accès pouvant être utilisés dans le cadre de politiques de sécurité ciblant l'environnement Android. Ensuite, une troisième partie offre un aperçu des différents types de politiques de sécurité de la littérature adaptés au système d'exploitation Android. Enfin, une dernière partie récapitule de manière synthétique les défis restants à aborder et présente les contributions de cette thèse en ce sens.

2.1 Terminologie de la sécurité informatique

Le but de cette section est de faire un état des lieux de la terminologie de la sécurité informatique qui est une composante de la sûreté de fonctionnement. Ensuite une attention plus particulière sera dévolue aux politiques de sécurité et aux termes leur étant associés.

2.1.1 Sûreté de fonctionnement

La *sûreté de fonctionnement* d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre [Laprie 1996]. Le service délivré correspond au comportement du système perçu par ses utilisateurs. La sûreté de fonctionnement peut être décomposée selon trois axes principaux : des *attributs* qui définissent les propriétés assurées, des *entraves* précisant les obstacles à surmonter, et des *moyens* permettant d'assurer la délivrance du service. Ces trois axes sont présentés dans les paragraphes suivants.

Selon les applications auxquelles le système est destiné, l'accent peut être mis sur différentes facettes de la sûreté de fonctionnement. Ainsi la sûreté de fonctionnement peut être vue selon des propriétés différentes mais complémentaires, qui permettent de définir ses attributs :

- Le fait d'être prêt à l'utilisation conduit à la *disponibilité*.
- La continuité du service conduit à la *fiabilité*.
- La non-occurrence de conséquences catastrophiques pour l'environnement conduit à la *sécurité-innocuité*.
- La non-occurrence de divulgations non-autorisées de l'information conduit à la *confidentialité*.
- La non-occurrence d'altérations inappropriées de l'information conduit à l'*intégrité*.
- L'aptitude aux réparations et aux évolutions conduit à la *maintenabilité*.

Ainsi ces attributs permettent, d'une part d'exprimer les propriétés devant être respectées par le système, et d'autre part d'évaluer la qualité du service délivré vis-à-vis de ces propriétés. Les causes du non respect de ces attributs se nomment entraves à la sûreté de fonctionnement. Ces dernières peuvent être de trois types :

- Une *défaillance* survient lorsque le service délivré dévie de l'accomplissement de la fonction du système.
- Une *erreur* est la partie de l'état du système qui est susceptible d'entraîner une défaillance.
- Une *faute* est la cause adjugée ou supposée d'une erreur.

Pour minimiser l'impact de ces entraves sur les attributs retenus d'un système, la sûreté de fonctionnement dispose de méthodes et techniques qui permettent de conforter les utilisateurs quant au bon accomplissement des fonctions du système.

Le développement d'un système sûr de fonctionnement passe donc par l'utilisation combinée de ces méthodes, appelés moyens, pouvant être classée en quatre types :

- La *prévention des fautes* empêche l'occurrence ou l'introduction de fautes.
- La *tolérance aux fautes* permet de fournir un service à même de remplir la fonction du système en dépit des fautes.
- L'*élimination des fautes* réduit la présence (nombre, sévérité) des fautes.
- La *prévision des fautes* estime la présence, la création et les conséquences des fautes.

Dans la suite de ce mémoire, nous nous intéressons uniquement à la sécurité-immunité qui se définit comme la combinaison des attributs de disponibilité, de confidentialité et d'intégrité.

2.1.2 Sécurité-immunité

La *sécurité-immunité* (en anglais security) a pour but de protéger un système contre les fautes intentionnelles dues à l'homme, aussi appelées *malveillances*. Elle s'oppose à la *sécurité-innocuité* (en anglais safety) dont le domaine d'application concerne la prévention de catastrophes ayant des répercutions sur l'environnement du système. Dans la suite de ce manuscrit, nous emploierons uniquement les termes de sécurité ou de sécurité informatique pour faire référence à la sécurité-immunité.

La sécurité d'un système informatique est intrinsèquement liée à la sécurité des informations qu'il possède. Ainsi, sécuriser un système informatique consiste, d'une part à prévenir les accès et les manipulations illégitimes des informations du système, et d'autre part à garantir les accès et les manipulations légitimes de ces mêmes informations. Toutes opérations vouées à mettre à mal la sécurité d'un système devront ainsi transgresser un certain nombre de règles et de propriétés de sécurité, définies dans le cadre d'une *politique de sécurité*, qui encadrent l'ensemble des accès et manipulations réalisables par les acteurs du système. Par ailleurs ces propriétés de sécurité sont inhérentes aux trois attributs de la sûreté de fonctionnement qui, lorsqu'ils sont combinés, définissent plus précisément la portée de la sécurité-immunité. Les prochains paragraphes présentent les définitions de ces trois attributs affinées au contexte de la sécurité informatique.

2.1.2.1 La confidentialité

La *confidentialité* désigne la propriété d'une information à ne pas être divulguée aux utilisateurs non autorisés à la connaître. Cela implique que le système doit rendre inaccessibles (ou incompréhensibles) les informations aux utilisateurs non autorisés à y accéder. Cette exigence nécessite que le système :

- empêche un utilisateur de consulter directement une information qu'il n'est pas autorisé à connaître.
- empêche un utilisateur qui accède légitimement à une information de la divulguer à un utilisateur non autorisé à y accéder.

Dans cette définition, le terme d'information désigne à la fois les données en elles-même, mais aussi les méta-données lui étant associées, c'est-à-dire toutes les données indirectes résultantes de la création, de la transmission et de la manipulation de cette information. On peut illustrer ceci en prenant l'exemple de la *protection de la vie-privée* (en anglais privacy), attribut de la sécurité-immunité dérivé de la confidentialité. En effet, cet attribut se rapporte directement à la confidentialité des informations à caractère privé (données personnelles) et de leurs méta-données (identité de l'auteur des informations, données temporelles et localisation ...).

2.1.2.2 L'intégrité

L'*intégrité* désigne la propriété d'une information à ne pas être corrompue ou altérée par un utilisateur non autorisé. Cela implique que le système doit :

- rendre impossible l'altération (création, modification ou destruction) d'une information par un utilisateur non autorisé à effectuer ce type d'opération sur l'information en question.
- faire en sorte qu'aucun utilisateur ne puisse empêcher les utilisateurs autorisés à légitimement modifier une information.

2.1.2.3 La disponibilité

La *disponibilité* désigne la propriété d'une information à rester accessible lorsqu'un utilisateur autorisé en a besoin. Cela implique que le système doit :

- fournir un moyen d'accès à l'information pour les utilisateurs autorisés.
- faire en sorte qu'aucun utilisateur ne puisse effectuer de rétentions d'informations auprès des utilisateurs autorisés.

2.1.2.4 Fautes malveillantes et mesures de sécurité

Toutefois, les violations d'un ou plusieurs de ces attributs peuvent être à la fois dues à des malveillances ou à des fautes non intentionnelles dues à l'homme. Cependant dans ces travaux nous nous concentrons exclusivement sur les fautes malveillantes.

Les *fautes intentionnelles*, ou *malveillances*, se déclinent en deux classes principales : les logiques malignes et les intrusions. Les *logiques malignes* sont des fautes internes intentionnelles, qui sont conçues pour provoquer des dégâts (bombes logiques) ou pour faciliter les futures intrusions par l'ajout de vulnérabilités. Elles peuvent être introduites dans le système, soit avant sa mise en service (par un concepteur malveillant), soit durant son exploitation à la suite de l'installation d'un cheval de Troie ou d'une intrusion. Le deuxième type de malveillance, l'*intrusion*, est intrinsèquement lié aux concepts d'attaque et de vulnérabilité :

- Une *attaque* est une faute d'interaction externe dont le but est de violer un ou plusieurs attributs de sécurité du système. Une attaque résulte obligatoi-

rement d'une volonté de nuire, même si celle-ci est lancée depuis des outils automatiques. Une tentative d'intrusion en est un synonyme.

- Une *vulnérabilité* est une faute accidentelle ou intentionnelle (avec ou sans volonté de nuire) placée dans les exigences, la spécification, la conception ou la configuration du système, ou dans la manière dont il est utilisé.
- Une *intrusion* est une faute malveillante interne d'origine externe, résultant d'une attaque qui a réussi à exploiter une vulnérabilité. Elle peut-être amenée à produire des erreurs pouvant provoquer une défaillance vis-à-vis de la sécurité-immunité, c'est-à-dire une violation de la politique de sécurité du système.

Pour faire face à l'ensemble de ces entraves menaçant la sécurité d'un système informatique, il existe de nombreux moyens pouvant être mis en place dont le but étant, bien entendu, de fournir une protection envers ces menaces. L'objectif de cette protection est double : elle vise d'une part à empêcher les opérations non autorisées dans un système, et d'autre part à limiter la propagation des erreurs dans ce système. Nous pouvons citer par exemple, les mécanismes cryptographiques, les méthodes d'isolation ou de cloisonnement, ainsi que d'autres techniques de tolérance et détection d'intrusions. Dans nos travaux, nous avons choisi de nous concentrer sur les politiques de sécurité.

2.1.3 Politiques de sécurité

La politique de sécurité d'un système est garante du respect des propriétés de sécurité que l'on souhaite assurer pour celui-ci. De plus, elle est aussi chargée de définir la façon dont ces propriétés seront garanties. Plus précisément, les ITSEC [European Commission 1992] définissent une politique de sécurité comme étant « l'ensemble des lois, règles et pratiques qui régissent la façon dont l'information sensible et les autres ressources sont gérées, protégées et distribuées à l'intérieur d'un système spécifique ». Il existe trois types de politiques de sécurité dont les secteurs d'applications diffèrent :

- Les *politiques de sécurité physique* qui s'attardent sur la situation physique du système (vol, catastrophes naturelles, ...).
- Les *politiques de sécurité administrative* qui traitent des considérations de sécurité organisationnelle au sein d'entreprises.
- Les *politiques de sécurité logique* font référence à la gestion du contrôle d'accès logique des systèmes informatiques. Elles doivent spécifier *qui* a accès à *quoi* et dans *quelles circonstances*. C'est à ce dernier type de politique que nous allons nous intéresser.

2.1.3.1 Politiques de sécurité logique

Une politique de sécurité logique peut être décomposée en trois sous-parties qui représentent les différentes étapes dont un utilisateur doit s'acquitter pour pouvoir

accéder à une information du système. Tout d'abord, un utilisateur devra décliner son identité (*identification*) et prouver qu'il est bien la personne qu'il prétend être (*authentification*). Une fois ces deux processus effectués, la liste des actions légitimes que cet utilisateur peut réaliser est définie dans la *politique d'autorisation*. Cette politique d'autorisation implémente ainsi l'intégralité des contrôles d'accès du système qui peuvent être conceptuellement séparés en deux branches.

D'une part, un ensemble de *propriétés de sécurité* à satisfaire. Celles-ci portent sur la confidentialité, l'intégrité et la disponibilité, et stipulent de multiples contraintes sur ces propriétés. Par exemple « une information classifiée ne doit pas être transmise à un utilisateur non habilité à la connaître ». D'autre part, un ensemble de règles décrivant les opérations autorisées, interdites ou obligatoires pour tous les acteurs du système, et ce conformément aux propriétés de sécurité établies. Par exemple, « le propriétaire d'une information peut accorder un droit d'accès pour cette information à n'importe quel utilisateur ». L'ensemble de ces règles forme un *schéma d'autorisation*.

Ainsi, si la politique d'autorisation est cohérente, il ne doit pas être possible, partant d'un état initial sûr (c'est-à-dire satisfaisant les propriétés de sécurité), d'atteindre un état d'insécurité (c'est-à-dire un état où les propriétés de sécurité ne sont pas satisfaites) en appliquant le schéma d'autorisation.

Les politiques de sécurité reposent habituellement sur les notions de sujets, d'objets et de droits d'accès, qui sont notamment utilisés pour définir l'ensemble des règles qui la composent. Un *sujet* est classiquement défini comme une entité active, correspondant à un processus qui s'exécute pour le compte d'un utilisateur. Dans ce contexte, un *utilisateur* est une personne physique ou morale connue du système informatique et enregistrée comme utilisateur. Un *objet* est une entité considérée comme « passive » qui contient ou reçoit des informations (données ou méta-données). Ainsi, un sujet a un *droit d'accès* sur un objet si et seulement si le processus correspondant au sujet est autorisé à exécuter l'opération correspondant à ce type d'accès sur cet objet. Cette décision d'autorisation est déterminée par la synthèse des règles (autorisations, interdictions, obligations) de la politique d'autorisation.

2.1.3.2 Types de schéma d'autorisation

Les politiques de sécurité, ou plus précisément leurs schémas d'autorisation, se classent en deux grandes catégories : les *politiques discrétionnaires* (nommées DAC pour *Discretionary Access Control*) et les *politiques obligatoires* (nommées MAC pour *Mandatory Access Control*).

Dans le cas d'une politique discrétionnaire, les droits d'accès à chaque information sont manipulés librement par le responsable de l'information (généralement le propriétaire), à sa *discrétion*. La gestion des droits d'accès aux fichiers des systèmes d'exploitation UNIX constitue un exemple d'implémentation de politique discrétionnaire, où les utilisateurs peuvent gérer les droits d'accès (lecture, écriture, exécution) des fichiers leur appartenant. Plus précisément, le propriétaire d'un fi-

chier peut librement attribuer ou non ces droits d'accès à lui-même, un groupe d'utilisateurs, et aux autres utilisateurs. Un des problèmes des politiques discrétionnaires est que celles-ci ne sont applicables que s'il est possible de faire entièrement confiance aux utilisateurs et aux sujets qui s'exécutent pour leur compte. Celles-ci sont en effet sensibles aux *fuites d'informations* (un utilisateur qui a le droit de lire une information peut, en général, la retransmettre à n'importe qui) ainsi qu'aux *abus de pouvoirs* (un cheval de Troie peut modifier les droits d'accès des fichiers de l'utilisateur pour lequel il s'exécute).

Pour répondre à ces problèmes, les politiques discrétionnaires peuvent être utilisées conjointement à des politiques obligatoires qui imposent, par leur schéma d'autorisation, des règles incontournables qui viennent s'ajouter aux règles discrétionnaires. Une politique obligatoire suppose que les utilisateurs et objets aient été étiquetés. Un exemple classique de politique obligatoire est celle du DoD (*Department of Defense*), formalisée par Bell et La Padula. Dans celle-ci, chaque objet se voit attribuer une *classification* représentant la sensibilité de l'information contenue, et chaque utilisateur est assigné à une *attribution* symbolisant son habilitation d'accès aux informations sensibles. En outre, les règles obligatoires définies dans le schéma d'autorisation permettent de valider les propriétés de confidentialité empêchant notamment un sujet ne possédant pas l'habilitation nécessaire à l'accès à une information sensible.

Il existe également des variantes de ces politiques étudiées pour répondre à des besoins particuliers, comme les politiques obligatoires basées sur la notion de rôles (nommées RBAC pour *Role-Based Access Control*) visant à faciliter l'administration de la sécurité. Un rôle représente de façon abstraite une fonction identifiée dans l'organisation (par exemple, chef de service, ingénieur d'étude, etc.). À chaque rôle, on associe des permissions (ou privilèges) définissant l'ensemble des droits nécessaires à la réalisation des tâches de chaque rôle. Ainsi les permissions ne sont plus associées d'une façon directe aux sujets, mais à travers des rôles. De plus, un sujet peut être membre de plusieurs rôles et inversement, un rôle peut être exécuté par plusieurs sujets. En outre, les autorisations de ces rôles peuvent être héritées via une hiérarchie de rôles.

D'autres efforts plus récents ont vu naître une autre variante de politiques de sécurité étant basées sur la notion d'attributs (nommées ABAC pour *Attribute-Based Access Control*) [Hu 2014]. Plus précisément, les accès du système sont accordés ou refusés en fonction des attributs du sujet, des attributs de l'objet auquel il cherche à accéder, des conditions environnementales, et d'un ensemble de règles conditionnées à partir de ces informations. Cette politique de sécurité peut être discrétionnaire ou obligatoire en fonction de son implémentation. De plus, on note l'apparition de la notion de contexte, qui désigne ici un ensemble de caractéristiques environnementales étant indépendantes des sujets ou des objets du système. De manière plus générale, les politiques de sécurité se basant sur l'utilisation du contexte sont regroupés sous l'appellation de politiques *sensibles au contexte*, ou CAAC pour *Context-Aware Access Control*.

2.1.3.3 Modèles de sécurité

Outre la classification de leur schéma, les politiques d'autorisation sont généralement représentées par un formalisme (souvent mathématique), appelé *modèle de sécurité*, permettant de les décrire de façon claire et non ambigüe. Ce formalisme peut avoir deux objectifs : d'une part, il peut permettre de faciliter la compréhension des objectifs de sécurité et du schéma d'autorisation sous-jacent par l'abstraction qu'il fournit ; d'autre part, il peut servir à vérifier que la politique d'autorisation est complète et cohérente, et que sa mise en œuvre par le système de protection est conforme aux propriétés attendues du système.

Il existe deux types de modèles : les modèles spécifiques, développés pour représenter une politique d'autorisation particulière, comme les modèles de Bell-LaPadula [Bell 1976], de Biba [Biba 1977], et XACML [Lorch 2003] ; et les modèles généraux, qui sont plutôt des méthodes de description formelle pouvant s'appliquer à n'importe quelle politique comme les modèles HRU [Harrison 1976] ou Take-Grant [Jones 1976].

Plus précisément, les modèles HRU et Take Grant s'appuient sur une *matrice de contrôle d'accès* pour représenter les droits d'accès ; les lignes et colonnes de celle-ci correspondent ainsi aux sujets et objets du système, et les droits leur étant associés sont inscrits dans les cellules de cette matrice. Des règles sont ensuite établies pour faire évoluer le contenu de la matrice. Les modèles de Bell-LaPadula et de Biba reposent quant-à-eux sur la notion de treillis permettant de représenter les politiques d'autorisation MAC dits « multi-niveaux », respectivement pour la confidentialité et l'intégrité. Enfin le modèle XACML (*eXtensible Access Control Markup Language*) est dédié à la représentation des politiques d'autorisation ABAC, et RBAC par extension. Il définit : d'une part, la liste des éléments logiciels devant mettre en œuvre la politique d'autorisation et leurs champs d'application respectifs ; d'autre part, un langage basé sur XML permettant de décrire les règles de la politique d'autorisation.

2.1.3.4 Mécanismes de contrôle d'accès et moniteur de référence

Pour faire appliquer la politique d'autorisation telle quelle a été conçue, un système a besoin d'implémenter un ensemble de solutions techniques adaptées, que l'on peut aussi appeler *mécanismes d'isolation*.

Ces mécanismes peuvent être destinés à créer une isolation forte entre plusieurs éléments d'un système. Ainsi, les sujets ne peuvent accéder aux objets dont ils ne possèdent pas les droits d'accès, car l'accès physique ou logique à ces objets ne leur est pas possible. Ce type d'isolation s'appuie sur des mécanismes de *cloisonnement*, dont le but est de couper les accès aux ressources en question, ou de *chiffrement*, dont l'isolation est rendue effective par l'inintelligibilité des données appartenant aux ressources.

Une autre famille de mécanismes repose sur l'interception et la médiation de l'ensemble des communications du système. Le ou les composants matériels ou logi-

ciels chargés de cette interception sont appelés *moniteur de référence*. Ainsi, le but de ce moniteur de référence est d'appliquer les règles de contrôle d'accès de la politique d'autorisation. Pour ce faire, un moniteur de référence doit obligatoirement posséder les trois propriétés suivantes :

- Il doit être inviolable, c'est-à-dire qu'il ne doit pas être possible de modifier le comportement d'un moniteur de référence lorsque le système est en fonctionnement.
- Il doit être toujours invoqué, c'est-à-dire qu'il ne doit pas être possible pour un sujet d'accéder à un objet du système sans que l'accès ne soit contrôlé par le moniteur de référence.
- Il doit être vérifié correct, c'est-à-dire qu'il doit être suffisamment simple pour être analysé et vérifié.

En outre, le contrôle d'accès implémenté dans ce moniteur de référence peut prendre plusieurs formes. Tout d'abord on retrouve les mécanismes de *filtrage* dont le fonctionnement peut se rapprocher de celui d'un pare-feu réseau (à états ou non). Par ailleurs, il existe des mécanismes de *contrôle de flux* dont le but est de suivre la manipulation et le transit de l'information à travers la *teinte* des données issues des objets ou le suivi de leur utilisation. L'état de l'art des mécanismes d'isolation pour l'environnement Android est détaillé dans la section suivante.

2.2 Mécanismes de contrôle d'accès pour Android

Dans cette section, sont présentés les mécanismes d'isolation de l'état de l'art ciblant particulièrement l'environnement Android. Pour observer plus facilement les différences entre cette multitude de solutions, celles-ci sont classifiées en fonction du niveau de privilège du logiciel les hébergeant.

2.2.1 Mécanismes de niveau système

Tout d'abord, certains mécanismes de l'état de l'art prennent place dans les couches logicielles les plus privilégiées, et peuvent par ailleurs exploiter des mécanismes d'isolation fournis par le matériel. Plus précisément, l'ensemble des travaux déployant ce type de mécanisme d'isolation de « niveau système » sont souvent implémentés au sein d'un hyperviseur, bénéficiant ainsi d'un niveau de privilège supérieur à celui du noyau, et des mécanismes d'isolation matériels offerts par le mode d'exécution EL2 [1] [Averlant 2017a].

On peut distinguer deux types de solutions de niveau système :

- Celles s'attachant à l'isolation entre deux environnements d'usages différents, où plusieurs instances d'Android s'exécutent de manière concomitante sur le smartphone.

1. ou son équivalent `hyp` sur ARMv7-A

- Celles dédiées à un environnement n'étant composé que d'un seul système d'exploitation, et dont le but est de fournir des solutions de sécurité pour celui-ci.

2.2.1.1 Isolation entre deux environnements

Le premier type de solution est notamment référencé dans les travaux ciblant le cas du BYOD. Par exemple, dans les travaux de Lengyel et al. [Lengyel 2014], l'isolation entre plusieurs instances d'Android est assurée par l'hyperviseur *Xen*. Les différentes instances d'Android sont ainsi lancées dans des machines virtuelles, isolées les unes des autres. Il ne peut y avoir qu'une seule machine virtuelle active dans ce cas, car chacune nécessite l'utilisation des périphériques d'entrées/sorties du smartphone, qui sont virtualisées par l'hyperviseur. L'utilisateur est ensuite capable d'effectuer la transition entre les machines virtuelles à l'aide d'un bouton physique. En outre, des opérations d'introspection sont effectuées par l'hyperviseur sur les différentes instances d'Android lancées sous son contrôle. Ces introspections ont pour but, d'une part de vérifier l'intégrité des instances Android via l'analyse statique des pages mémoire des machines virtuelles, et la reconstruction des données de haut niveau qu'elles contiennent pour leur analyse et, d'autre part de vérifier l'exécution d'instructions ou routines sensibles via le détournement de l'exécution des OS Android guests dans l'hyperviseur. L'intégrité de l'hyperviseur est par ailleurs vérifiée par un logiciel s'exécutant dans *TrustZone*, ainsi que par l'utilisation des *Xen Security Modules*, et par la bonne configuration de l'*Input–Output Memory Management Unit* (IO-MMU).

Une autre approche à ce type de solution concerne l'exécution sécurisée et isolée de code critique. Les moyens mis en œuvre reposent à nouveau sur un *hyperviseur bare-metal* [Cho 2016] ou sur un logiciel s'exécutant dans *TrustZone* [Sun 2015]. Dans les deux cas, le code critique n'est en aucun cas accessible par le système Android, mais peut cependant communiquer avec celui-ci par le biais d'APIs spécifiques.

2.2.1.2 Sécurisation d'un système d'exploitation

Concernant le deuxième type de solution, l'hyperviseur est cette fois-ci utilisé, non pas pour héberger de multiples systèmes d'exploitation, mais pour contrôler l'exécution d'une seule instance d'Android de par sa position privilégiée. C'est par exemple le cas des travaux menés par Horsch et al. [Horsch 2015] qui présentent un hyperviseur bare-metal capable de tracer ou d'intercepter et contrôler le flux d'information en provenance du noyau ou de l'espace utilisateur. Pour ce faire, l'hyperviseur joue sur la configuration des droits d'accès des pages de la MMU associées à la mémoire accessible depuis le noyau ou l'espace utilisateur. Ainsi, il est capable de suivre à la trace, de manière assez fidèle, le flux d'exécution des logiciels moins privilégiés, et peut de cette manière intercepter les flux non autorisés.

Un autre exemple d'utilisation d'hyperviseur est employé par Shen et al.

[Shen 2018]. Celui-ci, appelé TinyVisor, propose la mise en place d'une interception des appels systèmes du noyau Linux (avant et après leur exécution) afin de protéger l'intégrité et la confidentialité des transactions du Binder. Là encore cette interception repose sur l'instrumentation dynamique du système d'exploitation Android virtualisé.

2.2.1.3 Synthèse

On peut observer que les capacités des solutions employant des mécanismes d'isolation de niveau système sont très vastes, de part leur position privilégiée, et peuvent très bien être utilisées pour mettre en œuvre des politiques de contrôle d'accès telles que le montrent les exemples cités dans les paragraphes précédents. Cependant, pour arriver à ce résultat, les solutions de l'état de l'art nécessitent de procéder à une introspection, plus ou moins poussée, du logiciel s'exécutant sous son contrôle. Cette introspection, aussi appelée VMI pour *Virtual Machine Introspection*, est indispensable pour obtenir des informations de plus haut niveau d'abstraction nécessaires à l'application des contrôles d'accès.

On peut aussi noter que la mise en place de ce type de solutions requiert le concours du constructeur du smartphone. Un utilisateur n'a, en effet, pas la capacité de modifier ou installer du logiciel aussi privilégié sur son appareil.

En outre, l'utilisation d'hyperviseur à travers les extensions de virtualisation matérielles offertes par l'architecture ARM permet d'avoir un contrôle accru pour contrer les menaces en provenance du matériel lui-même (attaques DMA...) par la configuration des unités matérielles les plus privilégiées. Enfin, on peut aussi observer que les travaux présentés dans ces paragraphes font souvent usage, de manière additionnelle, des extensions de sécurité pour effectuer des vérifications d'intégrité supplémentaires sur l'hyperviseur, ou le reste du système.

2.2.2 Mécanismes de niveau OS

Un deuxième ensemble de mécanismes repose quant à lui sur du logiciel hébergé dans le noyau Linux. Ces solutions d'isolation peuvent avoir des buts divers concernant la sécurité de l'espace utilisateur sous-jacent.

2.2.2.1 Isolation entre deux environnements

On retrouve par exemple des travaux similaires à ceux de Lengyel et al. [Lengyel 2014], mais souhaitant apporter une approche comblant certains des problèmes inhérents à l'utilisation d'un hyperviseur. En effet, maintenir plusieurs environnements Android complets n'est pas sans impact sur un smartphone possédant des ressources limitées, tant en termes de stockage que de mémoire vive. Pour réduire cet impact, la solution Condroid [Xu 2015] utilise un unique noyau Linux capable d'exécuter plusieurs « conteneurs » isolés, chacun incluant les logiciels de l'espace utilisateur Android. L'isolation repose sur le concept de *namespace* implémentés par une fonctionnalité du noyau Linux appelée cgroup (*Control Groups*).

À travers ce mécanisme, le noyau effectue une séparation complète des processus appartenant à des namespaces différents. Cependant, il est possible de partager certaines ressources entre les namespaces, c'est une fonctionnalité qu'utilise Condroid pour mutualiser la partition système entre les conteneurs pour réduire l'espace nécessaire au stockage. De plus, Condroid permet le partage, de manière configurable, de certains services système Android dans le but de réduire la consommation globale de mémoire vive. En outre, cgroup permet aussi de définir des quotas d'utilisation des ressources hardware (CPU, RAM ...) pour les conteneurs. Enfin, certaines modifications du noyau et du framework Android ont été effectuées afin de rendre possible le partage des ressources matérielles du smartphone, telles que les périphériques d'entrées/sorties.

Une approche similaire, reprenant l'utilisation de cgroup a été employé dans AirBag [Wu 2014]. Le but de cette solution n'est cependant pas de définir des conteneurs correspondant à des usages d'Android différents comme Condroid, mais il est question d'offrir la possibilité à l'utilisateur de tester des applications dont il n'a pas confiance dans un environnement isolé. Toute modification de cet environnement, ou attaques potentielles contre celui-ci, ne seraient ainsi pas en mesure d'impacter le reste du système. Là encore des modifications du noyau ont été nécessaires pour le partage des ressources matérielles entre les conteneurs. De plus, le stockage utilisé par les conteneurs de « test » repose sur un système de fichiers *unionfs*, ne stockant que les différences avec le système de fichiers original, permettant ainsi de ne pas à avoir à dupliquer ce dernier pour chaque conteneur et de pouvoir les réinitialiser très simplement.

Par ailleurs, une solution très similaire à Condroid et plus récente a été publiée par Huber et al. [Huber 2016]. Leurs travaux définissent des restrictions supplémentaires sur les capacités de chaque conteneur par la configuration des *Linux capabilities*, et proposent une API de gestion des conteneurs dont les communications sont surveillées afin d'éviter toute compromission.

2.2.2.2 Interception et médiation des communications

D'autres solutions se concentrent quant à elles sur la mise en œuvre de contrôle d'accès pour l'ensemble des communications de l'espace utilisateur d'une seule instance Android. On retrouve notamment SE Android [Smalley 2013], une solution visant à porter sur Android le système de contrôle d'accès *SELinux*. SELinux est un exemple de « module de sécurité » utilisé traditionnellement sur des environnements Linux pour implémenter des politiques de sécurité de type MAC. Pour ce faire, il repose sur une API spécifique du noyau, nommée *Linux Security Module* (LSM), définissant des *hooks* lui permettant de contrôler l'ensemble des communications des objets du noyau à différents endroits de leurs traitements par Linux. Smalley et al. ont donc dû adapter ce système existant aux spécificités d'Android, et notamment à son système d'IPC particulier, le Binder. Ce travail d'intégration a consisté à ajouter les hooks supplémentaires à la gestion de ce canal de communication, ainsi qu'à l'assignation d'étiquettes de sécurité sur ces objets pour que des règles puissent

leur être associées. En outre, certaines modifications du framework Android ont été nécessaires pour la prise en compte de ces changements. Par ailleurs, Google a choisi d'intégrer ces modifications à Android depuis la version 4.3. Cependant, SE Android définit aussi d'autres fonctionnalités qui n'ont elles pas été intégrées. C'est le cas d'un système de contrôle des *intents*, ainsi qu'une API de contrôle des objets en provenance du framework nommée *Middleware MAC* (MMAC).

Cette manière de contrôler les objets du framework a, par ailleurs, inspiré les travaux de Bugiel et al. [Bugiel 2013] qui a poussé le concept plus loin dans sa solution nommée FlaskDroid. Outre l'utilisation de SE Android pour le contrôle d'accès aux objets du noyau, des modifications importantes opérées dans le framework au niveau du *System Server* et des différents services système permettent de définir un contrôle d'accès à la granularité très fine sur les objets issus du framework. Un langage de définition de politique pour les objets du framework est présenté et s'appuie sur la capacité de SELinux à mettre en œuvre des règles de contrôle d'accès dynamiques basées sur des booléens représentant des états du système.

Certains travaux ont exploré la possibilité de fournir une interface logicielle exposant la capacité d'effectuer du contrôle d'accès sur les objets du noyau ou du framework, à la manière de ce que propose les LSM pour le noyau Linux. Deux travaux très similaires, *Android Security Modules* (ASM) [Heuser 2014] et *Android Security Framework* (ASF) [Backes 2014] ont ainsi vu le jour. Plus précisément, ASM permet à n'importe quelle application, possédant la permission appropriée, de définir des fonctions pouvant contrôler l'accès à n'importe quel objet du système (noyau ou framework), et de manière assez précise. Les détails de l'accès en question sont fournis à chacune des fonctions qui doivent ensuite déterminer si oui ou non l'accès est conforme à la politique de sécurité mise en œuvre par l'application. La fonction est aussi capable de modifier le contenu de la transaction de certains types d'accès. Concernant ASF, c'est cette fois-ci sous la forme de deux modules que les développeurs de politiques doivent fournir les contrôles d'accès : un pour les contrôles d'accès des objets du noyau, et un pour ceux du framework. Sans surprise, ces deux solutions font à nouveau appel aux LSM pour le contrôle d'accès dans Linux, et modifient de façon intensive le framework pour l'instanciation des différents hooks et l'étiquetage des objets.

2.2.2.3 Contrôle des flux d'information

Enfin, deux dernières solutions sont dédiées au contrôle des flux d'information d'un environnement Android. La première, nommée AndroBlare [Andriatsimandefitra 2015], construit à l'exécution un graphe de contrôle des flux d'information et compare celui-ci à un ensemble de patterns correspondant à des comportements malicieux. Les LSM sont à nouveau choisis pour implémenter l'interception des flux d'information du système, même si le niveau de détails ainsi obtenu n'est pas aussi riche de part l'absence de connaissance sur les données abstraites du framework encapsulées dans ces communications.

La deuxième solution, nommée Weir [Nadkarni 2016], permet à chaque appli-

cation qui le souhaite (et qui utilise donc son API) de définir un contrôle d'accès multi-niveau sur les objets lui appartenant. Chacun de ses objets peut ainsi être associé à un niveau de confidentialité, et chaque application peut être lancée dans plusieurs instances de différents niveaux de criticité. Le respect de cette politique est ensuite assuré par un système dit de *contrôle de flux d'information décentralisé* (DIFC), qui est implémenté à nouveau à l'aide des LSM, et de la modification du framework (Zygote et AMS).

2.2.2.4 Synthèse

Pour résumer, on peut observer que la majorité des solutions présentées dans les paragraphes précédents font appel aux LSM pour le contrôle des accès relatifs au noyau. Il est aussi possible d'utiliser les cgroups pour forcer l'isolation de certains composants du système ou de sous systèmes, et ce de manière à impacter plus faiblement l'utilisation de ressources vis-à-vis des solutions basées sur un hyperviseur (on parle d'isolation « légère »). Enfin, les travaux présentés font aussi très souvent appel à des modifications du framework, soit pour obtenir un contrôle plus fin sur les objets de celui-ci, soit tout simplement pour assurer la compatibilité des mécanismes du noyau. En contrepartie, les menaces ciblant directement le noyau ne peuvent être complètement exclues par ce type de solutions, car une compromission du noyau les rendraient possiblement inopérantes. Cette remarque est d'autant plus importante que de nombreuses solutions présentées reposent sur d'importantes modifications du noyau dont les impacts sont difficilement auditables.

2.2.3 Mécanismes de niveau framework

Un troisième ensemble de travaux ciblent la capacité de fournir des mécanismes de contrôle d'accès en s'intégrant directement dans le framework Android.

2.2.3.1 Services système interchangeables

C'est le cas de PINPOINT [Ratazzi 2015], solution fournissant la capacité de changer dynamiquement les services système accessibles par une application de manière transparente pour celle-ci. Plus précisément, PINPOINT propose à l'utilisateur de choisir si une application a la possibilité d'accéder à chaque service système correspondant aux permissions qu'il possède, ou à un équivalent plus respectueux de la vie privée. Par exemple, le service de localisation peut être remplacé par un *stub* appliquant un bruit sur la position exacte du smartphone. Ainsi, les applications peuvent continuer à s'exécuter normalement, et l'utilisateur peut adapter le niveau des informations privées leur étant disponibles. Pour ce faire, PINPOINT s'appuie sur la modification du *ServiceManager* et le développement de *stubs* pour les services système les plus sensibles. Ensuite, lorsqu'une application requiert la communication avec un service système, le *ServiceManager* renvoie la référence du *Binder Node* correspondant au service approprié ou à son *stub* en fonction des propriétés de vie privée définies par l'utilisateur. Des mesures supplémentaires ont été

prises pour que les applications ne puissent pas se transmettre les références aux *Binder Nodes* des services système entre elles, les obligeant à passer par *ServiceManager* pour récupérer celles-ci. Cette interdiction est implémentée par l'ajout d'un traitement à SELinux dans le noyau.

2.2.3.2 Interception des intents

De manière plus générique, un autre type de communication peut être spécifiquement contrôlé depuis le framework. Il s'agit des intents émis par les applications et qui transitent exclusivement par AMS (*Activity Manager Service*). Yagemann et al. ont notamment choisi de s'intéresser à ces communications dans leurs travaux [Yagemann 2016]. Plus précisément, le contrôle d'accès opéré sur les intents est rendu possible par l'utilisation d'un outil existant nommé *Intent Firewall*. Cet outil, intégré à Android depuis la version 4.3, est appelé par AMS pour chaque intent qu'il doit traiter. Les fonctions de cet outil ont à leur disposition les informations complètes de chaque intent, telles que l'émetteur, le récepteur [2], et les détails de l'intent.

2.2.3.3 Interception des appels systèmes

Une autre approche a été préférée pour la conception de DeepDroid [Wang 2015]. En effet, celui-ci utilise un unique service système, s'exécutant en tant que root, pour instrumenter dynamiquement le reste du framework, ne nécessitant ainsi que de très peu de modifications pour être intégré dans un système Android. Cette instrumentation dynamique est effectuée à l'aide de *ptrace* qui est un appel système permettant au processus appelant de contrôler un autre processus, et notamment d'avoir accès à sa mémoire. Ces modifications permettent à DeepDroid d'intercepter et de contrôler les appels systèmes utilisés par les applications en traçant automatiquement ces dernières. Le détail des intents est ensuite reconstruit à partir de l'interception des appels ioctl ciblant le périphérique binder.

2.2.3.4 Instrumentation de l'environnement d'exécution Java

Enfin un dernier groupe de solutions d'isolation repose sur l'instrumentation de l'environnement d'exécution Java. Le plus connu, nommé TaintDroid [Enck 2014], a pour but de définir un système dynamique de surveillance et d'analyse des informations échangées entre les applications par un procédé de teinte. Toutes les variables sensibles des objets des applications sont ainsi teintées (on parle de *taint source*), et la solution doit ensuite s'assurer que celle-ci ne sera pas divulguée hors du smartphone, par exemple à travers l'interface réseau (*taint sink*). Le système de teinte est ainsi implémenté dans l'environnement d'exécution Java et la teinte est ensuite propagée dans les communications du binder lorsque la variable teintée est envoyée à une autre application.

2. Les fonctions sont invoquées après la résolution du destinataire pour les intents implicites.

Des travaux plus récents [Sun 2016, Backes 2017] ont montré qu'il était possible de porter le concept à l'environnement d'exécution Java actuel [3]. Cependant, un des problèmes de ces solutions est leur incapacité à intercepter les communications en provenance du code natif des applications, mais on peut très bien imaginer les coupler avec d'autres mécanismes dans ce but.

2.2.3.5 Synthèse

En résumé, de nombreuses solutions existantes utilisent des mécanismes de contrôle d'accès hébergés dans le framework. Cette proximité avec les objets ainsi isolés permettent de bénéficier d'informations plus précises sur ces objets, ou à moindre coup (en comparaison avec l'introspection nécessaire aux niveaux de privilèges plus élevés). Cependant, les accès visant les objets du noyau ne peuvent être contrôlés sans la concurrence de modification du noyau, ou de l'utilisation d'outils mis à disposition de l'espace utilisateur par celui-ci (*ptrace* par exemple).

2.2.4 Mécanismes de niveau applicatif

Cette dernière partie s'attache aux mécanismes d'isolation de l'état de l'art implémentés en tant que « simples » applications Android. Ces mécanismes ne sont pas à confondre avec les *Inline Reference Monitor* (IRM) qui consistent à insérer le code du moniteur de sécurité dans l'application devant être contrôlée. Ici le moniteur de sécurité consiste en une application spécifique, qui se retrouve en conséquence naturellement isolée des autres applications du système grâce aux propriétés de l'environnement Android.

La première solution, nommée Dr. Android and Mr Hide [Jeon 2012], implémente ce type de solution en modifiant les applications devant être contrôlées avant leur installation. Ces modifications consistent à : d'une part supprimer l'ensemble des permissions déclarées par l'application dans son manifeste, en les remplaçant par des permissions déclarées par l'application de monitoring ; d'autre part modifier le bytecode de l'application pour remplacer les appels aux services du framework, par des appels à des services « proxy » implémentés dans l'application de monitoring. Ainsi l'ensemble des appels aux services du framework peuvent être contrôlés par l'application de monitoring. Cependant, les communications de plus bas niveau ne peuvent être interceptées de cette manière.

Deux autres solutions de l'état de l'art règlent ce problème via une approche plus sophistiquée. La première, nommée Boxify [Backes 2015], consiste à nouveau en une application de monitoring, qui contrairement à la solution précédente va charger le code des applications cibles au sein de l'un de ses processus. Pour éviter que le code ainsi chargé puisse effectuer de lui même les communications (intents ou appels systèmes), un type de processus déprivilégié est spécifiquement choisi. Ces processus spéciaux sont habituellement utilisés par les navigateurs Internet dans un contexte de sandboxing. Ils sont en effet lancés sous un UID différent et ne possèdent

3. TaintDroid a été conçu pour le prédécesseur de ART nommé *Dalvik VM* (DVM).

pas la capacité d'effectuer des appels systèmes. Ces derniers sont redirigés vers l'application de monitoring qui servira ainsi de proxy à ces appels systèmes, tout en contrôlant leur adéquation avec les règles de la politique de sécurité. La deuxième solution, nommée NJAS [Bianchi 2015], possède une conception très proche tout en employant une technique d'isolation différente. Plus précisément, pour chaque application devant être contrôlée est générée une application *stub* adaptée à celle-ci. L'application *stub* charge ensuite le code de l'application dans un processus, et surveille celui-ci à l'aide de *ptrace*, interceptant de cette manière les appels systèmes effectués. L'inconvénient de cette solution est que les applications ainsi isolées peuvent se rendre compte qu'elles sont surveillées puisque le fait d'être tracées n'est pas totalement transparent.

Pour conclure, ce type de solution peut s'avérer intéressant car permettant de contrôler l'ensemble des communications d'une application. Cependant la logique de relayer les communications ajoute un overhead non négligeable. Un autre problème repose sur le fait qu'il n'y a pas de barrières empêchant un utilisateur de lancer directement l'application, en dehors de la surveillance des applications de monitoring.

2.3 Politiques d'autorisation pour Android

Après avoir identifié les principaux mécanismes techniques permettant de mettre en place du contrôle d'accès dans un environnement Android, il est important d'étudier les différents types de politique d'autorisation dans lesquelles ceux-ci peuvent prendre place. Le but étant : d'une part, de déterminer les caractéristiques des politiques d'autorisation existantes de la littérature ; et d'autre part, d'identifier les solutions techniques répondant aux besoins de contrôle d'accès de ces politiques. Mais avant de rentrer dans les détails, il est nécessaire de spécifier les termes d'une politique d'autorisation appliquée à un environnement Android.

2.3.1 Considérations relatives à l'environnement Android

Comme défini précédemment, une politique d'autorisation est constituée de propriétés de sécurité à satisfaire, et d'un ensemble de règles à faire respecter formant le schéma d'autorisation. La spécification de ces contraintes s'appuie cependant sur la notion de sujets, d'objets, et d'administrateurs de la politique, qu'il est intéressant de spécifier pour l'environnement Android. Par exemple, dans un système d'information classique, les sujets considérés sont majoritairement les utilisateurs du système. Cet état de fait ne peut être appliqué au monde d'Android, ne possédant en très grande majorité qu'un seul utilisateur physique.

La notion de sujet dans un environnement Android est intimement lié aux applications en elles-même. De manière générale, une application est la plus petite entité pouvant initier des communications avec le système. Ainsi, le système de permissions existant reste au niveau de granularité de l'application. Il est en effet

impossible pour le système de déterminer plus finement l'origine d'une communication. Il existe cependant des travaux considérant les bibliothèques tierces utilisées par une application comme sujet [Seo 2016, Fu 2017], mais nous considérons cet aspect hors de propos pour nos travaux car l'environnement Android ne propose pas nativement de moyens d'isoler les bibliothèques des applications les utilisant.

Un autre cas particulier repose sur l'utilisation d'un même appareil par plusieurs utilisateurs physiques. Ce cas d'utilisation est très rare pour un smartphone mais peut apparaître pour l'utilisation d'une tablette qui serait partagée par les membres d'une même famille par exemple. De manière similaire, on peut considérer que pour l'utilisation d'un smartphone dans un contexte professionnel, un même utilisateur physique peut être discriminé en deux utilisateurs « logiques » différents associés à l'utilisation personnelle ou professionnelle de l'appareil. Le sujet peut ainsi être défini comme le couple utilisateur/application.

Concernant la notion d'objet dans un environnement Android, on peut en distinguer de deux catégories, les objets du framework et ceux du noyau. Les objets du framework désignent l'ensemble des fonctions ou données accessibles depuis les APIs du framework, c'est-à-dire les données exposées par les applications (système ou utilisateur) ainsi que les services système, ou plus précisément :

- Les activités des applications installées.
- Les fonctions exposées par les *services* (système ou des applications).
- les données exposées par les *content providers.*
- Les évènements associés aux *broadcast receiver.*
- Les données (structurées ou non) échangées via le binder.

En outre, dans la littérature, ces objets sont souvent regroupés par permissions Android leur étant associées, ou bien directement par applications d'origine.

Les objets du noyau désignent, quant à eux, l'ensemble des objets directement gérés par le noyau auxquels les applications peuvent accéder. Ces accès peuvent être effectués directement via des appels systèmes ou indirectement par l'utilisation des APIs Android qui effectueront ces appels systèmes. Il s'agit donc par exemple du contenu des fichiers et de leurs méta-données, des sockets (INET, UNIX ...), des zones de mémoire partagées (ashmem), ou tout autre objet du noyau exposé par les appels systèmes.

Une autre facette des politiques de sécurité réside en la personne (physique ou morale) chargée de définir les termes de celles-ci et de les administrer. Là encore, dans le cas d'un environnement Android, la situation est différente du cas que représente un appareil informatique classique appartenant à une organisation. Outre le cas d'usage différent, une autre spécificité vient de l'existence d'une multitude d'acteurs impliqués dans la conception d'un smartphone, qui vont potentiellement faire partie des concepteurs de cette politique. En effet, un niveau de sécurité minimum est en droit d'être attendu par les utilisateurs d'un smartphone en tant que service basique associé à l'appareil. Pour chaque type de politique d'autorisation, il est important de prendre en compte lequel ou lesquels de ces acteurs sont désignés comme administrateurs de la politique, qu'ils soient :

- Le constructeur du smartphone [4].
- Le responsable de la sécurité d’une entreprise dans le cas du BYOD.
- L’utilisateur final du smartphone.
- Le développeur d’une application souhaitant définir des propriétés de sécurité supplémentaires sur les objets générés par son application.

Dans les paragraphes suivants, sont présentés les politiques d’autorisation de l’état de l’art. Elles seront abordées en mentionnant leur couverture en termes d’objets du système, ainsi que le ou les administrateurs considérés.

2.3.2 Politiques d’autorisation de la littérature

La notoriété d’Android a incité de nombreuses personnes à s’intéresser à la création de politiques de sécurité pour cette plateforme dans le but de renforcer sa sécurité ou de fournir des outils supplémentaires à la préservation de la vie privée de ses utilisateurs. Les exemples les plus notables sont SE Android [Smalley 2013] définissant un schéma d’autorisation de type MAC pour les objets du noyau, et MPdroid [Guo 2013] qui ajoute à un schéma d’autorisation similaire une politique d’autorisation RBAC ciblant les objets du framework. Nous nous intéressons cependant plus particulièrement aux politiques d’autorisation prenant en compte la notion de contexte (CAAC), qui représentent les travaux les plus proches de ce que nous proposons dans cette thèse.

2.3.2.1 Politiques d’autorisation ABAC

ConUCON [Bai 2010] est le premier article mentionnant la notion de contexte dans le cadre d’une politique d’autorisation. Cette dernière s’appuie sur un modèle nommé UCON (*Usage Control*) implémentant une politique d’autorisation de type ABAC qui prône un contrôle d’accès continu, c’est-à-dire définissant des règles à appliquer avant, pendant et après chaque accès. Le modèle est augmenté pour prendre en compte la notion de contexte dans ses règles. Plus précisément, sont considérés les informations temporelles (date et heure) et spatiales (géolocalisation) issues des différents capteurs du smartphone, ainsi que de l’état global de ce dernier (niveau de batterie, écran allumé...). La politique est représentée sous forme de fichiers XML, et est administrée par l’utilisateur via une application dédiée. Les objets protégés par la politique sont les fichiers du système, ainsi que les services et ressources des applications.

D’autres travaux implémentent aussi une politique d’autorisation ABAC. On peut par exemple citer Crêpe (*Context-Related Policy Enforcement*) [Conti 2012] et SecureDroid [Arena 2013] qui possèdent toutes les deux une architecture basée sur le modèle XACML. Dans Crêpe, les droits d’accès entre les sujets (applications) et les objets (applications, ressources système du framework et accès réseaux) sont stockés dans une matrice qui est accompagnée d’un ensemble de vecteurs stockant

4. Dans cette catégorie est aussi inclus le concepteur du SoC et l’opérateur de téléphonie mobile.

les règles obligatoires associées directement aux objets. Le modèle XACML a cependant été modifié pour permettre l'ajout de conditions spatiales (géolocalisation) ou temporelles (date/heure) aux règles de contrôle d'accès. La politique est administrée par le responsable de l'entreprise (cas BYOD) ou directement par l'utilisateur via des fichiers XML ou Ponder [5].

SecureDroid définit quant à lui une politique de contrôle des accès aux objets du framework dont la granularité se limite aux permissions associées à ces objets. Dans cette optique, il utilise un sous ensemble du modèle XACML pour définir les règles de son schéma d'autorisation. La particularité de ces travaux repose sur la capacité de la politique à être configurée par plusieurs parties. En effet, plusieurs administrateurs sont capables de définir des politiques d'autorisation complémentaires de plus ou moins haute priorité. Ainsi, le constructeur du smartphone pourra définir la politique possédant un plus haut niveau de priorité que respectivement celle définie par : un opérateur, une entreprise ou un utilisateur.

2.3.2.2 Politiques d'autorisation RBAC

La notion de contexte a aussi été utilisée conjointement à des schémas d'autorisation RBAC dans DR-BACA (*Dynamic RBAC for Android*) [Rohrer 2013] et CA-ARBAC (*Context-Aware Android RBAC*) [Abdella 2017]. Dans le premier cas, la solution de Roher et al. a pour but de définir des droits d'accès associés à différents profils d'utilisateurs physiques dans un contexte de BYOD. RBAC est donc utilisé de manière assez traditionnelle dans le sens où les rôles sont associés aux utilisateurs physiques et non pas aux applications. Les règles déterminent, pour un rôle donné, les droits d'accès des applications aux ressources associées aux permissions Android. Elles permettent par ailleurs d'interdire ou non le lancement d'une application, ou de restreindre globalement l'accès aux objets du framework. Chacune de ces règles est liée à des contraintes dynamiques définissant si celle-ci est active ou non en fonction du contexte spatial, temporel, ou de l'occurrence de certains évènements système (e.g. activation du Wi-Fi, ...). L'administrateur peut distribuer les paramètres de la politique via un serveur de configuration. Il est cependant possible pour les utilisateurs de créer des règles additionnelles, et de les partager via le NFC.

Pour le deuxième cas, CA-ARBAC utilise différemment la politique RBAC en affectant les rôles directement aux applications installées. Les rôles correspondent ainsi aux catégories fonctionnelles auxquelles sont rattachées les applications. Le bénéfice recherché est de réduire le nombre de règles effectives, diminuant par la même occasion le poids de la configuration de celles-ci. Les règles peuvent être statiques, i.e. toujours actives, ou dynamiques. Pour ces dernières, leur activation dépend du contexte spatial et temporel, ainsi que de l'état du smartphone (statut de la batterie, de l'écran, en cours d'appel). L'utilisateur est amené à configurer les rôles et les règles associées via une application dédiée.

5. `http://ponder2.net/`

2.3.2.3 Politiques basées sur des modèles d'apprentissage

Une autre catégorie de travaux est consacrée à l'utilisation de modèles d'apprentissage « machine learning » pour la génération automatisée des règles de la politique d'autorisation dans le but d'améliorer la sécurité ou la vie privée des utilisateurs n'ayant aucune connaissance technique. C'est le cas de ConXsense [Miettinen 2014] qui utilise le machine learning pour déterminer le niveau de risque d'atteinte à la vie privée à partir de données de contextes issues des capteurs matériels du smartphone (GPS, Bluetooth, heure). En fonction de ce niveau de risque, des règles supplémentaires ou plus strictes sont mises en place par la couche de contrôle d'accès de l'architecture. Cette couche est par ailleurs basée sur la solution FlaskDroid [Bugiel 2013] (voir §2.2.2.2) qui implémente plusieurs types de politique d'autorisation. En effet, comme tous travaux basés sur SeLinux, plusieurs types de politiques d'autorisation de type MAC peuvent être employées (conjointement ou non) : 1) une politique multiniveau (ou MLS pour *Multi-Level Security*) similaire à celle de Bell et Lapadula ; 2) une politique basée sur les types (ou TE pour *Type Enforcement*), dont le fonctionnement repose sur la déclaration de domaines associés aux sujets, de types associés aux objets, et de règles définissant des droits d'accès entre les domaines et les types [6] ; 3) une politique RBAC (moins utilisé). L'ensemble de ces règles peuvent ensuite être administrées par le constructeur (politique au niveau système), les développeurs des applications (politique protégeant les objets de l'application), ou l'utilisateur.

Une deuxième solution nommée SmarPer [Olejnik 2017] est plus orientée vers la protection de la vie privée de ses utilisateurs, en incluant la possibilité d'obfusquer les données des objets en plus de la simple interdiction ou autorisation d'accès. Par exemple, un bruit est appliqué lors de l'obfuscation des données de géolocalisation. En outre, le machine learning sert ici à prédire les décisions des utilisateurs (autorisation/obfuscation/interdiction) en fonction du contexte courant. Celui-ci est plus fourni que pour les précédents travaux. Il considère en effet : les informations concernant l'application ayant déclenchée l'accès (type d'application, premier/arrière plan ...) ainsi que celle étant au premier plan (si différente) ; les informations relatives à l'accès en question (méthode de l'API invoquée ...) ; en plus du contexte spatial et temporel classique et de l'état du smartphone (verrouillé ou non, type de connectivité, niveau de batterie, ...).

2.3.2.4 Synthèse

Malgré les différents cas d'usage des travaux mentionnés ci-dessus, on peut observer que le contexte considéré se restreint aux données environnementales (spatiales et temporelles) issues des capteurs du smartphones, ainsi qu'aux informations sur l'état global du smartphone (niveau de batterie, écran éteint, périphériques de communications actifs ...). Seul SmarPer pousse plus loin cette définition en prenant

6. Fonctionnement similaire à l'implémentation d'une politique ABAC ne possédant qu'un seul attribut par sujet et objet.

en compte certaines informations relatives aux applications en cours d'exécution. Par ailleurs, on note un effort croissant quant à l'adoption de ces solutions par les utilisateurs n'ayant a priori aucune expertise dans le domaine, notamment à travers l'emploi de machine learning. Concernant les choix techniques, le tableau 2.1 synthétise les politiques d'autorisation mentionnées ci-dessus en précisant les mécanismes de contrôle d'accès utilisés.

Nom de la solution	**Type de shéma d'autorisation**	**Mécanismes de contrôle d'accès**	
		Framework	Noyau
ConUCON [Bai 2010]	ABAC (UCON)	Hooks AMS + PM	
Crêpe [Conti 2012]	ABAC (XACML)	Hooks AMS + PM	Config IPtables
SecureDroid [Arena 2013]	ABAC (XACML)	Hooks AMS + PM	
DR-BACA [Rohrer 2013]	RBAC	Hooks AMS + PM	
CA-ARBAC [Abdella 2017]	RBAC	ASM [Heuser 2014]	
ConXsense [Miettinen 2014]	MAC (MLS + TE)	FlaskDroid [Bugiel 2013]	
SmarPer [Olejnik 2017]	MAC	Hooks DVM	

TABLE 2.1 – Synthèse des politiques d'autorisation de l'état de l'art

On remarque que la grande majorité des solutions de l'état de l'art se reposent uniquement sur des modifications du framework pour la mise en œuvre de leur politique d'autorisation. Ce constat est à mettre en parallèle avec le modèle d'attaque considéré par celles-ci. En effet, certains travaux cités ne considèrent que des attaquants simplement curieux qui n'utiliseraient que les API Java officielles pour interagir avec le reste du système. L'inverse nécessiterait effectivement la collaboration avec des mécanismes de niveau OS (§1.2.2).

2.4 Conclusion et démarche scientifique

Dans cette thèse nous nous intéressons tout particulièrement à deux aspects : d'une part aux politiques d'autorisation d'Android, et d'autre part à leur implémentation à travers des mécanismes d'isolation.

Pour le premier point, nous avons souhaité orienter notre recherche sur une politique d'autorisation exploitant de manière plus poussée le contexte du smart-

phone, et plus précisément celui des applications en cours d'exécution. En effet, cette considération spécifique, qui est insuffisamment prise en compte dans les travaux de l'état de l'art, permet d'apporter de nouveaux outils aptes à contrer un ensemble de menaces émergentes. Nous avons par ailleurs prêté une attention particulière à l'ergonomie de notre solution afin de la rendre acceptable aux yeux d'utilisateurs quelconques étant curieux de sécurité mais ne possédant pas de pré requis particuliers.

Pour le second point, nous avons désiré élaborer une architecture de sécurité implémentant les mécanismes d'isolation nécessaires à la mise en place de politiques d'autorisation. Comme l'étude de l'état de l'art le montre, de nombreuses solutions ont déjà été proposées dans la littérature. Cependant, nous souhaitions que notre architecture soit à la fois capable de : 1) répondre aux attaques ciblant les logiciels les plus privilégiés, ou directement le matériel ; 2) rester proche des données isolées, afin d'éviter la problématique du gap sémantique. Pour cela, notre contribution s'appuie sur une architecture multi-niveau novatrice, répondant à ces besoins.

Ainsi, l'organisation des prochains chapitres est la suivante. Le chapitre 3 est consacré à la présentation de notre première contribution, une politique de sécurité sensible au contexte d'exécution. Le chapitre 4 est quant à lui dédié à une architecture de sécurité mettant en œuvre le contrôle d'accès nécessaire à notre politique, et qui constitue notre deuxième contribution. Enfin, le chapitre 5 présente un prototype d'implémentation de notre architecture de sécurité ainsi qu'une validation expérimentale de ce prototype.

Chapitre 3

Politique de sécurité pour Android sensible au contexte

Sommaire

3.1 Présentation, objectifs et modèle d'attaque 44
3.1.1 Objectifs de la solution 44
3.1.2 Objets considérés par la politique 44
3.1.3 Types de règles de contrôle d'accès 46
3.1.4 Contexte nécessaire à l'application des règles 47
3.1.5 Modèle et scénarios d'attaque 47
3.2 Format, exemples et distribution 48
3.2.1 Format des règles de sécurité : le *Secure Manifest* 48
3.2.2 Exemples d'utilisation 50
3.2.3 Distribution des *Secure Manifest* 52
3.3 Les conflits et leur gestion 53
3.3.1 Définition et objectifs de la gestion des conflits 53
3.3.2 Aspects pratiques 54
3.3.3 Aspects ergonomiques 55
3.3.4 Améliorations possibles 56
3.4 Encourager l'utilisateur à paramétrer la politique 57
3.5 Conclusion 59

Ce chapitre est dédié à la présentation de notre proposition de politique de sécurité adaptée à l'environnement Android [Averlant 2018]. Il est constitué de trois parties. La première partie récapitule les objectifs de cette politique de sécurité tout en les mettant en phase avec le modèle d'attaque considéré. Une deuxième partie détaille la politique, et plus particulièrement le format de ses règles, leur distribution, ainsi que des exemples pour illustrer différents cas d'utilisation. Une dernière partie énonce les moyens mis en place pour la gestion des conflits, et les considérations particulières quant au cycle de vie des applications. Enfin, une dernière partie propose une approche pour encourager les utilisateurs à adopter une politique de ce type.

3.1 Présentation, objectifs et modèle d'attaque

3.1.1 Objectifs de la solution

Nos travaux se positionnent comme une extension au système de permissions Android. Le but étant de combler le manque d'expressivité de ce dernier en le complétant par des règles dynamiques. En effet, une des problématiques du système de permissions Android réside dans le caractère statique de ses règles conditionnant l'accès aux ressources du système. Pour rappel, celles-ci sont accordées par l'utilisateur, soit dès l'installation, soit une unique fois avant la première tentative d'accès à la ressource, cet accord étant modifiable ensuite dans les paramètres système. Cependant, il n'existe aucune manière de conditionner ces accès, c'est-à-dire de définir des conditions pour lesquelles les accès seraient interdits ou autorisés dynamiquement. Cette situation est renforcée par le fait qu'il est impossible pour un utilisateur de définir le niveau de confiance qu'il possède en chaque application installée.

En outre, les problèmes sus-mentionnés sont exacerbés par leur inefficacité à traiter certaines menaces émergentes. En particulier, les attaques basées sur l'utilisation concurrente d'un périphérique par plusieurs applications. Ce type d'attaques permet par exemple de détourner des communications BLE [Sivakumaran 2018] ou Bluetooth [Naveed 2014]. Il serait en effet possible de mitiger ce genre d'attaque en définissant des règles dynamiques interdisant l'utilisation simultanée de ces périphériques. Par ailleurs, certaines permissions augmentent significativement les capacités des applications auxquelles elles sont octroyées en leur permettant par exemple :

- D'effectuer des actions à la place de l'utilisateur
- D'intercepter les entrées de l'utilisateur
- De s'afficher en superposition à d'autres applications
- D'intercepter les communications réseau de n'importe quelle application

Là encore, il serait intéressant pour l'utilisateur de pouvoir désactiver l'utilisation de ces permissions dynamiquement lors de l'exécution d'applications critiques, comme une application bancaire par exemple.

Le but de notre politique de sécurité est donc de répondre à ces menaces en complétant le système de permissions actuel, tout en étant assez facile d'utilisation pour pouvoir être adoptée par des utilisateurs non experts. Pour ce faire, nous avons implémenté deux types de contrôle d'accès restreignant l'exécution ou l'accès aux ressources système des applications, et ce en fonction du contexte d'exécution.

3.1.2 Objets considérés par la politique

Dans cette optique, nous avons regroupé les objets du système, qu'ils soient du framework ou du noyau, en ressources dont la dénomination est assez proche de celle des permissions. L'ensemble des ressources considérées est présenté dans le tableau 3.1. Ces ressources sont à leur tour rassemblées en groupes.

Cette classification permet de regrouper les ressources issues de fonctionnalités similaires afin, d'une part d'identifier plus aisément les risques de sécurité associés

Groupe	**Ressources**
Localisation	Position approximative, GPS
Périphériques	Appareil photo, Microphone, Capteurs de mouvement, Capteur environnementaux
Données personnelles	Contacts, Calendrier, SMS, logs des Appels, Infos d'identification (utilisateur et smartphone)
Stockage	Stockage interne, Carte SD, Stockage externe
Communication	Téléphonie (appels & SMS), Données mobiles, Wi-Fi, Bluetooth, NFC, Autres
Ressources à haut risque	Paramètres système, Affichage superposé, Services d'automatisation, VPN, Clavier

Table 3.1 – Ressources considérées

aux ressources, et d'autre part d'expliquer plus facilement ces risques aux utilisateurs.

Ainsi, les groupes *Localisation* et *Périphériques* s'attachent à regrouper les ressources issues des différents capteurs du smartphone dits « passifs », c'est-à-dire sans possibilité d'interaction avec l'environnement autre que l'écoute passive. On peut remarquer que les ressources de ces deux groupes sont particulièrement exposées aux risques relatifs à la confidentialité et à la vie privée. En effet, les données issues de ces ressources sont particulièrement sensibles, que ce soit directement ou indirectement car pouvant être utilisées pour inférer d'autres données. Par exemple, il est possible de déterminer l'adresse du lieu de travail d'un utilisateur en analysant ses déplacements au cours de la semaine.

Les ressources des groupes *Données personnelles* et *Stockage* sont également concernées par cette problématique de confidentialité et vie privée. Elles concernent en effet : d'une part l'ensemble des données personnelles de l'utilisateur, ainsi que les informations permettant d'identifier directement le smartphone ; et d'autre part les systèmes de fichiers accessibles par l'ensemble des applications possédant la permission adéquate. L'accès à ce type de ressources représente donc un risque relatif à la confidentialité (fuite d'informations, tracking ...), mais aussi une menace pour l'intégrité des applications propriétaires. Par exemple, un système de fichiers partagé peut être utilisé pour le stockage de fichiers de mise-à-jour, qui peuvent ainsi servir de vecteur d'attaque. De plus, un accès simultané à ces ressources de stockage peut ouvrir la porte à des attaques de type TOCTOU (*time of check to time of use*).

En outre, le groupe *Communication* contient l'ensemble des ressources permettant au smartphone de communiquer avec d'autres appareils. Ces ressources induisent automatiquement des risques de sécurité inhérents à leur fonctionnalité de réception et transmission d'informations. Plus précisément, elles représentent à la fois : 1) un vecteur d'attaque pouvant cibler les applications légitimes ou le système

directement (intégrité et disponibilité) ; 2) le moyen principal d'exfiltrer des informations personnelles (confidentialité). De plus, il est possible d'utiliser ces ressources pour attaquer les appareils à proximité, notamment via l'utilisation simultanée de ces ressources.

Le dernier groupe concerne les *Ressources à haut risque* regroupant les ressources donnant aux applications des capacités supplémentaires de contrôle sur le smartphone en lui même et ses interactions. Plus précisément, ces ressources fournissent aux applications la capacité de :

- Modifier les paramètres système, permettant par exemple l'altération de paramètres de sécurité.
- S'afficher en superposition (comme un *overlay*) aux autres applications, ce qui permet d'empêcher l'accès aux autres applications (ransomware), ou de se faire passer pour l'application sous-jacente (fuite de données).
- Automatiser les actions de l'utilisateur (services d'accessibilité, de remplissage automatique de formulaires, ...).
- Définir un service VPN permettant d'intercepter les communications IP des autres applications.
- définir une application jouant le rôle de clavier permettant de contrôler les saisies clavier de l'utilisateur.

L'utilisation de ces ressources a donc un impact relativement important sur la sécurité de l'utilisateur. Par ailleurs, ces ressources (ou plus précisément les permissions associées) nécessitent habituellement un accord explicite de l'utilisateur via un affichage dédié pour pouvoir être utilisées.

La liste des ressources de cette classification couvre de manière assez complète l'ensemble des objets du système. Cependant, il est possible de rajouter des ressources supplémentaires dans les groupes de notre classification en cas d'évolution de l'OS Android, ou de modification des objectifs de la politique de sécurité (e.g. contrôle à grain plus fin des ressources).

3.1.3 Types de règles de contrôle d'accès

Pour servir les objectifs de notre politique de sécurité, son schéma d'autorisation peut être composé de deux types de règles dynamiques contrôlant les accès aux ressources identifiées ci-avant. Chacune de ces règles est associée à une application d'appartenance, et n'est active que lorsque cette-dernière est en cours d'exécution. L'ensemble des règles actives est donc l'union des règles des applications en cours d'exécution.

Le premier type de règle a pour but d'interdire le lancement d'une ou plusieurs applications tant que l'application associée à la règle est en cours d'exécution. Le but de ce type de règle est d'introduire la notion de *niveau de criticité* des applications, qui n'est pas exprimable avec le modèle de permission classique d'Android. Ainsi, lorsqu'une application, jugée critique par l'utilisateur, est en cours d'exécution, il

est possible de restreindre l'utilisation d'applications jugées moins critiques. Ces dernières peuvent, en effet, être considérées comme une menace potentielle pour la sécurité de l'application en question.

Le second type de règle vise à restreindre la liste des ressources accessibles par une ou plusieurs applications tant que l'application associée à la règle est en cours d'exécution. Ces règles, moins restrictives que les précédentes, peuvent ainsi être utilisées pour interdire à d'autres applications l'accès aux ressources dont l'usage pourrait mettre en péril la sécurité de l'application associée ou de son utilisateur. Par ailleurs, nous avons vu que de nouvelles attaques reposent sur l'utilisation simultanée de périphériques de communication. Il est également envisageable de penser que l'accès simultané à n'importe quelle ressource correspondant à des données non volatiles puisse être utilisé pour mener des attaques de type TOCTOU. En ce sens, ce deuxième type de règles peut aussi être employé pour restreindre l'accès à une ressource donnée, mais uniquement lorsque qu'il s'agit d'un accès simultané entre l'application d'appartenance et d'autres applications. Plus précisément, lorsque cette dernière souhaite accéder à la ressource en question, aucune autre application n'est autorisée à faire de même.

En outre, il est possible d'associer ces règles à un composant de l'application d'appartenance afin d'être plus précis concernant les conditions d'activation des règles. Par exemple, des règles supplémentaires plus restrictives peuvent n'être activées que lors de l'utilisation d'un composant critique (activité, service ...) de l'application.

3.1.4 Contexte nécessaire à l'application des règles

Afin de définir les conditions d'activation de ces règles dynamiques, il est nécessaire de spécifier les états du système ou de l'environnement à surveiller, c'est-à-dire le contexte considéré. À partir de la description des règles qui précèdent, nous pouvons affirmer que le contexte minimum à considérer est composé de :

- La liste des applications en cours d'exécution
- La liste des composants actifs de chaque application
- La liste des ressources activement utilisées par les applications

Nous pouvons en effet déterminer, à partir de ces trois listes, l'ensemble des règles courantes du schéma d'autorisation devant être appliquées. Cependant des informations supplémentaires sont prises en compte pour la gestion des conflits (voir §3.3). Par ailleurs, il est possible d'enrichir ce contexte dans le futur à partir d'informations supplémentaires telles que des informations environnementales (spatiales et/ou temporelles) comme ce qui est fait pour les travaux de l'état de l'art.

3.1.5 Modèle et scénarios d'attaque

Concernant le modèle d'attaque de notre politique de sécurité, la cible envisagée est un utilisateur lambda de smartphone possédant un système d'exploitation

Android. Le matériel embarqué comprend un processeur principal d'architecture ARMv8-A ; le reste du matériel, tel que les différents périphériques inclus, importe peu. De plus, ce smartphone doit être équipé d'une version récente du système d'exploitation Android n'ayant pas subi de modifications [1] de la part de l'utilisateur. Ce dernier est cependant libre d'installer n'importe quel nombre d'applications, depuis le magasin par défaut, nommé *Play Store*, ou bien d'autres sources.

Les scénarios d'attaques considérés sont les suivants :

- L'utilisateur installe une application malveillante, c'est-à-dire dont l'attaquant possède le contrôle.
- L'attaquant exploite une vulnérabilité d'une application précédemment installée par l'utilisateur, et s'en sert en tant que proxy pour son attaque.
- L'attaquant interagit avec les périphériques de l'appareil pour exploiter une faille du système ou d'une application installée.

Les vecteurs d'attaques se limitent donc, d'une part aux applications installées sur le smartphone, et d'autre part aux périphériques embarqués. Dans le premier cas, nous apportons une réponse à travers l'utilisation de notre politique de sécurité ; alors que dans le deuxième cas, cette tâche incombe à l'architecture de sécurité employée pour la mettre en œuvre. À partir de ces vecteurs d'attaque, l'individu malveillant peut ensuite chercher à effectuer les actions suivantes :

M1 Compromettre la sécurité d'une application installée.

M2 Compromettre la sécurité des données personnelles de l'utilisateur.

M3 Compromettre la sécurité du système Android.

M4 Nuire à la sécurité des appareils auxquels le smartphone peut se connecter.

Pour les menaces **M1** et **M4**, nous nous intéressons tout particulièrement dans cette thèse aux attaques rendues possibles par l'exécution concurrente d'applications ou par l'utilisation concurrente de ressources ; le reste des attaques pouvant être évitées par une politique de sécurité classique employant un système de restriction statique telle que celles présentés dans l'état de l'art. Par ailleurs, il est important de garantir la sécurité du système Android, notre politique reposant sur son bon fonctionnement. Ainsi, la menace **M3** est considérée dans son entièreté.

Parmi les individus à caractère malveillant, les personnes curieuses peuvent aussi chercher à ébranler la vie-privée des utilisateurs (**M2**) en accédant à des données personnelles n'étant accessibles que lors de l'utilisation de certaines applications.

3.2 Format, exemples et distribution

3.2.1 Format des règles de sécurité : le *Secure Manifest*

Afin de définir les règles de sécurité mentionnées dans la section précédente, nous avons choisi d'utiliser un fichier XML, nommé *Secure Manifest*, associé à chaque application. La lecture de l'ensemble des *Secure Manifests* permet ensuite de constituer

1. Aucune modification de la partition système, excluant toute possibilité de *root* du système.

la liste des règles du schéma d'autorisation qui doivent être appliquées. Le listing 1 présente la sémantique de ces fichiers *Secure Manifest*.

```
<?xml version="1.0" encoding="utf-8"?>
<secure_manifest xmlns:secure="./secure.xsd">
    <!-- Informations d'identification de l'application -->
    <pkg_info name="com.target.app" pub_key="..." minversion="4.2" />
    <!-- Secure Context pour le composant "login_activity" -->
    <secure_context target="login_activity">
        <!-- Interdiction de l'exécution concurrente de app1 -->
        <app_restriction>
            <app_list>
                <app name="com.example.app1" />
            </app_list>
        </app_restriction>
        <!-- Interdiction globale de l'utilisation du nfc -->
        <resource_restriction>
            <resource name="nfc" />
        </resource_restriction>
        <!-- Restriction d'utilisation simultanée du wifi pour app2 -->
        <app_restriction>
            <app_list>
                <app name="com.example.app2" />
            </app_list>
            <resource_restriction>
                <resource name="wifi" concurrent="true" />
            </resource_restriction>
        </app_restriction>
    </secure_context>
</secure_manifest>
```

Listing 1 – Format du *Secure Manifest*

Le premier élément du *Secure Manifest*, `<pkg_info />`, permet d'identifier l'application pour laquelle le *Secure Manifest* est destiné. Plus précisément, la balise est constituée de deux attributs indiquant le nom du package de l'application en question et la clé publique de son développeur. Ces deux attributs permettent ainsi d'associer le *Secure Manifest* à l'application en question, ainsi que de vérifier la signature associée à celle-ci. Ce dernier point est notamment utile pour s'assurer que l'application n'a pas été modifiée par une tierce partie, par exemple si celle-ci est récupérée depuis un store non officiel. Par ailleurs, deux autres attributs optionnels permettent de préciser la version minimum ou maximum de l'application pour laquelle le *Secure Manifest* a été conçu. Par conséquent, plusieurs *Secure Manifests* peuvent exister pour une même application afin de pouvoir faire évoluer les règles en fonction des changements de l'application.

Le deuxième élément, `<secure_context>`, est dédié au recueil des règles dynamiques qui doivent être actives pour une « partie » de l'application. Plus précisément, l'attribut « target » désigne le ou les composants qui doivent être actifs

pour que les règles prennent effet. Si celui-ci n'est pas précisé, les règles sont actives pour tous les composants de l'application. De plus, il peut y avoir plusieurs `<secure_context>` ciblant différents composants. En outre, un même composant peut être la cible de plusieurs `<secure_context>`.

Ensuite, les règles associées aux `<secure_context>` sont spécifiées en tant que balises filles. Comme indiqué dans la section précédente, ces règles peuvent être de deux types :

- Les règles empêchant l'exécution concurrente d'applications, définies dans des balises `<app_restriction>`.
- Les règles restreignant l'utilisation de ressources, spécifiées dans des balises `<resource_restriction>`. Un attribut *concurrent* précise pour chaque `<resource/>` si la restriction concerne uniquement une utilisation simultanée.

Il est aussi possible de définir des règles du deuxième type mais dont le champ d'action est réduit à un certain nombre d'applications via la combinaison de ces deux balises (voir listing 1).

3.2.2 Exemples d'utilisation

Pour illustrer les capacités de notre politique, les paragraphes suivants présentent plusieurs exemples de *Secure Manifests* relatifs à différents cas d'utilisation.

Le premier exemple, présenté dans le listing 2, montre un cas d'utilisation où l'utilisateur souhaite préventivement interdire l'exécution des autres applications lors du lancement de son application bancaire. Cette mesure, certes un peu extrême, vise à prévenir l'exploitation de potentielles failles de sécurité par les autres applications du système. Cette option peut ainsi être choisie par des utilisateurs exigeants vis-à-vis de la sécurité de leurs applications les plus critiques, ou bien lorsque peu de documentation est disponible concernant les ressources employées par ces applications.

```xml
<?xml version="1.0" encoding="utf-8"?>
<secure_manifest xmlns:secure="./secure.xsd">
    <pkg_info name="com.banking.app" pub_key="..." />
    <secure_context>
        <app_restriction>
            <!-- Aucune app est autorisée à s'exécuter en parallèle -->
        </app_restriction>
    </secure_context>
</secure_manifest>
```

Listing 2 – Exemple d'exclusion totale

Le listing 3, présente quant à lui un exemple illustrant les bénéfices des règles empêchant l'utilisation de ressources. Plus précisément, le *Secure Manifest* associée à une application bancaire inclut une règle de ce type pour les ressources NFC,

services d'automatisation et affichage superposé. Ainsi, l'usage du NFC est formellement interdit pour les autres applications lorsque l'activité de paiement sans contact est active, ceci dans le but d'éviter toute compromission des transactions effectuées. De plus, les services d'automatisation et l'affichage superposé sont désactivé lors de l'exécution de l'application bancaire afin d'empêcher un quelconque vol ou manipulation de coordonnées bancaires ou d'autres données saisies par l'utilisateur via une attaque similaire à *Cloak and Dagger* [Fratantonio 2017].

```
<?xml version="1.0" encoding="utf-8"?>
<secure_manifest xmlns:secure="./secure.xsd">
    <pkg_info name="com.banking.app" pub_key="..." />
    <!-- Accès interdit au NFC lors du paiement sans contact -->
    <secure_context target="proceed_payment_activity">
        <resource_restriction>
            <resource name="nfc" />
        </resource_restriction>
    </secure_context>
    <!-- Blocage des services d'automatisation et affichage superposé -->
    <secure_context>
        <resource_restriction>
            <resource name="automation_services" />
            <resource name="overlay" />
        </resource_restriction>
    </secure_context>
</secure_manifest>
```

Listing 3 – Exemple de restrictions de ressources pour une application bancaire

Enfin, un utilisateur peut souhaiter interdire l'utilisation d'une ressource spécifiquement lorsque une application qui recueille des données sensibles pour la vie privée est active. Ainsi, le listing 4 présente un *Secure Manifest* dédié à une application de fitness qui évalue les performances sportives de son utilisateur en utilisant des données de localisation et les capteurs de mouvement du smartphone. Ces performances, ou toutes autres données associées [2], pourraient être inférées par une application peu soucieuse de la vie privée en observant les valeurs brutes issues de ces ressources. Ainsi, une exclusion de l'utilisation de ces ressources permet de conserver les propriétés de confidentialité attendues.

Ces exemples assez simples montrent ainsi quelques cas d'utilisation concrets pour les trois types de règles évoqués précédemment. Il est bien sûr possible de créer des *Secure Manifest* plus complets en multipliant le nombre de `<secure_context>` et de règles associées pour obtenir une politique de sécurité plus affinée pour chacune des applications.

2. E.g. les données de santé, le type d'effort sportif, ...

```xml
<?xml version="1.0" encoding="utf-8"?>
<secure_manifest xmlns:secure="./secure.xsd">
    <pkg_info name="com.fitness.app" pub_key="..." />
    <!-- Interdiction d'accès aux ressources sensibles par facebook -->
    <secure_context>
        <app_restriction>
            <app_list>
                <app name="com.facebook.app" />
            </app_list>
            <resource_restriction>
                <resource name="gps" concurrent="true" />
                <resource name="coarse_location" concurrent="true" />
                <resource name="motion_sensors" concurrent="true" />
            </resource_restriction>
        </app_restriction>
    </secure_context>
</secure_manifest>
```

Listing 4 – Exemple d'utilisation exclusive de ressources

3.2.3 Distribution des *Secure Manifest*

Concernant la distribution de ces *Secure Manifest*, nous avons envisagé plusieurs possibilités. Tout d'abord, les utilisateurs peuvent eux-même créer ces *Secure Manifest* à l'aide d'une application facilitant l'élaboration de ces derniers. Ils seraient ensuite en mesure de partager les fichiers ainsi créés avec d'autres utilisateurs par n'importe quel moyen de partage de fichier. Cependant, nous pensons que la création d'un *Secure Manifest* peut être complexe et fastidieuse pour les utilisateurs non experts.

Cette problématique est renforcée par le fait que, pour définir des règles de manière assez précise, il est important de connaître la liste des ressources nécessaires au bon fonctionnement de chaque composant de l'application. En effet, une ressource sensible ne peut être employée que pour une activité rarement utilisée de l'application en question. De plus, toutes les ressources utilisées ne sont pas nécessairement sensibles d'un point de vue de la sécurité de l'application. Ainsi, déterminer la liste des ressources, dont l'utilisation exclusive améliorerait la sécurité de l'application, n'est pas une chose aisée pour des utilisateurs qui n'ont aucune connaissance spécifique de l'application. En outre, cette connaissance peut être apportée soit par le développeur de l'application lui-même, soit par des utilisateurs experts.

Dans notre implémentation, nous avons pris en compte cette problématique en permettant aux développeurs de chaque application de fournir un *Secure Manifest* « de base » dans l'archive de l'application. Ce *Secure Manifest* peut ensuite être enrichi par les utilisateurs via une interface dédiée afin d'affiner la correspondance aux préférences de chaque utilisateur en termes de vie privée. Cependant, une autre piste envisagée repose sur un site de crowd-sourcing, permettant aux dé-

veloppeurs et aux utilisateurs de maintenir un référentiel de *Secure Manifest* pour chaque application. Le site de crowd-sourcing serait ainsi administré par un collège d'experts qui examineraient et corrigeraient les *Secure Manifests* proposés par la communauté. Cette approche a comme principal avantage d'être indépendante de l'adoption de notre solution par les développeurs d'applications, mais nécessite la mise en place du-dit site. Par ailleurs, l'implémentation de ce dernier peut revêtir différentes formes (autre qu'un simple site web) telles qu'un store d'application alternatif (proposant cette fonctionnalité) ou une application de partage de *Secure Manifest* basée sur l'utilisation de protocoles P2P par exemple.

3.3 Les conflits et leur gestion

3.3.1 Définition et objectifs de la gestion des conflits

Après avoir décrit les capacités de la politique, et précisé son fonctionnement au travers des règles du *Secure Manifest*, il est important de se demander si des conflits peuvent émerger lors de sa mise en place. Dans notre cas, un conflit peut se manifester : d'une part, lorsqu'une action effectuée par une application n'est pas conforme aux règles actives de la politique ; et d'autre part, lorsque des règles devant être activées ne peuvent être respectées compte tenu du contexte courant d'exécution. Lorsque de tels conflits se produisent, il est important de maintenir le système dans un état sécurisé, tout en offrant la possibilité à l'utilisateur d'effectuer les actions qu'il souhaite. En effet, la sécurité apportée par notre solution ne serait d'aucun intérêt si elle rendait le système inutilisable. Ainsi, nous considérons l'expérience utilisateur comme un facteur qu'il ne faut pas négliger, car essentielle pour l'adoption de n'importe quel produit. La gestion des conflits mise en place dans notre solution a donc pour but d'apporter des réponses en ce sens.

Cependant, pour comprendre ce qu'est une gestion efficace des conflits, il est tout d'abord important de déterminer leurs possibles causes. Ainsi, il existe plusieurs classes de conflits :

- Les conflits **EC** (Execution Conflicts), résultant de la tentative de transgression d'une règle d'interdiction d'exécution.
- Les conflits **RC** (Resource Conflicts), résultant de la tentative de violation d'une règle de restriction d'accès à une ressource.

Plus précisément, si on considère A, une application devant être lancée, et B une application en cours d'exécution, les conflits de type **EC** peuvent se produire : soit parce que le lancement de A transgresse une règle d'interdiction d'exécution issue du *Secure Manifest* de B (**EC1**), soit parce que le *Secure Manifest* de A interdit à B d'être lancée (**EC2**).

Concernant les conflits de type **RC**, on peut envisager deux cas différents. En effet, si on considère A, une application essayant d'accéder à une ressource R, et B, une autre application en cours d'exécution, ce type de conflit peut se produire uniquement du fait de la présence d'une restriction sur l'usage de R dans le *Secure*

Manifest de B (**RC1**). Cependant, si l'application B utilise elle aussi la ressource R au moment ou A souhaite y accéder, un conflit **RC** peut être issu : de la présence d'une restriction sur l'usage R dans le *Secure Manifest* de A (**RC2**) ; ou bien, si la restriction ne concerne que l'usage simultané de la ressource R, de la présence d'une telle règle dans le *Secure Manifest* de B ou de A (**RC3**).

Dans le cadre de cette politique, nous avons fait le choix de laisser à l'utilisateur le soin de déterminer l'issue de chacun des conflits. Nous considérons en effet que seul l'utilisateur peut légitimement décider des actions qu'il souhaite effectuer en priorité sur son smartphone. Ceci est par ailleurs cohérent avec notre choix de laisser l'utilisateur configurer les règles de la politique.

Cependant, face à ces six types de conflits, deux problématiques principales émergent. Tout d'abord, comment régler ces conflits d'un point de vue pratique ? Ou plus précisément, comment faire en sorte que le système évolue pour faire respecter la volonté de l'utilisateur, tout en conservant un système sûr ? Deuxième point, comment mettre en place cette gestion des conflits d'un point de vue ergonomique ? C'est-à-dire, comment limiter les interactions, issues de cette gestion des conflits, qui peuvent nuire à la qualité de l'expérience vécue par les utilisateurs de notre solution [3] ? Les paragraphes suivant présentent nos réponses à ces deux problématiques.

3.3.2 Aspects pratiques

Tout d'abord, la nature même d'un conflit est liée à la tentative de violation d'une ou plusieurs règles actives, ou devant être activée, de la politique de sécurité. Ces règles étant intrinsèquement liées à l'exécution d'applications ou de leurs composants, il n'existe que deux possibilités pour résoudre le conflit :

- Interdire l'action à la source du conflit.
- Faire évoluer le système pour que cette action puisse être autorisée.

Par ailleurs, choisir l'une ou l'autre de ces possibilités doit dépendre entièrement de la volonté de l'utilisateur. Plus précisément, celui-ci doit donner son accord explicite pour autoriser ou non l'évolution du système, et ce en ayant connaissance des conséquences de cette évolution. Cette nécessité est due au fait qu'il est impossible pour le système de prendre cette décision à la place de l'utilisateur sans nuire à son expérience.

Ainsi, si l'utilisateur donne son accord, l'évolution du système peut prendre plusieurs formes en fonction du type de conflit. Pour ceux de type **EC1**, cela consiste à fermer la ou les applications dont le *Secure Manifest* comporte la règle interdisant le lancement de la nouvelle application. De manière similaire, pour les conflits de type **EC2**, l'évolution du système se caractérise par la fermeture des applications n'étant plus autorisées à s'exécuter, de part les nouvelles règles activées par le lancement de la nouvelle application ou composant.

3. On parle d'expérience utilisateur, aussi appelé *UX*.

Concernant les conflits de type **RC**, afin de faire respecter les règles restreignant l'usage de ressources ou garantissant leur utilisation exclusive, l'évolution du système consiste à : 1) couper les accès des applications étant en train d'utiliser la ressource en question ; ou 2) fermer ces applications de manière similaire aux conflits **EC**. Dans les deux cas cette évolution permet soit de rendre conforme les applications à la nouvelle règle ou bien de libérer l'accès exclusif à la ressource, soit de supprimer la restriction d'accès en fermant l'application associée à cette règle.

Par ailleurs, le fait de fermer automatiquement certaines applications ne pose aucun problème vis-à-vis du système, le procédé étant semblable à celui du système LMK du noyau. Nous avons donc choisi de privilégier ce type d'action pour résoudre les conflits lorsque l'utilisateur préfère effectuer une action et faire évoluer le système en conséquence. Ce choix est de notre point de vue le plus approprié, même pour les conflits de type **RC**, car les applications ne sont pas habituellement conçues pour gérer la perte d'accès à une ressource [4]. De plus, en gérant ainsi les conflits, nous évitons toutes possibilités d'interblocages : l'évolution du système conduira dans le pire des cas à la fermeture de toutes les applications sauf une. En outre, utiliser ce mécanisme permet de s'affranchir de plusieurs contraintes techniques, comme expliqué dans le chapitre 4.

3.3.3 Aspects ergonomiques

Comme évoqué précédemment, il est nécessaire de questionner l'utilisateur pour déterminer l'issue d'un conflit, ne connaissant pas préalablement son intention. Ces interactions prennent la forme de questions fermées dont voici deux exemples : « Le lancement de l'application X entraînera la fermeture de l'application Y, approuvez-vous ce lancement ? » ; « L'application X souhaite accéder à la ressource R, ce qui entraînera la fermeture de l'application Y. Approuvez-vous cet accès ? ».

Cependant, cette interaction peut ajouter de la lourdeur à l'utilisation, il est donc nécessaire de la minimiser le plus possible. Dans ce but, notre solution s'appuie sur deux points. Nous souhaitons d'une part n'effectuer que les interactions strictement nécessaires, et d'autre part adapter ces interactions au contexte d'exécution pour qu'elles apparaissaient moins dérangeantes aux yeux de l'utilisateur.

Un moyen d'atteindre ce but réside dans la prise en compte de l'état d'exécution des applications, et plus particulièrement de leurs composants actifs. En effet, comme évoqué dans le chapitre 1, chaque composant possède un cycle de vie évoluant en fonction d'un *état* associé au composant, et qui dépend du type de composant. Or en observant l'évolution de cet état, il est notamment possible de déterminer si un composant n'est maintenu actif par le système que pour des raisons de performances.

Par exemple, après qu'une activité ait été « fermée [5] », par appui sur la touche « retour » ou en étant explicitement effacée de la liste des applications récentes, nous considérons qu'un conflit mettant en cause ce composant peut être résolu de ma-

4. Couper dynamiquement l'accès à une ressource déclencherait alors un crash de l'application.
5. Du point de vue de l'utilisateur et non pas du système.

nière automatique. Autre exemple, un service « lié » ne possédant plus d'application cliente peut-être traité de manière similaire.

Par ailleurs, cet état peut aussi nous fournir des informations permettant de déterminer l'urgence de la résolution du conflit. En effet, un conflit issu d'une action réalisée en arrière plan ne nécessite pas d'interaction prioritaire de la part de l'utilisateur. Ainsi pour ce cas précis, on informe l'utilisateur via le système de notification, qui de manière asynchrone peut régler le conflit ultérieurement.

Plus de détails sur la considération des états des composants applicatifs sont fournis dans le chapitre 4. En outre, il est possible pour l'utilisateur d'enregistrer le résultat d'un conflit. Ainsi, si un conflit se produit dans les même conditions, il pourra être résolu sans interventions de l'utilisateur.

3.3.4 Améliorations possibles

Nous avons réfléchi à des améliorations supplémentaires pour agrémenter cette gestion des conflits, mais qui n'ont pas pu être intégrées à la solution car elles auraient nécessité un travail d'ingénierie supplémentaire que nous n'avons pas pu nous permettre par manque de temps. Cependant, la réflexion associée à ces améliorations mérite d'être abordée pour des travaux futurs.

La première consiste à donner la possibilité à l'utilisateur de redémarrer les composants d'arrière plan [6], ayant été fermés à la suite d'un conflit, une fois que les règles actives le permettent à nouveau. Ainsi, les fonctionnalités associées à ces composants seraient automatiquement redémarrés, évitant à l'utilisateur de devoir penser à les relancer manuellement. Par exemple, pour une application de messagerie instantanée, cela permettrait de recevoir dès que possible les messages ayant été envoyés pendant le laps de temps ou l'application était fermée.

Une deuxième amélioration repose sur l'adaptation de la manière de contrôler les accès aux ressources en fonction de la nature de celles-ci. Plus précisément, on peut faire la distinction entre les ressources volatiles, issues des capteurs ou autres périphériques matériels, et les ressources permanentes, qui sont stockées d'une manière ou d'une autre sur les supports de stockage du smartphone. À partir du tableau 3.1 des ressources considérées, on peut observer que ces ressources permanentes appartiennent aux groupes « Données personnelles » et « Stockage ». Or, étant donné que ces données permanentes sont de nature à persister dans le temps, il serait intéressant de protéger les objets associés à ces ressources et issus d'applications critiques, et ce même après leur utilisation. En effet, si ces applications critiques définissent un accès exclusif à ces ressources, pour permettre notamment d'éviter les attaques de type TOCTOU, l'intégrité et la confidentialité de ces données peut être compromise après leur utilisation par l'application.

Bien évidemment, des contre-mesures peuvent être implémentées au sein même des applications, via l'utilisation de chiffrement ou de contrôle d'intégrité, mais nous pensons qu'intégrer ce type de fonctionnalité à notre solution pourrait être utile pour les applications n'ayant pas envisagé ces problématiques de sécurité.

6. C'est-à-dire les services, content-providers et broadcast receivers

Cependant, il est important de faire la différence entre les données à caractère public et privé, car il ne faut pas oublier que ces ressources peuvent aussi être utilisées pour partager des données entre les applications. Cependant on peut assez facilement déterminer la nature des données stockées en fonction de certains paramètres. En effet, d'une part les ressources du groupe « Données personnelles » sont de nature à être partagées entre les applications désignées par l'utilisateur [7]. D'autre part, les ressources du groupe « Stockage » sont de nature publique si et seulement si leur destination de stockage est un répertoire conventionnellement destiné au partage de données [8].

3.4 Encourager l'utilisateur à paramétrer la politique

Au delà des différents aspects touchant à la politique, son efficacité, et l'expérience utilisateur qui découle de son utilisation, il est nécessaire de se questionner sur la volonté de l'utilisateur de véritablement employer cet outil pour appliquer les restrictions satisfaisant ses besoins en sécurité et vie privée.

Premièrement, les utilisateurs ne sont souvent pas conscients des implications sur la sécurité et la vie privée lorsqu'ils accordent leur accès aux ressources au travers des permissions Android. En effet, dans l'étude menée par Bonné et al. [Bonné 2017], de nombreux utilisateurs consultés regrettent d'avoir accordé l'accès d'applications à certaines ressources une fois les implications de ces accès leur ayant été explicitées. De plus, certains utilisateurs ne sont parfois même pas conscients d'avoir accordé ces accès aux applications.

Deuxièmement, les travaux de Almuhimedi et al. [Almuhimedi 2015] ont montré qu'encourager les utilisateurs à mieux configurer un gestionnaire de permissions sur Android, à travers l'apport d'informations sur l'utilisation de ces permissions, entraîne plusieurs effets bénéfiques :

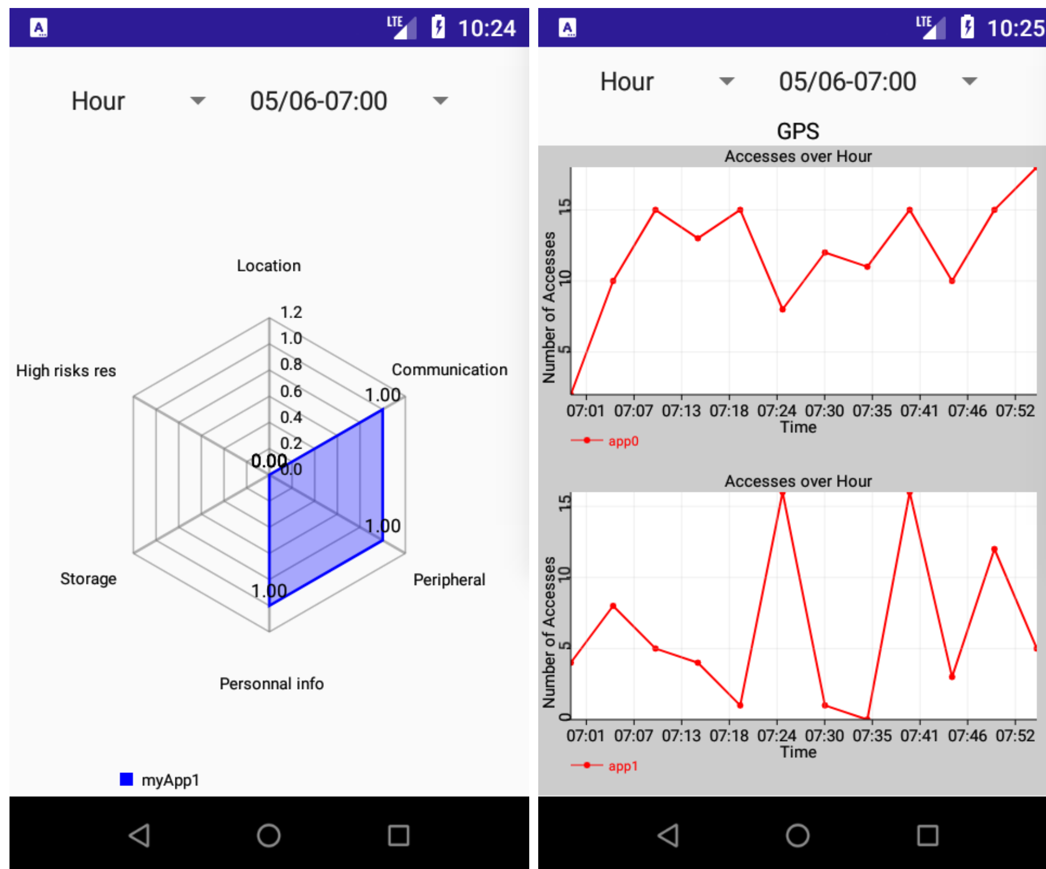
- Les utilisateurs déjà impliqués dans la configuration de gestionnaire de permissions ont plus de connaissances pour mieux configurer leur choix vis-à-vis de leur attentes en terme de sécurité/vie-privée, tout en pouvant observer les effets des choix effectués.
- Les utilisateurs qui ne se seraient pas intéressés à configurer un gestionnaire de permissions sont plus enclins à le faire : les confronter à ces données pique leur curiosité et les invite à se poser des questions sur l'utilisation de ces données et les problématiques de sécurité/vie-privée qui en découlent.

Par ailleurs, la visualisation de l'utilisation des permissions par les applications a déjà été étudié par Canbek et al. [Canbek 2017]. Dans leurs travaux, il est préconisé de rassembler les permissions en plusieurs groupes pour améliorer l'efficacité et l'intelligibilité des informations représentées. Dans notre cas, nous avons souhaité ajouter une fonctionnalité similaire à notre solution, afin d'augmenter son

7. Auxquelles l'utilisateur à accordé les permissions adéquates.
8. Par exemple, les répertoires `DCIM`, `Download`, `Music`, `Video` ...

attrait auprès des utilisateurs. Nous avons cependant conservé notre classification des permissions sous forme de ressources, et proposé des schémas de représentation originaux que nous pensons adaptés pour la visualisation de ce type d'information. Ainsi, nous avons développé une application capable de présenter l'utilisation des différentes ressources considérées, et ce de différentes manières :

- Un affichage rapide, sous forme de diagramme en étoile (figure 3.1a), permettant d'esquisser le profil d'utilisation des ressources d'une application. Dans ce diagramme, les ressources sont rassemblées par groupe d'appartenance afin d'améliorer la lisibilité.
- Un affichage plus détaillé, sous forme de graphe (figure 3.1b), pour chaque ressource permettant de récapituler leur utilisations pour chaque application à différentes granularités de temps (mois, semaine, jour, heures).



(a) Visualisation rapide (b) Visualisation approfondie

FIGURE 3.1 – Visualisation de l'utilisation des ressources par une application

L'utilisation du diagramme en étoile permet notamment de comparer les profils d'utilisation des ressources dans le temps. En effet, ce type de diagramme permet de représenter aisément plusieurs profils afin de les comparer visuellement. Quant

au graphe plus détaillé, il permet aux utilisateurs d'observer le comportement de l'utilisation de chaque ressource, et ce pour chaque application. Ces deux affichages combinés permettent donc d'observer les effets d'une quelconque modification de la politique de sécurité en cours, ou d'un changement de comportement d'une application suite à une mise-à-jour. Nous n'avons cependant effectué aucune mesure quantitative pour tester l'influence de ce type d'aides visuelles sur l'adoption de notre solution, ni contacté d'ergonomes à ce sujet.

3.5 Conclusion

Dans ce chapitre nous avons présenté notre proposition de politique de sécurité adaptée à l'environnement Android. Cette politique a été conçue dans le but d'offrir la possibilité à ses utilisateurs de définir des règles dynamiques conditionnant les accès aux ressources du système, ainsi que le démarrage des applications installées. Ces règles viennent ainsi compléter le système de permissions d'Android afin de répondre à un certain nombre de menaces émergentes, comme le détournement de communications BLE [Sivakumaran 2018], ne pouvant être traitées par ce dernier. Nous avons ainsi détaillé les objets qu'une telle politique nécessite de contrôler pour prendre en compte ces menaces dans l'environnement spécifique d'Android. Plus précisément nous avons réuni l'ensemble des ressources considérées en groupes, et indiqué les risques de sécurité leur étant associés. Nous avons ensuite énuméré les contraintes de mise en œuvre du schéma d'autorisation de notre politique vis-à-vis des informations de contexte d'exécution devant être recueillies. Nous avons enfin explicité le modèle de menace et les scénarios d'attaques considérés pour l'élaboration d'une telle politique.

Suite à ces détails de conception, nous avons présenté l'implémentation de cette politique et des règles qui la composent. Ainsi, nous avons illustré le format de ces règles, stockées dans des fichiers xml nommés *Secure Manifests*, à travers un ensemble d'exemples démontrant les capacités concrètes de notre politique. Nous avons par ailleurs précisé la manière de distribuer ces *Secure Manifest* par les utilisateurs ou les développeurs, et ce en prenant en compte le niveau d'expertise requis pour leur création.

Cependant la mise en œuvre des règles de notre politique est amenée à générer un certain nombre de conflits devant être résolus par le système qui l'implémente. Nous avons ainsi listé les différents types de conflits, caractérisés leur origine possible, et indiqué les évolutions du système nécessaires à leur résolution. De plus, nous avons explicité les mesures d'ergonomies utilisées pour que l'usage de notre solution impacte le moins possible l'utilisateur.

Enfin, nous avons abordé l'adoption de notre solution à travers l'implication accrue de l'utilisateur. Dans ce but, nous avons proposé une application permettant la visualisation de la consommation des différentes ressources du smartphone. De cette manière, un utilisateur curieux est capable de se renseigner sur le profil de chaque application installée afin de paramétrer plus efficacement notre politique.

Suite à cette première contribution, le chapitre suivant est consacré à la description détaillée d'une architecture de sécurité capable de mettre en application une telle politique.

Chapitre 4

Architecture de sécurité multi-niveau

Sommaire

4.1 Besoins et contraintes techniques de notre politique 61
4.1.1 Contrôle d'accès et contexte 61
4.1.2 Protections contre le modèle d'attaque envisagé 62
4.1.3 Un moniteur de référence multi-niveau 63
4.2 Présentation de l'architecture 64
4.2.1 Aperçu des composants et caractéristiques 64
4.2.2 Détails des composants 65
4.3 Mise en œuvre de la politique 67
4.3.1 Récupération des *Secure Manifest* 67
4.3.2 Génération, activation et mise en application des règles . . . 68
4.3.3 Gestion des conflits et expérience utilisateur 74
4.4 Contrôle de l'intégrité de notre solution 76
4.4.1 Périmètre de la protection 76
4.4.2 Protection contre les attaques logicielles 77
4.4.3 Protection contre les attaques matérielles 79
4.4.4 Protection au démarrage et chaîne de confiance 80
4.5 Conclusion 81

Faisant suite à la présentation de notre politique de sécurité, ce chapitre a pour but de détailler les moyens logiciels et matériels permettant sa mise en œuvre : une architecture de sécurité multi-niveau [Averlant 2018, Averlant 2017b]. Ce chapitre s'articule en trois parties. La première récapitule les besoins d'interception et les contraintes à prendre en compte pour cette architecture. Une deuxième partie donne un aperçu de l'architecture dans son ensemble. Enfin, une troisième partie présente les détails de chacun des composants de cette architecture.

4.1 Besoins et contraintes techniques de notre politique

4.1.1 Contrôle d'accès et contexte

Dans le chapitre 3, nous avons présenté notre politique de sécurité définissant des règles dynamiques, régissant les accès entre les applications et les ressources du

système, et dont l'activation est conditionnée par certains paramètres du contexte d'exécution. Par ailleurs, nous avons vu dans le chapitre 2, que les objets du système derrière ces ressources peuvent provenir du framework ou bien du noyau.

Dans un système classique, le contrôle des accès aux objets d'un système est usuellement effectué par un moniteur de référence (§2.1.3.4). Cependant, dans notre cas, ce moniteur de référence nécessite d'intercepter les communications ciblant les objets hébergés à différents niveaux de privilèges et d'abstractions : le framework et le noyau. Ainsi la conception du moniteur de référence doit être adaptée pour satisfaire sa propriété d'être « toujours invoqué ». Par conséquent, il est nécessaire pour celui-ci de posséder la capacité d'intercepter l'ensemble des communications pour ces deux types d'objets.

Pour répondre à cette problématique, un moniteur de référence standard hébergé dans le noyau peut utiliser des mécanismes d'introspection pour obtenir les connaissances nécessaires à l'interception des communications du framework. Cependant cette méthode se heurte à plusieurs contraintes. D'une part, les ressources matérielles réduites d'un smartphone sont un facteur limitant pour l'adoption de méthodes d'introspection. Celles-ci imposeraient en effet un coût en performances trop important pour le système. D'autre part, notre système nécessite de nombreuses informations issues du framework pour constituer le contexte d'exécution nécessaire à l'activation dynamique des règles, ainsi qu'à la résolution des conflits. Il est donc essentiel de répondre à cette problématique autrement.

4.1.2 Protections contre le modèle d'attaque envisagé

Le modèle d'attaque considéré pour les travaux de cette thèse (§3.1.5) est assez vaste. Notamment, nous prenons en compte les attaques ciblant le logiciel s'exécutant dans les couches les plus privilégiées. De plus, les vecteurs d'attaque considérés ne se limitent pas seulement aux applications présentes sur le smartphone. En effet, nous souhaitons que notre solution soit également robuste vis-à-vis des attaques pouvant transiter par les périphériques matériels du smartphone. Par conséquent, il est nécessaire de fournir à notre moniteur de référence des mécanismes de protections adaptés à ces types d'attaque, afin que celui-ci soit bel et bien « inviolable ».

La garantie de cette propriété pourrait être apportée par un périphérique matériel dédié [Morgan 2015a], chargé de vérifier l'intégrité du moniteur de référence. Cependant, adapter un tel type de matériel pour le contexte du smartphone nécessiterait un travail d'intégration très poussé, impliquant de nombreux inconvénients :

- Le matériel nécessiterait d'être miniaturisé et intégré au PCB du smartphone [1], ce qui nécessiterait le concours des fabriquants de PCB, en plus d'avoir un coût non négligeable.
- Notre solution serait dépendante de la présence d'un tel matériel, ce qui limiterait la disponibilité d'une telle solution pour le grand public.

1. Dans [Morgan 2015a], le matériel dédié prend la forme d'une carte d'extension PCIExpress.

Là encore, notre architecture devra prendre en compte cette problématique pour y apporter une réponse appropriée.

4.1.3 Un moniteur de référence multi-niveau

Afin de s'adapter aux besoins de notre politique de sécurité, et de répondre aux défis relatifs à notre environnement matériel, nous souhaitons proposer une évolution du concept de moniteur de référence, à travers une architecture dite *multi-niveau*.

Comme son nom le laisse penser, cette architecture a la particularité d'être implémentée dans différentes parties du système, ou plus précisément dans du logiciel de différents niveaux de privilège et d'abstraction. Ainsi, nous n'avons plus « un » moniteur de référence, mais une décomposition en modules de ce dernier, répartis dans plusieurs couches logicielles.

Il y a plusieurs avantages à utiliser ce type d'architecture. Premièrement, la médiation des communications est effectuée au plus près des objets du système, ce qui permet de ne pas avoir à recourir à des procédés d'introspection. De manière analogue, des informations sur le système peuvent être glanées à différents niveaux d'abstraction sans impact important sur les performances. Ensuite, chaque module de l'architecture est spécialisé, et peut assurer séparément une ou plusieurs propriétés de sécurité en fonction de son niveau de privilège. Enfin, la sécurité des éléments de cette architecture est renforcée par un mécanisme de *chaîne de confiance* : les logiciels les plus privilégiés sont garants de l'intégrité des logiciels les moins privilégiés.

En gardant ces considérations à l'esprit, nous avons défini quatre objectifs de conception pour notre architecture multi-niveau, qui est présentée dans la section suivante :

O1 *Médiation complète des communications des applications.* L'ensemble des canaux de communication des applications doivent être interceptés afin de garantir une application rigoureuse des règles de la politique de sécurité.

O2 *Protection de l'intégrité de la solution.* L'intégrité de notre solution doit être protégée pour assurer sa robustesse face aux attaques envisagées.

O3 *Nombre restreint de modifications du code existant.* Pour pouvoir adapter notre solution aux futures versions d'Android, il est nécessaire de réduire le nombre de modifications apportées au code Android existant, ainsi que de réduire la « dispersion » de ce code.

O4 *Efficacité et expérience utilisateur.* L'impact sur les performances doit être aussi faible que possible. De plus, les interactions obligatoires avec l'utilisateur devraient être limitées pour une meilleure facilité d'utilisation.

4.2 Présentation de l'architecture

4.2.1 Aperçu des composants et caractéristiques

Afin de satisfaire ces objectifs, nous avons conçu une architecture multi-niveau en quatre composants distincts, présentés dans la figure 4.1. Plus précisément, cette figure présente l'intégration de ces composants au sein de l'architecture Android et spécifie les différentes interactions entre ces composants.

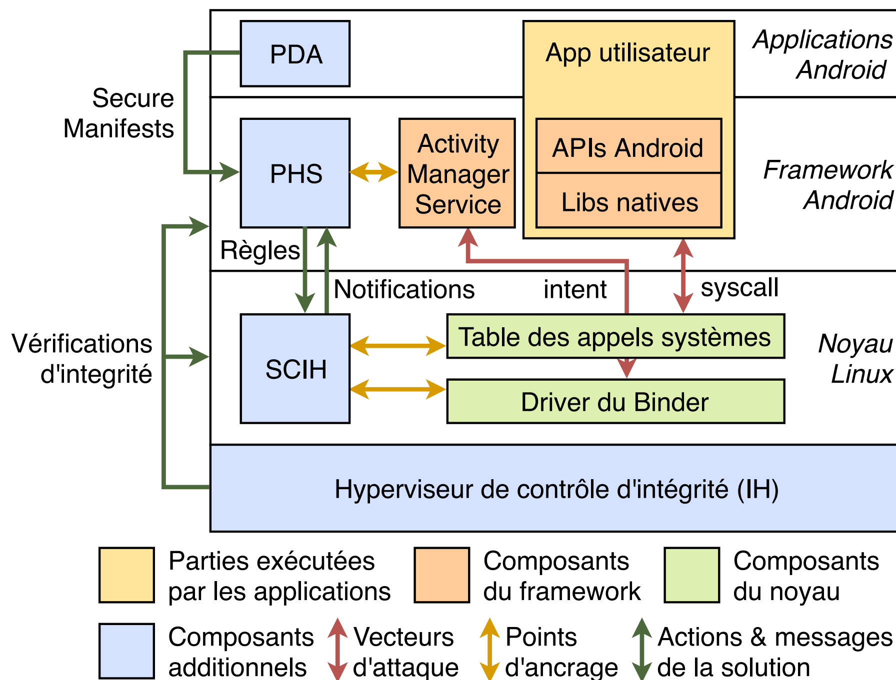


Figure 4.1 – Aperçu de l'architecture

Ainsi, les quatre composants de notre architecture sont respectivement nommés (par ordre croissant de privilèges associés) :

- Le **PDA** (*Policy Definition App*), est une application Android jouant le rôle d'interface utilisateur de notre solution.
- Le **PHS** (*Policy Handler Service*), est un service du framework responsable de la mise en application des règles de la politique.
- Le **SCIH** (*System Call Interception Handler*), est un module du noyau Linux s'assurant du respect des règles de la politique en complément du PHS.
- L'**IH** (*Integrity Hypervisor*), est un hyperviseur *bare-metal* utilisé comme *composant de confiance* de notre architecture.

Le reste de la section est dédié à la spécification du fonctionnement de ces quatre composants et de leurs interactions.

4.2.2 Détails des composants

4.2.2.1 L'application de définition des politiques (PDA)

Le premier composant, appelé *PDA*, est une application système fournissant l'interface utilisateur de notre solution. Il vise à :

- Récupérer le *Secure Manifest* par défaut de chaque application installée
- Permettre à l'utilisateur de modifier les *Secure Manifest* en ajoutant ou supprimant des règles
- Informer l'utilisateur de manière appropriée de tout conflit concernant les règles actives de la politique de sécurité (**O4**).

Le *PDA* communique avec le *PHS* en utilisant des transactions binder classiques. Ce type de communication permet notamment au *PDA* d'être notifié lors de l'occurrence d'un conflit requérant l'intervention de l'utilisateur, puis d'envoyer au *PHS* la réponse de l'utilisateur concernant ce conflit. Ces transactions peuvent être protégées par l'utilisation d'une permission spécifique, octroyant au *PDA* cette capacité de communication, et signée avec les *platform keys*.

4.2.2.2 Le service de gestion des politiques (PHS)

Le but de ce deuxième composant est d'administrer et d'appliquer la politique de sécurité au sein du framework Android. Ainsi, il a la charge de s'occuper de la gestion des informations issues des *Secure Manifest* de chaque application installée, et de générer les règles dynamiques à appliquer par notre moniteur de référence. De plus, il vise à intercepter les *IPC* de haut niveau, c'est-à-dire les *intents* initiés depuis les applications en cours d'exécution (**O1**).

Pour remplir ces objectifs, nous avons conçu un composant du framework, nommé *PHS*. Ce dernier constitue la clé de voûte de notre solution qui relie l'ensemble des autres composants de notre architecture. En effet, étant hébergé dans le framework, il est le candidat idéal pour cette tâche, du fait de sa proximité avec les données du framework. Cette proximité lui permet effectivement d'obtenir efficacement le contexte d'exécution nécessaire à l'activation dynamique des règles, ainsi que les informations nécessaires à la gestion des conflits.

Nous avons choisi d'implémenter le *PHS* comme un service système du framework, ayant une très grande proximité avec l'*AMS*. Pour rappel ce dernier est le service par lequel transite l'ensemble des *intents*, et qui s'occupe additionnellement de la gestion du cycle de vie des composants des applications. Plus précisément, le *PHS* est relié à l'*AMS* par un ensemble de points d'ancrages permettant d'intercepter les *intents* lors de leurs traitement par l'*AMS*. En outre, des modifications additionnelles fournissent au *PHS* la capacité de récupérer des informations sur l'état des composants des applications en cours d'exécution. Ainsi, les informations nécessaires à la mise en œuvre de la politique sont obtenues avec un coût minimal sur les performances (**O4**).

4.2.2.3 Le gestionnaire d'interception des appels systèmes (SCIH)

Le positionnement du *PHS* au sein du framework impose cependant une limitation : il lui est impossible de contrôler les communications de plus bas niveaux transitant directement par le noyau Linux. Pour ce faire, contrairement à certaines solutions de l'état de l'art utilisant les API de tracing du noyau (comme *ptrace*), nous avons préféré créer un composant dédié au sein du noyau pour effectuer cette tâche. Ainsi, nous évitons de substantielles modifications du daemon *Zygote* (**O3**), et pouvons bénéficier d'informations supplémentaires sur les objets du noyau à moindre coût (**O4**). Une autre possibilité aurait reposé sur la modification du code des différents services systèmes ou bien du *ServiceManager* (comme effectué dans [Ratazzi 2015]) afin d'intercepter les transactions binder bas niveau les concernant. Là encore, cette approche souffre de plusieurs inconvénients : impossibilité d'intercepter les communications bas niveau autres que celles issues du binder (**O1**), et modifications conséquentes du code des services système devant être adapté à chaque mise-à-jour majeure d'Android (**O3**).

Ainsi, le troisième composant de notre architecture, appelé *SCIH*, vise à intercepter les canaux de communication des applications qui ne peuvent être gérées par le *PHS*, à savoir les transactions bas niveau du binder et les appels systèmes (**O1**). Ce composant est implémenté en tant que module du noyau Linux, qui est automatiquement chargé au démarrage du système. Contrairement à de nombreuses solutions de l'état de l'art, nous avons décidé de ne pas utiliser l'API *LSM* (§2.2.2) pour éviter toute interférence avec le sous-système SELinux, ainsi que pour conserver plus de liberté sur les données interceptées. Ainsi le *SCIH* effectue l'interception des appels systèmes en modifiant directement la « table des appels systèmes » qui recueille les points d'entrée de chacun des appels systèmes. Ces modifications ont pour but de dérouter l'exécution de ces derniers avant leur traitement par le noyau, et ainsi pouvoir appliquer le contrôle d'accès requis. En outre, le module effectue une introspection dynamique de la mémoire du pilote du binder pour identifier la cible de chacune des transactions de ce type, évitant ainsi de modifier directement le code de ce pilote (**O3**). Enfin, le module implémente un pilote de périphérique « en mode caractère » permettant la communication entre le *SCIH* et le *PHS* via des appels systèmes *ioctl*.

4.2.2.4 L'hyperviseur de contrôle d'intégrité (IH)

Le dernier composant, nommé *IH*, est un hyperviseur *bare-metal*. En d'autres termes, il ne s'appuie sur aucunes fonctionnalités ou couches d'abstraction du matériel offertes par un système d'exploitation, contrairement à un hyperviseur dit *hébergé*. L'*IH* est ainsi amené à jouer le rôle de *composant de confiance* de notre architecture (**O2**). Dans ce but, il effectue des contrôles d'intégrité sur les parties critiques du noyau Linux afin de détecter les possibles altérations malicieuses. Cette détection est enrichie par l'interception des accès à certaines pages mémoires critiques du noyau. De plus, l'*IH* s'assure de l'intégrité des autres composants de la

solution au démarrage en vérifiant leur signature numérique.

Le choix d'implémenter ces fonctionnalités dans un composant possédant des privilèges plus élevés que ceux du noyau Linux a été effectué pour prendre en compte la propriété « vérifié correct » d'un moniteur de sécurité. Pour être plus précis, il est probablement impossible de s'assurer qu'aucune faille ne soit présente dans le noyau Linux, et a fortiori dans le framework Android, ceci dû à la complexité de leur base logicielle. Il semble en effet très difficile d'appliquer des procédures de type *méthodes formelles*, sur des bases de code si importantes et complexes [2]. Ainsi, comme l'*IH* possède un nombre de fonctionnalités réduit, il sera plus aisé d'appliquer ce type de vérification à son sujet, même si cela n'a pas été effectué dans le cadre de cette thèse. La confiance légitime que l'utilisateur peut ainsi avoir dans ce composant est applicable au reste de l'architecture sous le couvert des vérifications de sécurité effectuées par l'*IH*.

4.3 Mise en œuvre de la politique

4.3.1 Récupération des *Secure Manifest*

Afin de mettre en œuvre la politique de sécurité, il est tout d'abord nécessaire de récupérer les *Secure Manifests* par défaut des applications. Cette tache qui incombe au *PDA* peut être effectuée lors de l'installation d'une nouvelle application, ou bien lors du premier lancement de cette application. Dans le premier cas, le *PDA* peut s'abonner aux évènements asynchrones envoyés par le système lorsqu'une nouvelle application est installée, modifiée ou supprimée. Dans le deuxième cas, le lancement d'une application ne possédant aucun *Secure Manifest* déclencherait le processus.

Lors de l'élaboration de notre politique de sécurité, nous avons considéré que les *Secure Manifest* par défaut étaient contenus dans l'archive des applications. Ainsi, il est possible de les récupérer depuis le répertoire d'installation de leurs applications d'appartenance. Cependant, le *PDA* ne peut effectuer cette action par lui même car il ne possède pas les droits d'accès lui permettant de consulter ces répertoires. Il nécessite donc le concours du *PHS* qui est chargé de recopier ces *Secure Manifests* dans le répertoire privé du *PDA*. Ainsi, ces précieux fichiers ne sont pas accessibles depuis les autres applications. Le *PHS* effectue néanmoins certaines vérifications supplémentaires avant d'effectuer cette action. Premièrement, il contrôle la conformité syntaxique du fichier. Deuxièmement, il vérifie l'adéquation du *Secure Manifest* avec l'application en question (nom du package, signature du développeur et version de l'application). Par ailleurs, si aucun *Secure Manifest* n'est fourni par le développeur de l'application, le *PDA* en génère un vide. Ensuite, il est demandé à l'utilisateur, lors du premier lancement de chaque application, de personnaliser ces règles par défaut via l'émission d'une notification issue du *PDA*.

Pour ce traitement, les communications entre le *PDA* et le *PHS* sont effectuées par des transactions Binder. En effet, le *PHS* expose un service système avec lequel

2. Le noyau Linux est composé de plus de 20 millions de lignes de code.

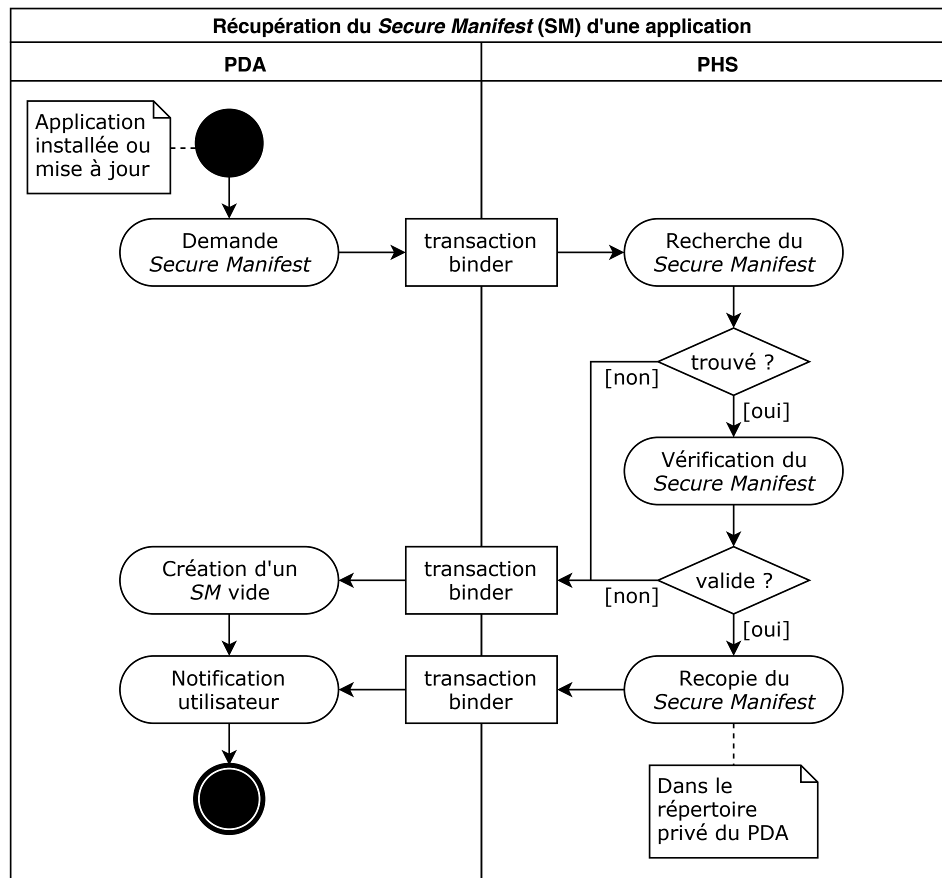


FIGURE 4.2 – Récupération du Secure Manifest

seul le PDA peut interagir. Ces communications sont résumées dans la figure 4.2.

4.3.2 Génération, activation et mise en application des règles

Une fois l'ensemble des *Secure Manifest* rassemblés par le processus décrit précédemment et édités par les utilisateurs à leur convenance, il est nécessaire d'analyser ces fichiers afin que le moniteur de référence génère les règles de contrôle d'accès à appliquer. Cette étape est directement effectuée par le *PHS*, qui va générer l'ensemble des règles issues des *Secure Manifest*, et les stocker dans des structures de données appropriées.

Concrètement, comme ces règles ne sont actives que sous certaines conditions, leur stockage est décomposé en deux parties. Un premier type de stockage recense l'ensemble des règles associées à chaque application, ou à ses composants le cas échéant, et qui peuvent être actives ou non en fonction du contexte d'exécution. De plus, une deuxième structure de données est dédié au stockage des règles actives que le moniteur de sécurité doit faire respecter à un instant donné. Cette séparation permet d'optimiser à la fois le stockage des règles et l'efficacité de leur mise en application. En outre, nous avons choisi d'affiner plus précisément ce stockage en fonction

du type de règle considéré. Ainsi, les règles actives empêchant l'exécution concurrente d'applications (**R1**) sont traduites par une simple liste noire d'applications n'étant pas autorisées à s'exécuter. De manière similaire, pour chaque ressource possédant au moins une restriction (règles **R2**), le *PHS* stocke la liste des applications autorisées (restriction globale) ou interdites (restriction spécifique) à accéder à celle-ci. Afin d'activer ou de désactiver les règles de la politique de sécurité, le *PHS* doit cependant se tenir à l'écoute de l'évolution du contexte d'exécution du système, comme expliqué dans la figure 4.3.

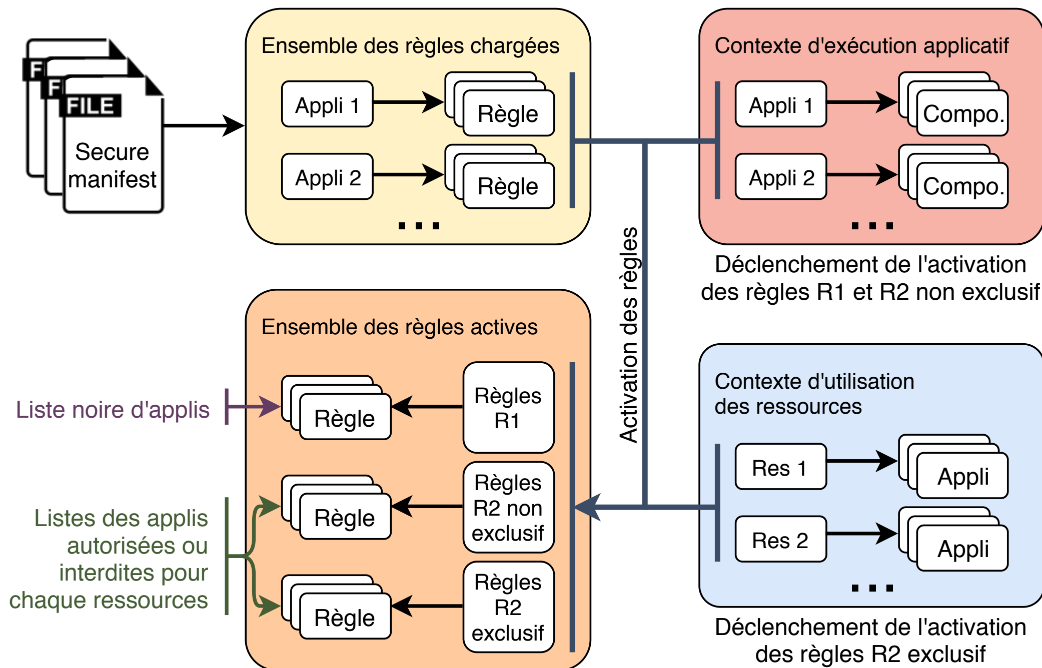


FIGURE 4.3 – Structures de données et activation des règles

Ainsi, il existe deux types d'évènements que le *PHS* doit surveiller :

E1 Le lancement ou arrêt d'une application ou d'un de ses composants.

E2 L'accès à une ressource non utilisée jusqu'à présent par une application.

Le premier type d'évènement (**E1**) est utilisé pour déclencher l'activation ou la désactivation des règles associées aux applications ou aux composants concernés. Au sujet du démarrage d'applications ou de composants, ceux-ci sont tous les deux le résultat d'intents envoyés par d'autres applications [3] ou par le système. Or, ces intents sont directement interceptés par le *PHS* grâce aux points d'ancrages qu'il possède dans l'*AMS*. De plus, concernant l'arrêt d'applications ou de composants, là encore la proximité avec l'*AMS* nous permet d'obtenir ces informations aisément, ce dernier maintenant la liste des composants actifs de chaque application. Le processus d'activation des règles issues de l'évènement **E1** est décrit dans la figure 4.4.

3. Notamment par l'application « launcher ».

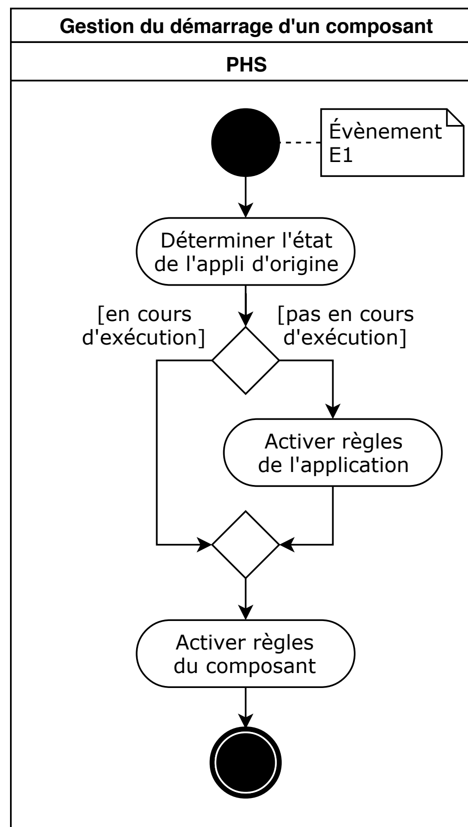


Figure 4.4 – Processus d'activation des règles (1/2)

Le second type d'évènement (**E2**) nécessite d'être pris en compte pour l'activation des règles restreignant l'utilisation simultanée de ressources. En effet, il n'est nécessaire de restreindre l'accès à la ressource en question que si l'application associée à la règle utilise bel et bien cette ressource. Cependant, accéder à une ressource, ou plus précisément à un des objets du système regroupés derrière cette dénomination, peut s'effectuer de différentes manières comme vu dans le chapitre 2. Encore une fois, la capacité d'interception des *intents* du *PHS* permet de détecter une partie des accès. Cependant l'identification des accès de plus bas niveau nécessite la collaboration du *SCIH*. La figure 4.5 représente le procédé de communication entre le PHS et le SCIH permettant de détecter les évènements de type **E2**, et précise le processus d'activation des règles **R1** et **R2**. Par ailleurs, comme on peut le voir sur la figure, la mise en application des règles de type **R2** nécessite également la collaboration du *PHS* et du *SCIH*. En effet, le premier accès à la ressource sera détecté soit par le *PHS*, soit par le *SCIH* en fonction du type d'accès, et la règle **R2** sera ensuite activée suite à cette détection. De plus, l'application de ce type de règle doit là encore être effectuée dans les deux composants.

La raison de cette coopération vient de la nécessité d'appliquer les règles à l'ensemble des communications du système. En effet, l'interception des communications doit nécessairement être effectuée dans le noyau afin de gérer l'accès aux objets de

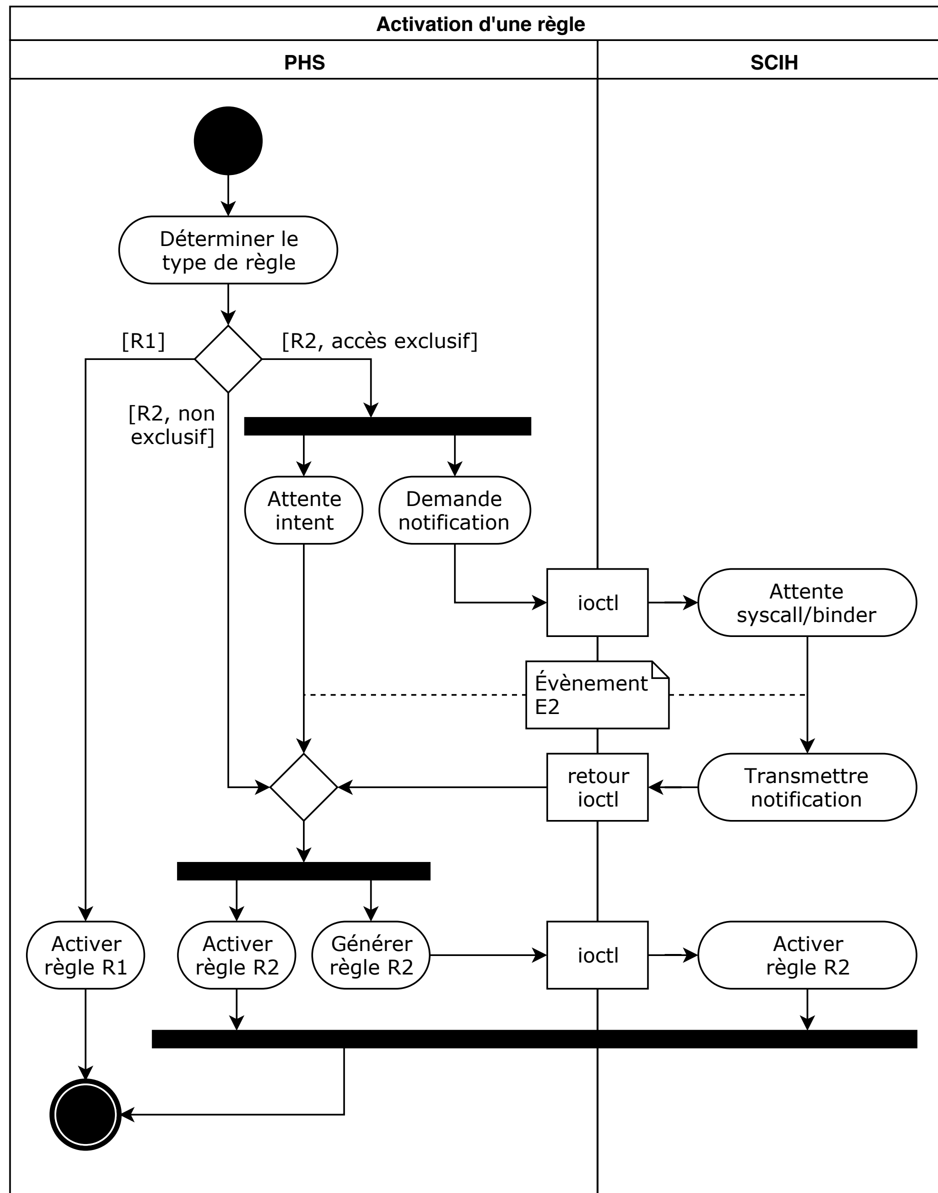


FIGURE 4.5 – Processus d'activation des règles (2/2)

bas niveau d'abstraction. Par ailleurs, les *intents* peuvent directement être interceptés dans le framework afin de bénéficier des informations de contexte nécessaires à l'application des règles leur étant associées. En outre, comme le *PHS* est le composant de notre architecture responsable du stockage et de l'activation/désactivation des règles de la politique de sécurité, il est indispensable qu'il communique ces règles au *SCIH* pour que celui-ci puisse les faire appliquer.

La communication entre ces deux composants est effectuée à l'aide du pilote de périphérique « en mode caractère » implémenté par le *SCIH*. Ainsi, c'est par ce canal de communication que transitent, d'une part les règles de sécurité que le

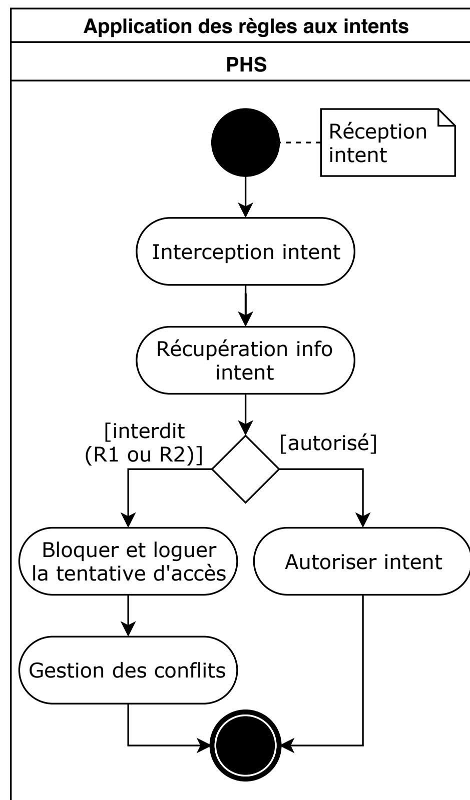


FIGURE 4.6 – Mise en application des règles aux *intents*

SCIH doit faire appliquer, et d'autre part certaines informations qui doivent être remontées au *PHS*. Ce dernier point concerne à la fois l'occurrence d'évènements de type **E2**, mais aussi les notifications nécessaires à la gestion des conflits.

Une autre approche à l'application des règles de la politique aux communications par appels systèmes aurait pu être possible. Elle aurait consisté à rediriger l'ensemble des appels systèmes des applications depuis le *SCIH* jusqu'au *PHS*. Cependant, le coût en performances aurait été bien trop important, car doublant le nombre effectif d'appels systèmes. Les figures 4.6 et 4.7 décrivent le processus d'application des règles de la politique, respectivement pour les règles s'appliquant aux *intents* ou aux appels systèmes.

Dans le premier cas, le *PHS* examine l'*intent* en question, et si celui-ci déroge aux règles **R1** (lancement d'une application de la liste noire) ou **R2** (accès à une ressource protégée par une liste noire ou blanche d'applications), l'accès est tout simplement refusé. Suite à ce refus, les détails de l'accès sont stockés à des fins de rapports, et un *processus de gestion des conflits* est entamé.

Pour le deuxième cas, le *SCIH* se charge de filtrer les appels systèmes ne respectant pas les règles **R2**. Là encore, l'appel système échoue si une des règles actives est violée. Cependant, une différence réside dans le traitement des accès pouvant générer des conflits. En effet, le *PHS* centralise cette gestion des conflits à l'aide du

PDA et a donc besoin d'être averti le cas échéant. Cette notification transite encore une fois par le pilote de périphérique « en mode caractère » du *SCIH*.

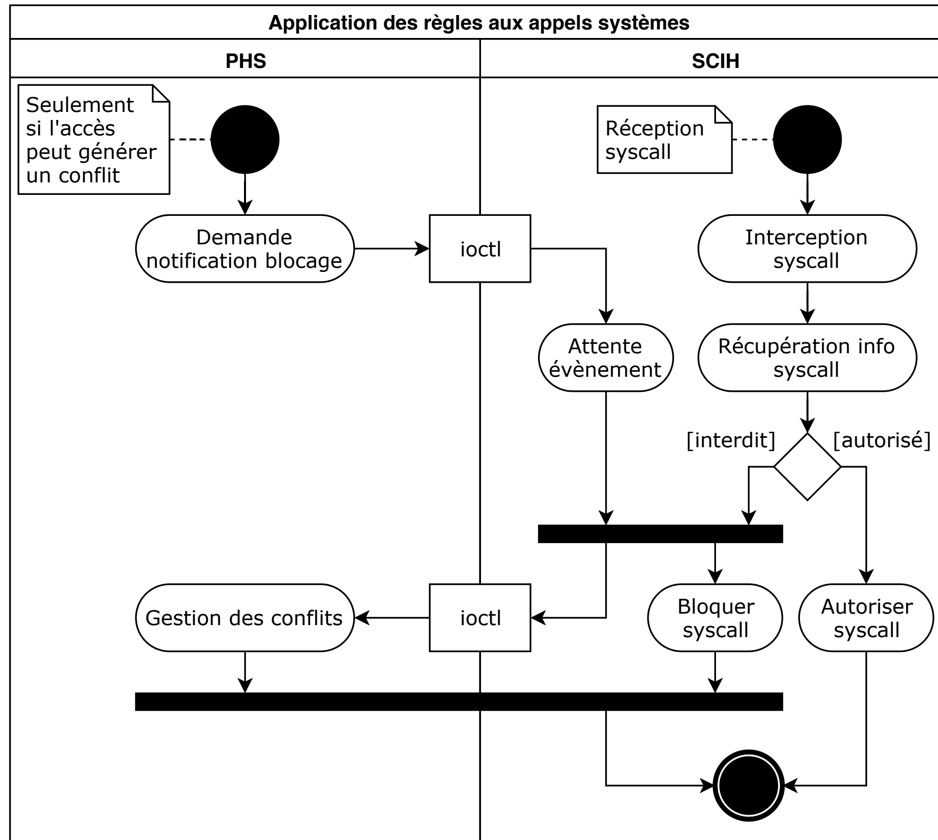


FIGURE 4.7 – Mise en application des règles aux appels systèmes

Complexité algorithmique associée

La vérification des règles qui doit être effectuée à chaque interception d'intent ainsi qu'a chaque appel appel système est une partie critique de notre architecture en termes de performances. Par conséquent, il est important de s'assurer que la complexité algorithmique de ce processus de vérification reste maîtrisée pour respecter les contraintes de performances du système. Cependant, la séparation des données relatives aux différentes règles actives nous permet de réduire cette complexité (se référer à la figure 4.3).

Premièrement, lorsqu'un intent émis par une application est destiné à une autre application n'étant pas déjà démarrée, le PHS parcours uniquement la liste noire des applications associées aux règles **R1**. La complexité associée est donc en $O(n)$ où n désigne le nombre d'applications inscrites dans cette liste noire.

Concernant les règles **R2**, le PHS ou le SCIH doivent consulter les structures stockant ces types de règles, respectivement pour les intents ou les appels systèmes destinés à accéder à une ressource. Pour ce faire, il est tout d'abord nécessaire de

savoir si une règle d'accès exclusif est active pour la ressource, et si oui déterminer quelle application possède l'exclusivité d'accès. Cette information est obtenue en parcourant la liste des règles **R2** exclusif. Si ce n'est pas le cas, le moniteur de références doit obtenir la liste des applications autorisées (restriction globale) ou interdites (restriction spécifique) à accéder à la ressource, issue des règles **R2** non exclusif. Dans les deux cas, nous sommes à nouveau face à un simple parcours de liste dont la complexité associée est en $O(n)$ où n désigne le nombre d'éléments contenus dans la liste parcourue.

4.3.3 Gestion des conflits et expérience utilisateur

Comme nous avons pu le voir dans le chapitre 3, la gestion des conflits est une étape importante quant à l'amélioration de l'expérience utilisateur. Le principe de cette gestion des conflits consiste en la collaboration du *PDA* et du *PHS*. Le *PDA* est chargé d'une part de personnaliser les alertes générées par ce processus pour les rendre ergonomiques pour l'utilisateur, et d'autre part d'adapter le type d'interaction afin de ne requérir une action de l'utilisateur que lorsque nécessaire. Le *PHS*, en tant que responsable de la bonne application de la politique, modifie l'état du système en fonction des réponses de l'utilisateur qui lui sont transmises par le *PDA*. De plus, c'est lui qui maintient la liste des accès pouvant générer ces possibles conflits, liste qui évolue nécessairement avec le contexte d'exécution du système et les retours de l'utilisateur. Ainsi, le processus de gestion des conflits est présenté dans la figure 4.8.

Lorsqu'un conflit à résoudre est détecté par le *PHS*, il transmet les informations relatives à ce conflit au *PDA*. Pour que cette transmission puisse avoir lieu, le *PDA* doit avoir communiqué une fonction de rappel qui sera utilisée pour traiter ces évènements. Ainsi, les informations transmises sont composées de :

- L'application source du conflit et son état.
- Le type d'accès effectué.
- La cible de l'accès (ressource ou application)
- La ou les règles violées par l'accès.
- La ou les applications ayant déclaré ces règles.

Ensuite, le *PDA* détermine la priorité de la résolution du conflit en fonction de ces informations. Plus précisément, si l'accès a été effectué depuis l'application possédant une activité au premier plan, il y a de fortes chances pour que cet accès soit le résultat d'une action initiée par l'utilisateur. Il est donc important de notifier ce dernier au plus vite, afin de le prévenir que l'action qu'il a souhaité (ou non) entreprendre est contraire aux règles actives de la politique de sécurité. Pour ce faire, le *PDA* génère un pop-up afin d'attirer l'attention de l'utilisateur sur le conflit rencontré. Dans le cas contraire, lorsque l'accès est effectué depuis une application en arrière plan, la résolution du conflit est moins prioritaire. Ainsi, le *PDA* alerte l'utilisateur de ce conflit en utilisant une notification, un élément graphique beaucoup moins invasif et pouvant être traité de manière asynchrone par l'utilisateur.

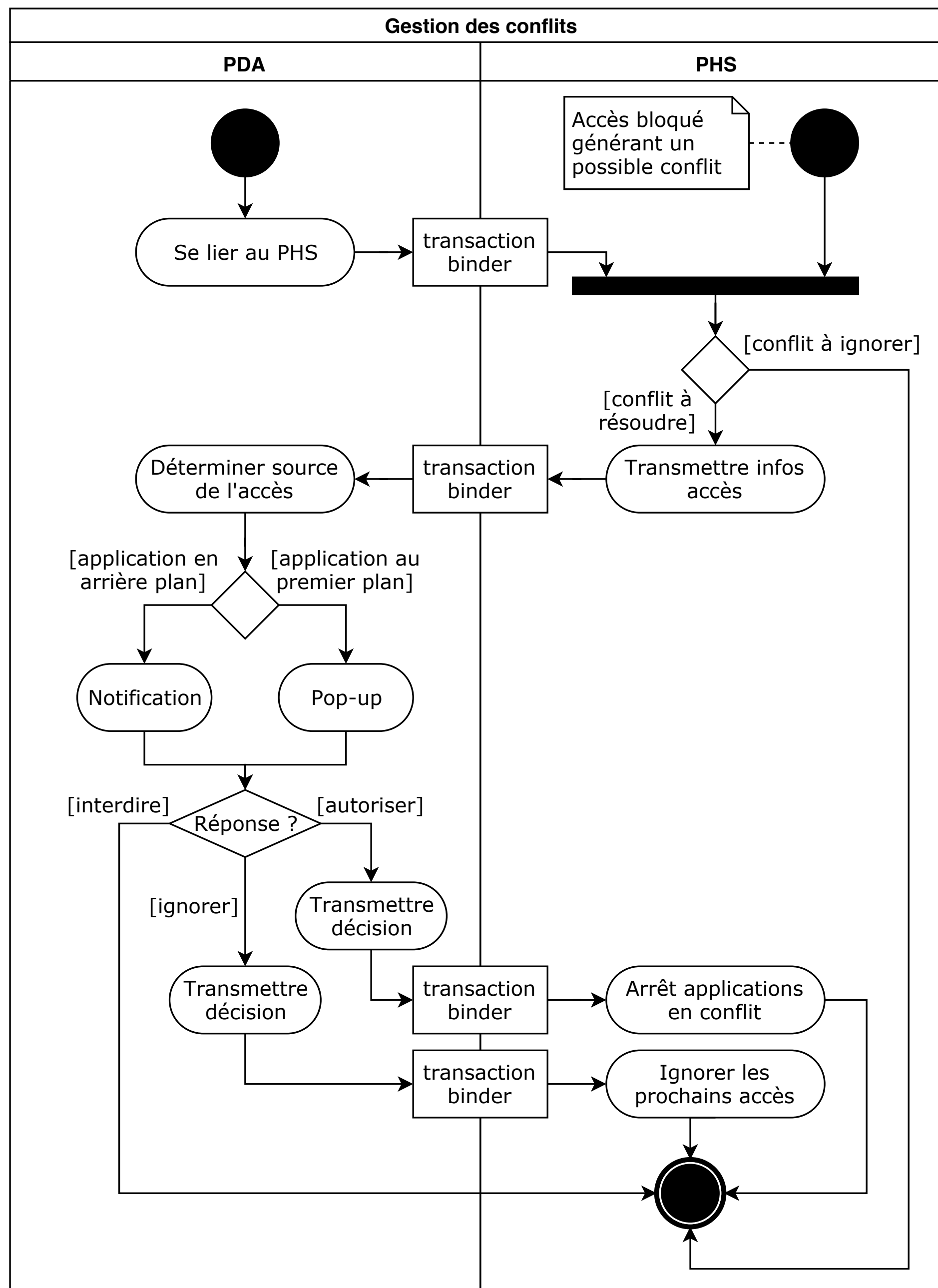


Figure 4.8 – Gestion des conflits

Une fois alerté du conflit, l'utilisateur est confronté au choix déterminant l'issue de celui-ci : soit il décide d'autoriser malgré tout l'accès responsable du conflit, soit il renonce à effectuer l'action. Dans le premier cas, la prochaine fois que le même accès se produit et dans la même configuration (mêmes règles en conflit), les applications dont les règles sont en conflit seront terminées par le *PHS*, conférant la capacité à l'action de se réaliser. Dans le deuxième cas, l'utilisateur peut aussi choisir de ne

plus être averti si cette action en particulier se répète dans le même contexte. Ce dernier point permet d'éviter les notifications à répétition si une application cherche à effectuer une action interdite en boucle. De plus, cela a aussi un effet bénéfique sur les performances de notre solution, car le *PHS* peut indiquer au *SCIH* de ne plus le notifier pour ces accès (si ceux-ci sont issus d'appels système).

Dans tous les cas, le système reste dans un état sécurisé car aucune des règles actives n'est transgressée. En effet, la résolution des conflits permet de faire évoluer le système pour éviter ce cas de figure.

4.4 Contrôle de l'intégrité de notre solution

Cet état sécurisé ne peut cependant exister si le moniteur de référence est compromis d'une quelconque façon. C'est donc l'objectif de notre *composant de confiance*, incarné par l'élément *IH* de notre architecture, de s'assurer que ce cas de figure ne se produise pas. Il est cependant important d'avoir en tête que cet hyperviseur n'a pas pour ambition d'implémenter un système de protection « ultime » pour l'ensemble du système Android. En effet, il est nécessaire de respecter les contraintes de conception que nous nous sommes fixées, et de ne pas impacter plus que ce qui est nécessaire les performances du système. L'*IH* a donc pour objectif de s'assurer que notre solution, et les mécanismes de sécurité sur lesquels elle repose, soient robustes face au modèle d'attaque considéré. Plus précisément, cette robustesse représente une garantie en terme d'intégrité de notre architecture.

4.4.1 Périmètre de la protection

Il est tout d'abord nécessaire d'identifier les éléments du système à protéger avant de pouvoir déterminer comment les protéger. Les paragraphes suivants présentent les éléments ainsi identifiés pour chaque composant de notre architecture.

Premièrement, concernant le *SCIH*, notre module du noyau Linux s'interface avec trois éléments essentiels à son bon fonctionnent :

- La table des appels systèmes, utilisée pour capturer les appels systèmes émis par les applications.
- Le driver du binder fournissant les informations essentielles au traitement des transactions binder.
- Les sous-systèmes du noyau permettant de récupérer les informations relatives aux objets manipulés par les applications (fichiers, sockets ...).

Pour les deux autres composants de notre architecture, il est cependant beaucoup plus complexe d'identifier rigoureusement l'ensemble des dépendances leur étant associées. Cependant, plusieurs mécanismes existants se chargent déjà de sécuriser l'ensemble des composants du framework (dont le *PHS*) et les applications (dont le *PDA)* : l'isolation native des processus (§1.2.2), et le système SELinux. Notre but sera donc de s'assurer que ces mécanismes, issus du noyau, soient eux-mêmes protégés.

On peut donc observer que l'ensemble des éléments additionnels à protéger sont hébergés dans le noyau Linux. Notre hyperviseur va donc devoir déployer des contre-mesures à l'égard des attaques logicielles ou matérielles ciblant ces composants.

4.4.2 Protection contre les attaques logicielles

Il existe plusieurs classes d'attaques logicielles pouvant cibler le noyau Linux. Celles-ci peuvent être décomposées en trois sous-familles :

- Les attaques (**A1**) visant à faire exécuter du code de l'espace utilisateur par le noyau pour bénéficier de ses privilèges (e.g. `ret2usr`, `ret2dir`).
- Les attaques ciblant l'intégrité des données du noyau en exploitant une vulnérabilité de corruption mémoire de ce dernier (e.g. `buffer overflow`, `use-after-free`, `race condition`). On peut distinguer les attaques se focalisant sur des données contrôlant le flux d'exécution [4] (**A2**), utilisées pour forcer le noyau à exécuter son code dans un ordre non prévu (`rop`), de celles ciblant les autres données [5] (**A3**) servant aux attaques `dop` [Hu 2016].

Pour contrer les attaques **A1**, le noyau peut configurer les droits des pages de la *MMU* traduisant les adresses virtuelles en adresses physiques. En effet, la *MMU* permet de définir cette translation par un ensemble de pages, associées aux adresses virtuelles manipulées par le logiciel, de granularité pouvant aller de 4ko à 1Go. Pour chacune de ces pages il est possible de préciser la politique de cache à employer, ainsi que les droits d'accès concernant les données de ces pages. Ainsi, grâce à la configuration de ces droits d'accès, et des registres système associés à la *MMU*, le noyau est capable de :

- Supprimer les droits en écriture sur les pages hébergeant le code du noyau.
- Interdire aux zones mémoires du noyau pouvant être modifiées de s'exécuter [6].
- Empêcher le code hébergé dans la mémoire de l'espace utilisateur de pouvoir être exécuté avec les privilèges du noyau [7].

Les noyaux Linux récents implémentent le support de ces fonctionnalités depuis la version 4.10. Il est ensuite possible de vérifier la bonne configuration de ces pages depuis l'hyperviseur en parcourant les tables de pages et en observant les registres de configuration de la *MMU*. Par ailleurs, si le noyau ne supporte pas ces fonctionnalités, il est aussi possible de les émuler depuis l'hyperviseur en utilisant la virtualisation de la *MMU* (figure 4.9). Ce mécanisme permet à l'hyperviseur de configurer un deuxième niveau de traduction d'adresse, qui vient compléter celui mis en place par le noyau. Cette approche a notamment été implémenté dans la solution XNPro [Nordholz 2015] pour l'architecture ARMv7-A. Dans leurs travaux,

4. Ou *control data*, e.g. pointeurs de fonction et adresses de retour.
5. Ou *non-control data*, e.g. données de configuration, d'identification (uid ...), ou décisionnelles.
6. *WxN* (Write eXecute Never) dans la nomenclature ARM.
7. *PxN* (Privilege eXecute Never) dans la nomenclature ARM.

le deuxième niveau de traduction d'adresse contient des tables interdisant l'exécution des adresses de l'espace utilisateur (resp. du noyau) lorsque le noyau (resp. les logiciels utilisateur) est en cours d'exécution. La vérification de ces droits d'accès par l'hyperviseur, ou leur émulation, permet donc effectivement de se prémunir des attaques **A1**.

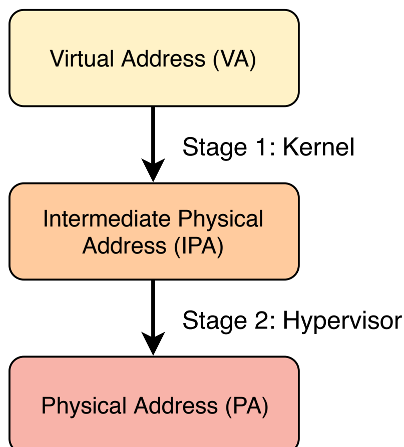


Figure 4.9 – Virtualisation de la MMU par un hyperviseur

Cependant, il est nécessaire d'employer des mécanismes de sécurité plus complexes pour se prémunir des attaques **A2** et **A3**. En effet, les contre-mesures bénéficiant du plus d'intérêt de la part de la communauté scientifique nécessitent une intégration importante dans le noyau. Une première contre-mesure, nommée *Kernel Address Space Layout Randomization* (KASLR), consiste à empêcher un attaquant de connaître par avance la topologie mémoire du noyau, information essentielle à la majorité de ces attaques.

Par ailleurs, deux autres contre-mesures peuvent être utilisées de manière complémentaire : *Control-flow Integrity* (CFI) [PaX Team 2015, Moreira 2017] et *Dataflow Integrity* (DFI) [Song 2016]. La première propose d'empêcher les attaques **A2** en étudiant les flux d'exécutions légitimes du noyau. Ainsi, tout flux non nominal (e.g. issu d'une attaque `rop`) sera détecté. Concernant la contre-mesure *DFI*, son fonctionnement est assez similaire à celui du *CFI* excepté le fait qu'elle s'intéresse aux flux de données, et non d'exécutions, ciblant par conséquent les attaques **A3**. Ces deux solutions s'appuient sur la construction de graphes représentant les flux légitimes, ceux-ci étant générés statiquement à partir de l'analyse du code du noyau et de l'instrumentation du compilateur utilisé (gcc ou llvm). Le respect de ce graphe à l'exécution est ensuite assuré par divers mécanismes que nous ne détaillons pas dans ce manuscrit. Nous invitons le lecteur à consulter [Moreira 2017, Song 2016] pour plus d'informations.

En effet, nous avons choisi une autre approche pour notre architecture, puisque l'implémentation de contre-mesures similaires depuis un hyperviseur requerrait un travail d'introspection poussé qui serait contre productif, car générant un surcout en performances trop important. Nous avons donc choisi de prémunir notre architecture

de sécurité contre les effets de ces attaques plutôt que de chercher à empêcher l'occurrence de ces attaques en elles-même. Pour cela, l'*IH* est chargé de surveiller et d'assurer certains invariants du noyau Linux :

- La non désactivation des propriétés de sécurité du système définies par la configuration spécifique de certains registres du processeur.
- La non désactivation des moniteurs de sécurité, i.e. du *SCIH* et de *SELinux*.
- La bonne configuration des pages mémoire de la *MMU*.

En effet, si un de ces invariants n'est pas respecté, par exemple en conséquence d'une attaque `rop`, la voie est ouverte à des attaques plus classiques et faciles à mettre en œuvre. La protection de ces invariants revient donc à handicaper les attaquants qui devront trouver des cibles plus complexes à exploiter dans le noyau, et uniquement accessibles par des attaques **A2** ou **A3**. Cette approche a été expérimentée dans *Hytux* [Lacombe 2009], un hyperviseur pour architecture x86 s'assurant du respect de certaines propriétés de sécurité du noyau, appelées dans ces travaux *Kernel-Constrained Objects* (KCO).

4.4.3 Protection contre les attaques matérielles

Bien que les attaques en provenance du matériel soient moins courantes que les attaques logicielles, il est important de les considérer et de déployer des contre-mesures les concernant. Ces attaques peuvent provenir par exemple, d'un périphérique malveillant installé lors de la fabrication du smartphone, ou bien ajouté ensuite dans le cas des smartphones personnalisables [8]. Il est aussi possible que le firmware d'un des périphériques, dont les sources sont fermées en très grande majorité, soit compromis d'une quelconque façon.

Sur les architectures ARM, les périphériques sont interconnectés par des bus *Advanced Microcontroller Bus Architecture* (AMBA) et communiquent via *Memory-mapped I/O* (MMIO). La mémoire centrale et les registres accessibles sont associés à des valeurs d'adresses mémoire, et les entrées/sorties (I/O) sont donc associées à des demandes de lecture/écriture mémoire. De plus, les périphériques étant « maîtres » du bus d'interconnexion peuvent effectuer des accès directs à la mémoire physique, ou aux registres *mappés* en mémoire, via le *Direct Memory Access* (DMA). Sans protection adéquate, les périphériques malveillants peuvent utiliser ce vecteur de communication pour lire ou écrire où ils le souhaitent dans l'espace mémoire, par exemple dans la mémoire du noyau, attaque souvent référencée par le terme « attaque *DMA* ».

Pour se prémunir contre ce genre d'attaques, un composant nommé *Input–Output Memory Management Unit* (IO-MMU) peut être utilisé. Ce composant matériel a pour but de virtualiser l'espace mémoire des périphériques du système.

8. Du smartphone entièrement modulaire comme le *Project Ara*, maintenant abandonné par Google, à l'ajout/remplacement de périphériques possible sur certains smartphones (Fairphone 2, LG G5, Moto Z ...)

Son fonctionnement est analogue à celui de la *MMU* classique, dans le sens où il permet de définir une traduction d'adresses entre un espace mémoire virtuel et l'espace mémoire physique, ainsi que des droits d'accès associés à ces adresses. Concernant l'architecture ARMv8, l'*IO-MMU* est implémenté dans une ou plusieurs entités matérielles, appelées *System MMU* (SMMU), s'intercalant entre un ou plusieurs périphériques système et leur bus d'interconnexion. En outre, le *SMMU* est capable d'associer une table de traduction différente pour chaque périphérique sous son contrôle. Pour plus de détails sur ce composant matériel, nous invitons le lecteur à consulter [Averlant 2017a].

La bonne configuration de cet *IO-MMU*, effectuée depuis l'*IH*, permet de restreindre l'espace mémoire ou les droits d'accès de chacun des périphériques sur la mémoire centrale et les registres *mappés* en mémoire. Ainsi, il est possible de se protéger efficacement face aux attaques *DMA*.

4.4.4 Protection au démarrage et chaîne de confiance

Outre la protection de l'intégrité de notre solution à l'exécution, il est important de vérifier que celui-ci est correctement chargé au démarrage du smartphone, et au sein d'un environnement logiciel intègre.

Sur cet aspect, de nombreux progrès ont été effectués par le système d'exploitation Android qui possède plusieurs processus de vérification du logiciel devant être exécuté. Dans sa version 8.0, cette vérification est effectuée en trois étapes.

La première étape est effectuée par le *firmware* stocké dans la *rom* du smartphone, qui constitue le premier maillon de la chaîne de confiance. Le principe d'une telle chaîne de confiance est de prouver l'intégrité et l'authenticité du logiciel par un ensemble de calculs cryptographiques [9]. En suivant ce principe, chaque maillon de la chaîne est chargé de vérifier le suivant. Ainsi, le *firmware* vérifie les différents logiciels privilégiés s'exécutant avant le lancement de l'*OS* [10]. Le dernier de ces logiciels, l'*Unified Extensible Firmware Interface* (UEFI), poursuit cette tâche en vérifiant l'image du noyau Linux et sa configuration. Enfin, ce dernier est chargé de vérifier les systèmes de fichiers contenant le framework Android par une fonctionnalité du noyau nommée *dm-verity*. Ce processus complet est appelé *Verified Boot* [11]. Les composants logiciels de notre architecture, bénéficieront par la même occasion de ce processus de vérification :

- L'*IH* est vérifié par le *firmware*.
- Le *SCIH* est vérifié par l'*UEFI*.
- Le *PHS* et le *PDA* sont vérifiés par la procédure *dm-verity*.

Cependant, nous réalisons au démarrage des vérifications d'intégrité supplémentaires depuis l'*IH* pour ajouter une couche de confiance ne dépendant pas du

9. Signatures, certificats et empreintes cryptographiques.
10. Plus de détails ici : `https://github.com/ARM-software/arm-trusted-firmware/blob/master/docs/trusted-board-boot.rst`.
11. `https://source.android.com/security/verifiedboot`.

reste de l'environnement Android. Ainsi, le code du *SCIH* doit être contrôlé avant que celui-ci puisse effectuer la modification de la table des appels systèmes. Cette procédure est déclenchée directement par le *SCIH* via l'utilisation d'un *hypercall*. Concernant le *PHS* une vérification de son intégrité sera également lancée lors de son enregistrement auprès du *SCIH*.

4.5 Conclusion

Dans ce chapitre, nous avons présenté la mise en œuvre de notre politique de sécurité par une architecture dédiée. Pour ce faire, nous avons premièrement défini les besoins et les contraintes imposées à la fois par l'environnement Android, ainsi que par la politique de sécurité en elle-même. Plus précisément, nous avons examiné les possibilités d'implémentation architecturales pour prendre en compte les objets dont la politique doit contrôler l'accès, les protections nécessaires pour se prémunir contre le modèle d'attaque envisagé, et ce de la manière la plus efficace possible.

À partir de cette analyse, nous avons décidé d'employer une architecture implémentée à plusieurs niveaux de privilèges et d'abstraction, ainsi appelée « multiniveau ». En effet, l'architecture proposée se compose d'une application Android, d'un service implémenté dans le framework, d'un module du noyau Linux, et d'un hyperviseur *bare-metal*.

Cette architecture devant appliquer la politique de sécurité détaillée dans le chapitre 3, nous avons expliqué quelles sont les missions associées à chacun de ces composants pour la récupération des *Secure Manifests*, l'extraction des règles qu'ils décrivent, ainsi que l'activation et la mise en application de ces dernières, sans oublier la gestion des conflits.

Enfin, et parce-qu'il est nécessaire de s'assurer de l'intégrité globale de la solution présentée et ce à tout instant pour veiller à la bonne application des règles qu'elle met en œuvre, nous avons décrit les différentes protections implémentées par l'hyperviseur *bare-metal* et destinées à prémunir notre architecture contre différentes attaques logicielles ou matérielles.

Cette architecture, représentant notre deuxième contribution, a fait l'objet d'une implémentation partielle au sein d'un prototype qui est présenté dans le chapitre suivant.

Chapitre 5

Prototype et performances

Sommaire

5.1 Plateformes d'expérimentation **84**
5.1.1 La plateforme de développement *96Boards hikey* 84
5.1.2 L'émulateur Android 85
5.2 Implémentation **85**
5.2.1 AOSP, base logicielle d'Android 86
5.2.2 Implémentation du *PDA* 87
5.2.3 Implémentation du *PHS* 90
5.2.4 Implémentation du *SCIH* 93
5.2.5 Implémentation de l'*IH* 96
5.2.6 Limitations du prototype 100
5.3 Mesures de performances **101**
5.3.1 Performances des actions atomiques de notre prototype . . . 102
5.3.2 Microbenchmarks 105
5.3.3 Macrobenchmarks 106
5.4 Validation par scénario **107**
5.4.1 Scénario n°1 108
5.4.2 Scénario n°2 110
5.4.3 Mise en relation avec les attaques récentes 113
5.5 Conclusion **113**

Après avoir abordé les détails de notre architecture de sécurité, ce dernier chapitre présente son implémentation sous forme de prototype. Une première partie expose dans un premier temps les plateformes d'expérimentation que nous avons utilisées pour créer ce prototype, et les spécificités inhérentes à chacune d'entre elles. Une deuxième partie détaille quant à elle les ajouts et modifications logicielles effectuées pour la création de ce prototype, et précise le processus de développement et de test mis en place. Ensuite, une troisième partie est dédiée à l'évaluation des performances de notre prototype via l'exécution de plusieurs campagnes de *benchmarks.* Enfin, une quatrième partie s'attache à effectuer une validation synthétique de notre prototype à travers l'exécution de scénarios simples mais transposables à des attaques concrètes, dans le but de prouver l'efficacité de notre prototype face aux menaces issues de notre modèle d'attaque.

5.1 Plateformes d'expérimentation

Pour les besoins de développement et de test de notre prototype, nous avons choisi deux plateformes d'expérimentation. La première est une plateforme de développement dont le matériel embarqué correspond à celui que l'on peut retrouver au sein des smartphones, et ceci dans le but de pouvoir coller le plus possible à la cible de notre solution. La deuxième est une plateforme virtuelle nous permettant d'effectuer plus facilement les tests et ainsi accélérer le développement de notre prototype.

5.1.1 La plateforme de développement *96Boards hikey*

Cette première plateforme, en photo figure 5.1, suit les spécifications du programme « 96Boards » conçues par l'association *Linaro*, et a été créée par l'entreprise *LeMaker* [1]. Elle possède l'avantage de correspondre assez sensiblement aux caractéristiques attendues d'un smartphone. En effet, on retrouve un SoC *HiSilicon Kirin 620* embarquant 8 coeurs *ARM Cortex-A53* d'architecture *ARMv8-A*, cadencés à 1.2GHz. De plus, le SoC inclut un GPU *ARM Mali-450 (MP4)*. Concernant la mémoire vive embarquée, il s'agit d'une puce de 2Go de LPDDR3. Le stockage est quant à lui assuré par une puce de 8Go d'eMMC ainsi que par la présence d'un port de carte microSDHC.

Par ailleurs, la connectivité est assurée par une puce *Texas Instruments WL1835MOD* fournissant le support du Wi-Fi 4 (802.11n) et du Bluetooth 4.0 (y compris le Bluetooth Low Energy). Un port HDMI permet de connecter la plateforme à un moniteur ou une TV. De plus, trois ports USB 2.0 (deux type A et un type Micro-B) permettent de connecter les périphériques d'entrée (clavier et souris) ainsi que de profiter des capacités de débogage des protocoles *Android Debug Bridge* (ADB) et *Fastboot* depuis un terminal du PC auquel la plateforme serait connectée. En outre, deux connecteurs d'extensions exposent des possibilités d'interactions supplémentaires avec la plateforme. Nous utilisons notamment l'une d'entre elles pour s'interfacer avec la sortie *UART* de la plateforme à des fins de débogage. Enfin, un autre avantage important de cette plateforme repose sur le fait que les sources de son *bootloader* sont disponibles, et qu'il est possible de changer celui présent de base sur la plateforme. C'est cet aspect qui nous a permis de développer le composant *IH* de notre solution.

Toutefois, certains périphériques que l'on s'attend à retrouver dans un smartphone manquent à l'appel. En effet, la plateforme ne contient ni puce GPS, ni modem WAN, ni puce NFC, ni quelconque capteur (appareil photo, microphone, capteurs de mouvements et environnementaux). C'est aussi un point que nous allons chercher à traiter avec la deuxième plateforme.

1. `https://www.96boards.org/product/hikey/`

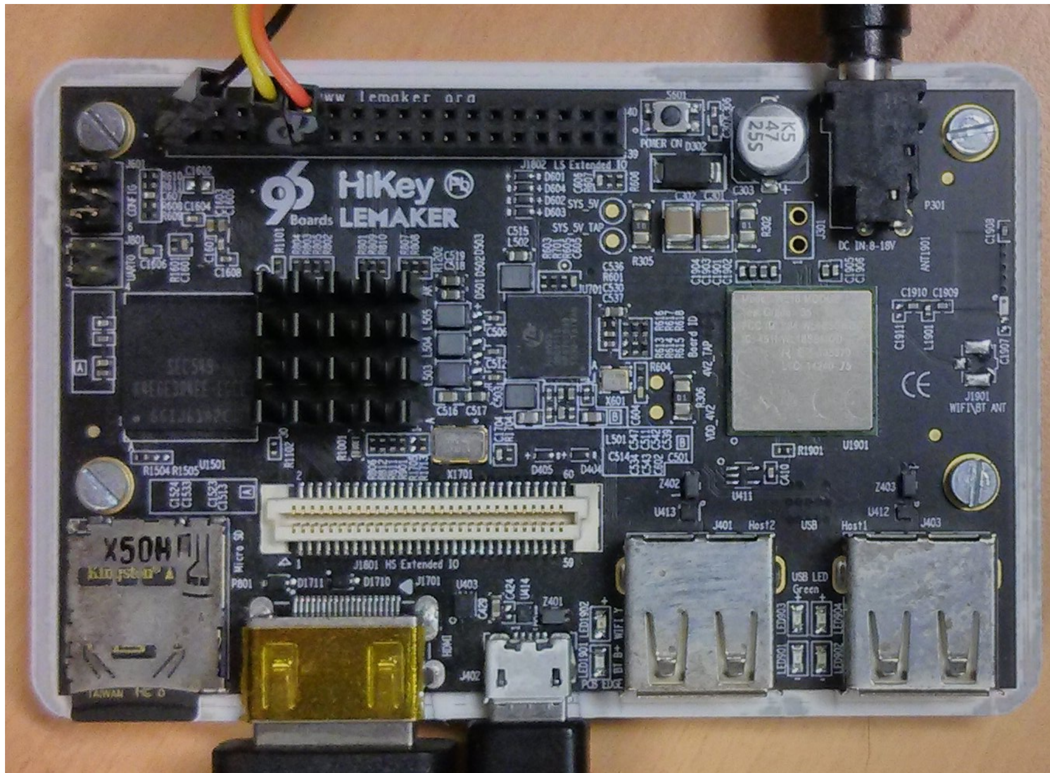


FIGURE 5.1 – La plateforme de développement *96Boards hikey*

5.1.2 L'émulateur Android

La deuxième plateforme considérée n'est autre que l'émulateur Android dont le code source est disponible conjointement avec celui de l'OS Android. L'utilisation de ce type de plateforme émulée permet donc d'envisager de tester notre solution en virtualisant certains des périphériques manquant à la plateforme de développement. D'autre part, ce genre de plateforme logicielle nous a permis de faire un prototypage et un développement plus rapide des composants de notre solution pour les couches les moins privilégiées : le *PDA*, le *PHS* et le *SCIH*. En outre, il est plus facile d'automatiser des tests sur cette plateforme émulée, ainsi que de capturer les différentes entrées et sorties, pour enregistrer des démonstrations par exemple.

Concernant les caractéristiques de la machine hébergeant l'émulateur, celle-ci possède un processeur *Intel Xeon E3-1270* comprenant quatre coeurs cadencés à 3.6GHz, ainsi que 16 Go de mémoire vive. De plus, le système d'exploitation hôte est une distribution Ubuntu dans sa version 17.10.

5.2 Implémentation

L'architecture de notre solution à implémenter étant découpée en différents composants hébergés à différents niveaux de l'écosystème Android (figure 4.1), leur im-

plémentation découle de modifications et d'ajouts d'une même base logicielle : le code source de base Android nommé *AOSP* (Android Open Source Project). Nous allons donc commencer la description de cette implémentation par la partie commune aux deux plateformes de tests, et en présentant dans un premier temps cette base logicielle commune, sa configuration et sa compilation. Nous rentrerons ensuite dans les détails d'implémentation de la solution, en prenant les composants par ordre décroissant de niveau d'abstraction, avant d'entamer les parties spécifiques à chaque plateforme.

5.2.1 AOSP, base logicielle d'Android

Le code source d'Android, aussi appelé *AOSP*, est en réalité constitué d'une multitude de dépôts git, ou *projets*, accessibles depuis l'url : `https://android.googlesource.com`. Ces projets mis bout-à-bout constituent ainsi l'ensemble du code open-source d'Android. Cette approche modulaire permet à la fois un développement séparé des différents composants d'Android, dont le nombre total de lignes de code dépasse les 10 millions, mais aussi d'ajouter aisément a posteriori des projets complémentaires (propriétaires ou open source) pour le support des différents smartphones. Par ailleurs, il est ainsi possible de ne sélectionner que des dépôts nécessaires à la compilation finale afin d'économiser de la bande passante et de l'espace de stockage. En effet, tous les dépôts ne sont pas nécessaires à la compilation d'une version d'Android installable (aussi appelé *ROM*) pour un smartphone : les dépôts *AOSP* servent à la fois à la compilation de *ROM* pour chaque appareil (parties communes et parties spécifiques aux appareils), mais aussi à la compilation d'outils (outils du SDK, compilateurs, firmwares, noyau Linux, émulateur ...). De plus, de nombreux appareils ne supportent pas la dernière version d'Android : on parle de fragmentation des versions d'Android. Ainsi, la version des différents projets utilisés dépendent de la version finale d'Android à compiler.

Pour mettre en place l'ensemble des projets nécessaires à la compilation d'une *ROM*, d'une version spécifique, et pour un appareil particulier, on s'appuie sur un outil nommé *repo*. Cet outil a pour but de récupérer, configurer et extraire l'ensemble des fichiers des projets nécessaires à la compilation. Pour ce faire, il s'appuie sur des fichiers xml, appelés *manifests*, contenant les informations nécessaires : 1) la liste des dépôts Git à récupérer ; 2) la branche et/ou le numéro de commit correspondant à la version de chaque projet ; 3) la destination d'extraction de chacun de ces projets. Ainsi, l'arbre des sources nécessaires à la compilation d'Android est reconstitué.

Une fois que tous les fichiers sources sont extraits au bon endroit et à la bonne révision, la suite logique est la compilation de ces sources en une image système pouvant être chargée sur l'appareil en question. Cette étape est, là encore, automatisée et effectuée par plusieurs outils. L'outil *lunch* permettant de préparer la compilation en vérifiant les dépendances et générant les données de configuration utilisées pour compiler l'image système requise (architecture du processeur, ...). Ensuite, la compilation est effectuée de concert avec le gestionnaire de dépendance *ninja*, les

compilateurs (C, C++, Java) fournis dans l'arborescence *AOSP*, et l'analyse des nombreux fichiers de configuration.

Pour notre plateforme *96Board hikey*, nous avons bénéficié du support natif de celle-ci dans les dépôts officiels d'AOSP. Les instructions de compilations sont disponibles à l'adresse `https://source.android.com/setup/build/devices#620hikey`. Nous avons généré l'image système utilisée par l'émulateur de manière équivalente.

Par ailleurs, pour reproduire un environnement applicatif plus fidèle, nous avons souhaité intégrer les applications Google, aussi appelées *gapps*. Celles-ci n'étant pas open-source, elles ne sont donc pas présentes dans les sources d'*AOSP*. Nous avons donc rajouté un dépôt Git contenant ces applications, nommé *opengapps* [2], afin qu'elles soient intégrées à la ROM de nos plateformes de test.

5.2.2 Implémentation du *PDA*

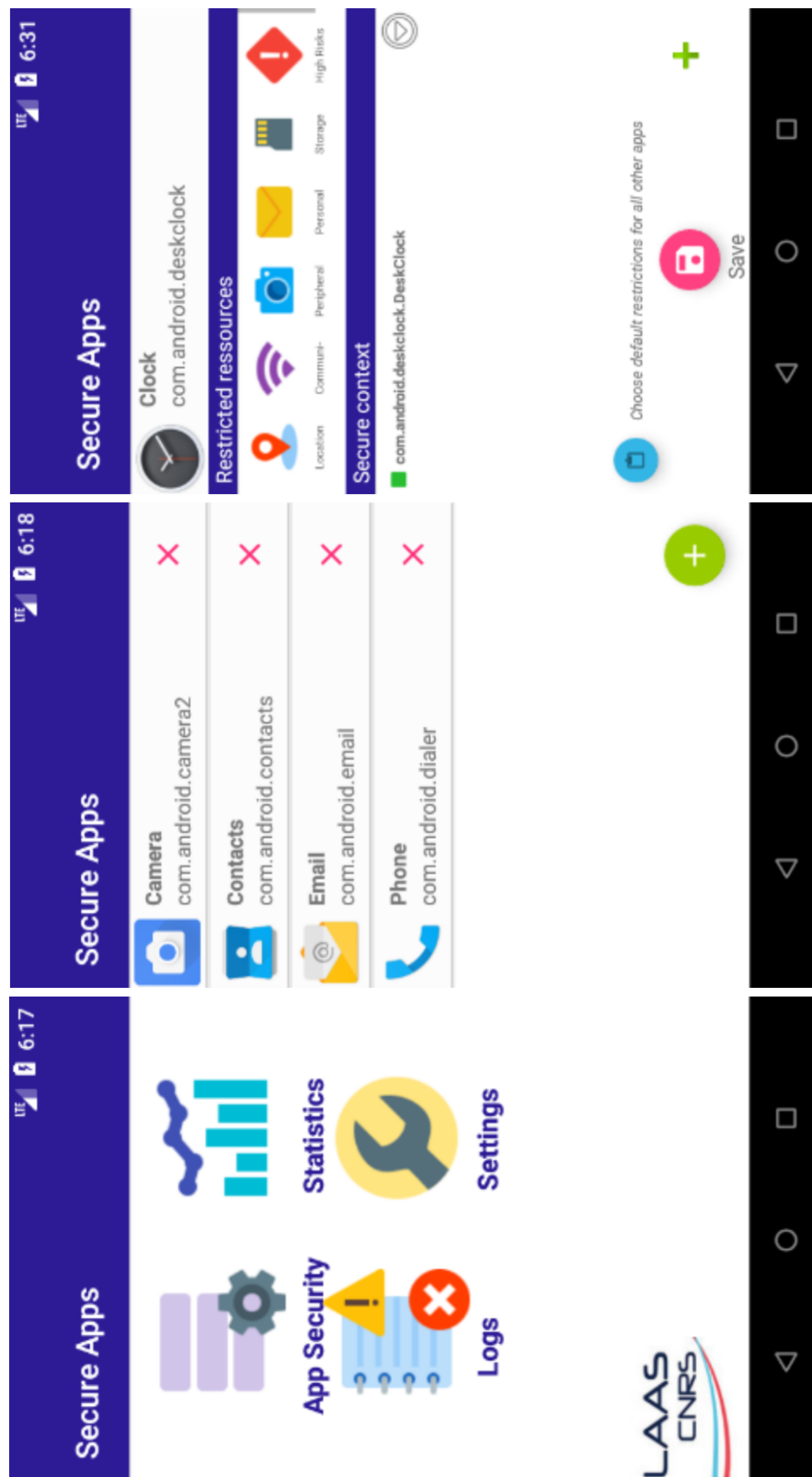
Le premier composant de notre prototype, l'application de définition des politiques *PDA*, a été développé à l'aide de l'IDE *Android Studio*. Cette application ne requiert aucune permission particulière, si ce n'est celle d'être notifiée au démarrage du smartphone pour pouvoir interagir au plus tôt avec le *PHS* pour s'occuper de la gestion des conflits. Quelques captures d'écran de l'application, commentées ci-dessous, sont fournies pour décrire les différentes fonctionnalités implémentées pour ce prototype. Ainsi, la figure 5.2a correspond à l'écran d'accueil présenté à l'utilisateur lorsque l'application est lancée depuis le *launcher*. Depuis cette activité, l'utilisateur peut choisir d'accéder à :

- La liste des *Secure Manifests* actifs (figure 5.2b).
- Les statistiques d'utilisation des ressources (voir section 3.4).
- Les logs des conflits antérieurs, pouvant être consultés pour ajuster les paramètres des *Secure Manifests*.
- Une liste de paramètres permettant de modifier l'apparence de l'application.

L'activité énumérant les *Secure Manifests* actifs permet aussi de modifier ceux-ci ou d'en ajouter pour n'importe quelle application installée sur le système. Ainsi, la figure 5.2c expose l'activité à partir de laquelle l'utilisateur peut observer les détails d'un *Secure Manifest* et de modifier les règles de celui-ci. Cette modification passe notamment par une activité permettant de choisir les ressources concernées par les règles d'utilisation exclusive (figure 5.3a).

Enfin, comme expliqué dans la section 3.3, la gestion des conflits est présentée à l'utilisateur de deux manières différentes en fonction du type de conflit rencontré : par popup pour les conflits nécessitant l'action immédiate de l'utilisateur (figure 5.3b), ou bien par notifications pour les conflits ne nécessitant pas d'interaction prioritaire de la part de l'utilisateur (figure 5.3c).

2. Consulter : `https://github.com/opengapps/aosp_build`

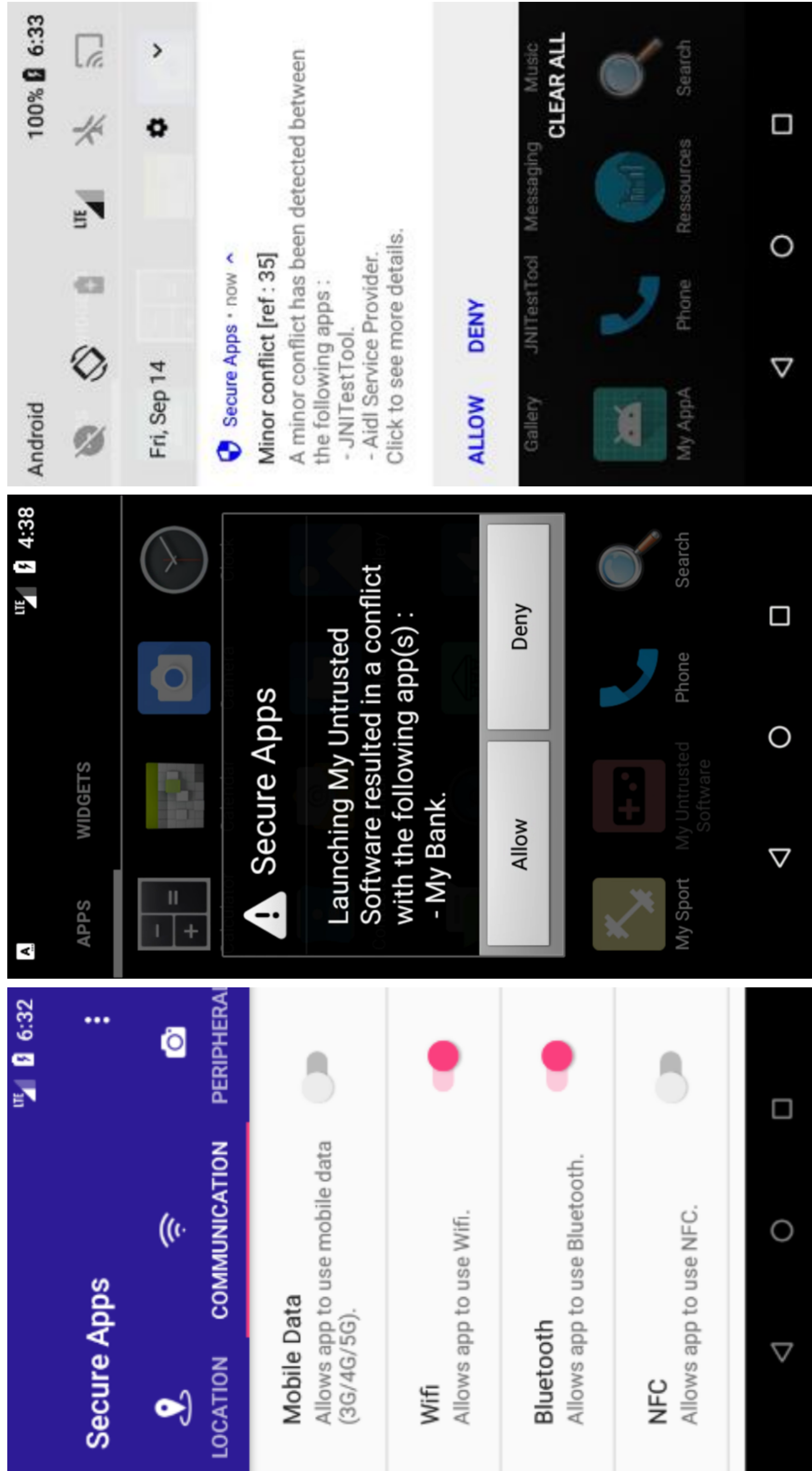


(a) Écran d'accueil

(b) Liste des *Secure Manifests*

(c) Affichage d'un *Secure Manifest*

FIGURE 5.2 – *PDA* : écran d'accueil et liste des *Secure Manifests*



(a) Configuration des ressources (b) Popup de conflit (c) Notification de conflit

FIGURE 5.3 – *PDA* : configuration des *Secure Manifests* et gestion des conflits

5.2.3 Implémentation du *PHS*

Le *PDA* est ensuite relié au composant central de notre architecture : le service de gestion des politiques ou *PHS*. Comme précisé dans le chapitre 4, ce composant s'intègre au coeur même du framework Android, en particulier en s'interfaçant avec le service système *AMS*, puisque les objectifs multiples devant être remplis par le *PHS* nécessitent cette proximité. Pour rappel, celui-ci est chargé de :

- Récupérer et garder en mémoire les règles issues des *Secure Manifests* associés aux applications installées.
- Surveiller quel composant de chaque application est lancé pour déterminer quelles sont les règles actives de la politique.
- Appliquer ces règles directement sur les intents.
- Définir les règles devant être appliquées par le *SCIH* sur les appels systèmes.
- Se tenir à l'écoute des évènements capturés par le *SCIH*.
- S'occuper de la gestion des conflits en collaboration avec le *PDA*.

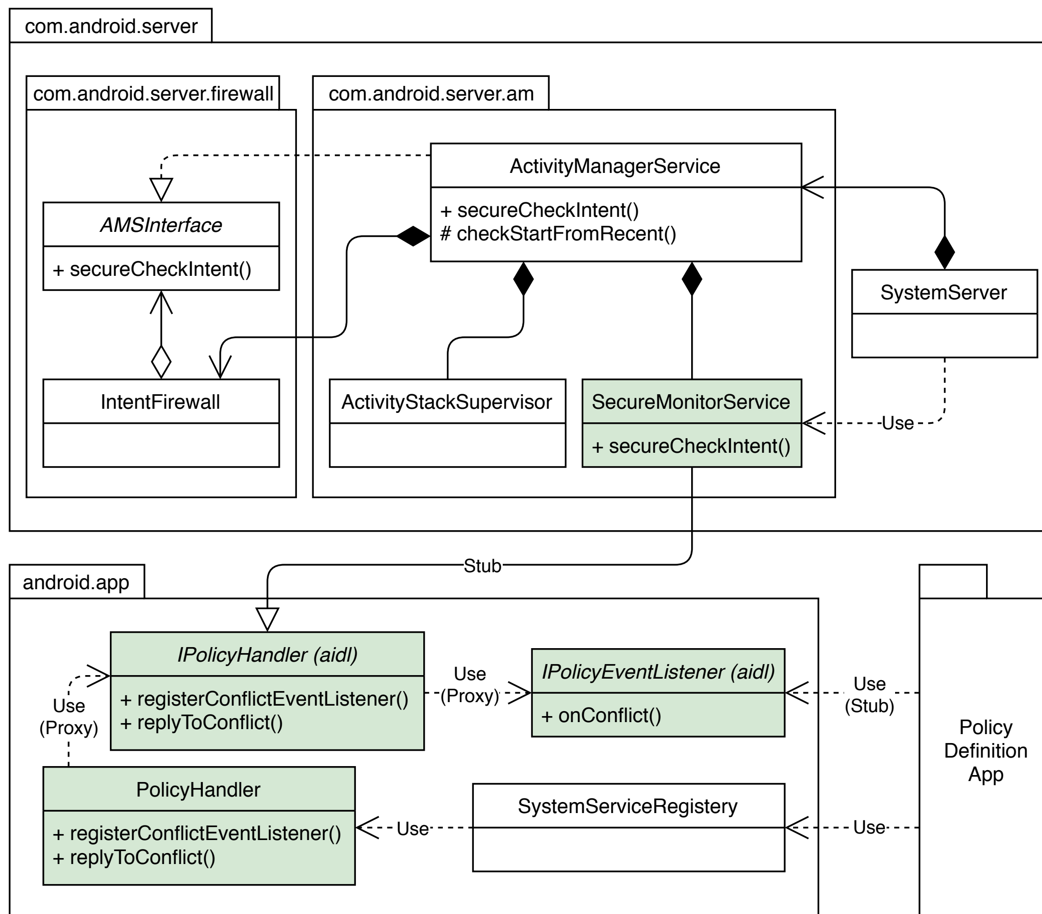


FIGURE 5.4 – Diagramme de classe UML simplifié du *PHS*

Pour accomplir ces missions, nous avons donc décomposé notre composant *PHS* en trois sous-éléments :

- Un nouveau service système, nommé « Policy Handler », implémenté dans une nouvelle classe Java dédiée (`SecureMonitorService`).
- Un ensemble de méthodes natives associées à cette classe permettant de communiquer avec le SCIH.
- Deux interfaces *AIDL* (voir §1.2.2) permettant une communication asynchrone avec le *PDA* pour la gestion des conflits.

Par ailleurs, il a été nécessaire de modifier certaines classes Java du framework pour, d'une part intégrer les différents éléments du *PHS* à l'existant, et d'autre part remplir les multiples objectifs qui incombent à ce composant de notre architecture. Cette intégration peut être visualisée sur la figure 5.4, diagramme de classes UML modélisant les éléments constitutifs du *PHS* (en vert sur la figure) et leurs dépendances.

Nous pouvons diviser ce diagramme de classe en deux sous-parties. La partie haute, consacrée au package `com.android.server`, décrit les changements apportés au framework concernant le nouveau service système et son intégration. La partie basse, dédiée au package `android.app`, définit les changements apportés à l'API Android, et introduit les modifications nécessaires à la communication entre le *PDA* et le *PHS*. Ces modifications sont abordées de manière plus détaillée dans la suite de cette section.

5.2.3.1 Lecture des *Secure Manifests* et gestion des conflits

L'ensemble des *Secure Manifests* est stocké dans l'un des répertoires privés du *PDA*. La lecture de ces fichiers XML est donc effectuée dès le démarrage du *PHS* depuis ce dossier particulier, inaccessible depuis les autres applications. Les règles issues de ces *Secure Manifests* sont ensuite stockées en mémoire. Lors de ce processus, le nom de chaque application est remplacé par son UID associé pour accélérer les traitements subséquents. Il en est de même pour les ressources dont les dénominations sont remplacées par des identifiants spécifiques ; le *PHS* possédant un tableau associant, à chacun de ces identifiants, une liste de références [3] vers des objets du système. De plus, le *PHS* s'enregistre auprès du framework pour que celui-ci le notifie en cas de modification du contenu du répertoire stockant les *Secure Manifests*. Ainsi, tout ajout, suppression ou modification effectuées sur ces fichiers est intégré au fur et à mesure par le *PHS*.

La gestion des conflits, permise par la communication avec le *PDA*, est rendue possible par une modification de l'API Android. Cette modification consiste en l'ajout de deux interfaces *AIDL* spécifiant : d'une part, les méthodes du service « Policy Handler » pouvant être invoquées depuis le *PDA* (interface `IPolicyHandler`) ; et d'autre part, le prototype de fonctions de rappel pouvant être implémentées par

3. E.g. Binder Node, descripteurs de fichiers, sockets ...

le *PDA* pour que celui-ci puisse être notifié de manière asynchrone par le *PHS* lors de l'occurrence d'un conflit (interface `IPolicyEventListener`).

5.2.3.2 Intégration avec l'*AMS* et capture des *Intents*

Notre nouveau service est lancé au démarrage du smartphone par le `SystemServer`, et ce après le lancement de l'*AMS*, ce dernier faisant partie des services fondamentaux du système. Par ailleurs, il est important de noter que ces deux services sont exécutés dans le processus du *SystemServer*, et font partie d'un même package. Ainsi, le *PHS* est capable de consulter les données internes de l'*AMS*. Tout particulièrement, il consulte la liste des `ProcessRecord` de l'*AMS*, qui stocke les informations relatives à toutes les applications en cours d'exécution. Parmi ces informations, on trouve par exemple l'état des composants de ces applications (du point de vue de leur cycle de vie), permettant d'adapter les traitements effectués par le *PHS* comme explicité dans la section 3.3.3.

De plus, la proximité de notre service avec l'AMS lui permet de bénéficier d'une fonctionnalité de capture des *intents* pré-existante. Cette dernière, utilisée dans les travaux de Yagemann et al. [Yagemann 2016] est implémentée dans la classe `IntentFirewall`. Nous avons donc effectué quelques petites modifications à cette classe, ainsi qu'à l'AMS, afin que le *PHS* soit informé lors de la réception d'*intents* et qu'il puisse ensuite filtrer ceux-ci selon les règles actives de la politique.

Cependant, après plusieurs tests, nous nous sommes rendus compte que cette méthode de capture ne prenait pas en compte les *intents* « internes » générés directement par des composants du `SystemServer`. C'est notamment le cas lorsqu'un utilisateur souhaite lancer une activité depuis la liste des applications récentes. Nous avons donc instrumenté la classe responsable de cette action (`ActivityStackSupervisor`) pour capturer l'évènement manquant.

5.2.3.3 Code natif et communication avec le *SCIH*

Afin de rendre possible la communication entre le *PHS* et le *SCIH*, il a été nécessaire de développer une partie de notre service en code natif, via l'utilisation de *JNI*. Il est en effet impossible d'effectuer un appel système directement depuis du code Java. Ainsi, cette section de code écrite en C++, invocable depuis la classe Java `SecureMonitorService`, s'occupe de préparer les données à envoyer au *SCIH* puis de réaliser les *ioctl* vers le pilote de périphérique « en mode caractère » exposé par celui-ci.

Au total, l'implémentation du *PHS* au sein de notre prototype aura représenté environ 2000 lignes de code (1850 loc de Java, 150 loc de C++). En outre, les modifications apportées aux classes existantes du Framework Android ne représentent que 150 lignes de code. En termes de comparaison, ce dernier est constitué d'environ 26700 lignes de Java et 1100 de C++.

5.2.4 Implémentation du *SCIH*

Le gestionnaire d'interception des appels systèmes (*SCIH*), complète le *PHS* en s'assurant de la bonne application des règles actives de la politique concernant les communications de bas-niveau (transactions binder et autres appels système). Pour ce faire, ce troisième composant a été implémenté en tant qu'un *Linux Kernel Module* (LKM). Celui-ci n'est pas directement intégré au noyau, mais est chargé au démarrage du smartphone, à l'aide de la commande `insmod`, afin de faciliter le prototypage : il n'y a ainsi pas besoin de regénérer une image du noyau Linux à chaque modification du code du *SCIH*. Pour mener à bien les missions qui lui sont confiées, c'est-à-dire mettre en œuvre la politique de sécurité sur les communications hors de portée du *PHS* et avertir ce dernier de l'occurrence de certains évènements pour la gestion des conflits, le *SCIH* doit être capable de :

- Intercepter efficacement les appels systèmes issus des applications pour savoir si ceux-ci sont en accord avec les règles actives de la politique.
- Collecter des renseignements supplémentaires sur le contenu des appels systèmes ainsi interceptés. Principalement, il est nécessaire d'interroger le driver du binder pour traiter les transactions lui étant adressées.
- Implémenter un pilote de périphérique « en mode caractère », utilisé pour la communication entre le *PHS* et le *SCIH*, et rendre possible son utilisation.

En définitive, l'implémentation de ces fonctionnalités aura nécessité environ 1300 lignes de code. Celles-ci et les défis d'implémentation qui leurs sont associés sont ainsi présentés dans les sous-sections suivantes.

5.2.4.1 Interception des appels système

Afin d'effectuer l'interception des appels systèmes, tâche nécessaire à l'application des règles de notre politique par le *SCIH*, nous avons choisi de positionner cette interception au plus tôt dans la chaîne de traitement de ces communications par le noyau Linux. Pour ce faire, nous modifions directement la « table des appels système » en remplaçant les entrées associées aux appels systèmes que nous souhaitons contrôler par des fonctions *wrapper*. Celles-ci sont ensuite chargées d'appliquer le contrôle d'accès de notre politique, avant de passer la main aux fonctions ainsi remplacées.

Cependant, afin de modifier cette table des appels systèmes, nous avons tout d'abord besoin de connaître son emplacement en mémoire centrale. Cette information est obtenue à l'exécution par l'utilisation de la table des symboles du noyau via la fonction `kallsyms_lookup_name()`. Ensuite, nous sauvegardons les entrées de cette table que nous souhaitons remplacer, avant de les écraser par nos fonctions *wrapper*. Toutefois, il est nécessaire de désactiver temporairement la protection en écriture mise en place par le noyau pour pouvoir effectuer cette modification [4].

4. La « table des appels système » est stockée dans la section `.rodata` du noyau, et est donc protégée en écriture à l'exécution.

La désactivation de cette protection est accomplie différemment pour les deux plateformes d'expérimentation. Pour l'émulateur, possédant une architecture *x86_64*, il est seulement nécessaire de modifier le bit 16 (`WP`) du registre de contrôle `CR0`. Pour notre plateforme matérielle ARMv8, il est en revanche nécessaire de modifier les règles d'accès de la page de la MMU associée à l'adresse de la table des appels systèmes. Le listing A.1 présente le code utilisé dans notre prototype pour accomplir cette tâche.

5.2.4.2 Collecte d'information et introspection du driver du Binder

Outre la capacité d'interception des appels systèmes, il est nécessaire de déterminer si ceux-ci sont conformes aux règles actives de la politique. Dans ce but, le *SCIH* se charge d'examiner les détails de ces appels systèmes afin d'autoriser ou non leur réalisation. Pour la plupart de ces appels systèmes, il est possible de recueillir les informations nécessaires à cette prise de décision en interrogeant directement le noyau. Les fonctions ainsi invoquées permettent par exemple de récupérer l'*UID* de l'application ayant émis l'appel système, ou bien de collecter des informations sur un fichier défini par un *descripteur de fichier*. Étant donné que les règles d'interdiction envoyées par le *PHS* au *SCIH* sont décrites selon un couple `<PID, ressource_noyau>`, il est ensuite aisé de mettre en application ces règles une fois ces informations recueillies. Cependant, les transactions binder, communications particulières à Android reposant sur l'appel système `ioctl`, nécessitent un traitement plus poussé de la part du *SCIH*. Ce traitement est décrit dans la figure 5.5.

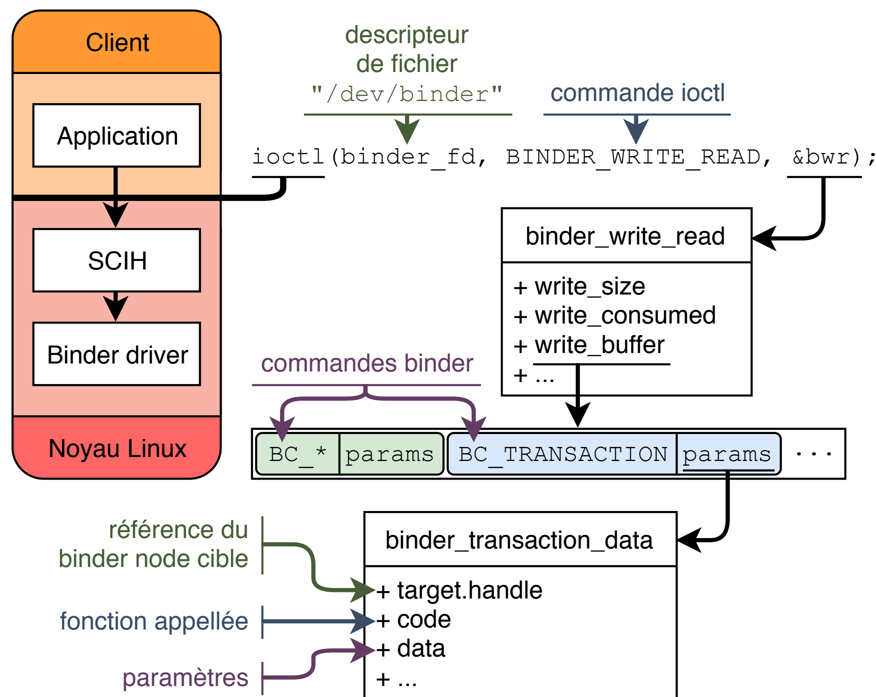


FIGURE 5.5 – Introspection réalisée par le SCIH sur les transactions binder

Cette figure présente les détails de l'encapsulation réalisée lors de l'envoi d'une transaction binder. Cette transaction est ainsi stockée dans une structure de donnée de type `binder_transaction_data` dans laquelle on peut retrouver une référence identifiant le *Binder Node* (BN) destinataire de la transaction, le numéro de la fonction invoquée, ainsi que les paramètres sérialisés associés à cet appel de fonction. Cette structure prend place dans un buffer répertoriant la liste des commandes à effectuer par le driver du binder. Enfin, ce buffer est contenu dans la structure `binder_write_read` passé en paramètre de l'*ioctl*.

Le filtrage effectué par le *SCIH* sur les transactions binder repose donc sur l'identification du BN destinataire de la transaction. Toutefois, la référence récupérée dans chaque transaction binder, n'est pas un identifiant permettant à lui seul de désigner un BN particulier : cette référence « locale » n'a de sens que pour le processus qui l'utilise. Plus précisément, le driver du binder associe à chaque processus un tableau recensant l'ensemble des BN connus par celui-ci, et cette référence locale correspond à l'indice de ce tableau permettant de désigner le BN souhaité. Par ailleurs, ce tableau est initialisé, à la création du processus, avec le BN du *service manager* correspondant à la référence locale `0`. Ensuite, à chaque fois que le *service manager* est interrogé, une entrée correspondant au BN recherché est ajoutée au tableau.

De plus, chaque BN possède un identifiant global unique utilisée en interne par le binder. Cette référence « globale » est utilisée par le *PHS* dans les règles qu'il transmet au *SCIH* pour identifier un BN de manière unique et indépendante de chaque processus. Ainsi pour identifier le BN destinataire d'une transaction, il est nécessaire d'effectuer une recherche additionnelle dans la mémoire du driver du binder. Le listing A.2 présente plusieurs exemples d'une telle introspection, et la figure 5.6 décrit la topologie des structures ainsi rencontrées.

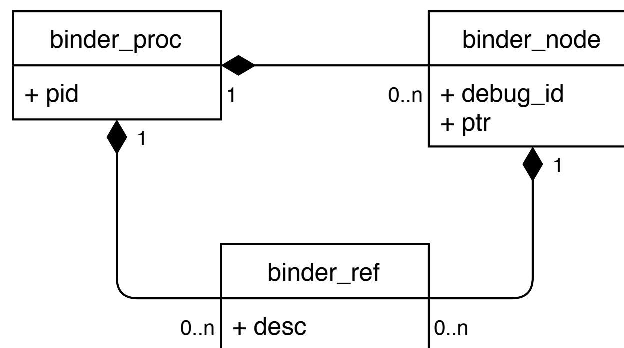


FIGURE 5.6 – Diagramme de classe UML des structures internes du Binder

Le point d'entrée de cette recherche correspond à une liste doublement chaînée contenant les informations sur chaque processus utilisant le binder, stockées dans des structures de type `binder_proc`. Le pointeur vers cette liste, et le mutex protégeant son accès, sont obtenues à travers la table des symboles, interrogée via la fonction `kallsyms_lookup_name`. Depuis cette structure, dont la majorité des champs sont

protégés par divers spinlock, il est possible de consulter :

- L'arbre bicolore répertoriant les BN (structures `binder_node`) appartenant au processus.
- L'arbre bicolore indexant les références (structures `binder_ref`) des BN connus [5] appartenant à d'autres processus.

C'est donc l'arbre des références qui nous intéresse pour notre recherche : le champ `desc` de la structure `binder_ref` correspond à la référence locale du BN pour le processus ; et dans la structure `binder_node` associée à cette référence on retrouve la référence globale contenue dans l'attribut `debug_id`.

5.2.4.3 Création d'un driver en mode caractère et règles SELinux

Pour rendre possible la communication entre le *PHS* et le *SCIH*, nous avons développé un pilote de périphérique « en mode caractère » exposant les fonctionnalités du *SCIH* à l'espace utilisateur via le nouveau fichier d'interface `/dev/hpm`. De manière similaire au driver du binder, ces fonctions sont rendues accessibles par l'utilisation de l'appel système `ioctl`. Cette interface permet ainsi au *PHS* :

- De s'enregistrer auprès du *SCIH* en tant que gestionnaire de la politique.
- D'ajouter ou supprimer une règle devant être appliquée par le *SCIH*.
- De récupérer des détails et références sur certains objets du noyau.
- De se mettre en attente d'un évènement (via un thread dédié).

La première fonction est invoquée dès le démarrage du *SCIH* afin d'enregistrer le *PHS* comme la seule entité capable d'invoquer les autres fonctions du *SCIH*, et le reste des fonctions est nécessaire pour le bon fonctionnement de l'architecture.

Par ailleurs, il a été nécessaire de modifier les règles SELinux afin d'autoriser l'utilisation de ce nouveau fichier par le *PHS*. De plus, dans le cadre de ce prototype, nous avons ajouté la capacité au PHS d'effectuer l'instruction `insmod` pour charger le module du *SCIH* au démarrage.

5.2.5 Implémentation de l'*IH*

L'implémentation de l'hyperviseur de contrôle d'intégrité (*IH*), dernier composant de notre architecture, a été uniquement effectuée sur notre plateforme matérielle *96Board hikey*. La base de ce composant consiste en un hyperviseur *bare-metal*, qui s'exécute par conséquent à un niveau de privilège matériel supérieur à celui de l'OS. Dans le cas de notre plateforme, possédant un CPU d'architecture ARMv8-A, le niveau de privilège *EL2* (voir chapitre 1) est approprié pour ce type de logiciel. Cependant, comme pour tout logiciel si bas niveau, le développement de l'*IH* et son insertion dans le firmware de la plateforme a constitué un challenge qui est expliqué dans les paragraphes suivants.

5. Partagés par le *service manager* ou par *intents* précédemment reçus.

5.2.5.1 Outils de développement et spécificités de la plateforme

Afin de créer un hyperviseur pour cette plateforme, il est nécessaire de posséder les outils de compilation adéquats. Pour cela, on utilise une *chaîne de cross-compilation* car la cible des binaires n'est pas la même que la machine chargée de la compilation, que ce soit au niveau architectural (jeu d'instructions différents) ou au niveau du système d'exploitation (bibliothèque système et format du binaire différents). Dans notre cas, l'hôte de la compilation est un ordinateur d'architecture *x86_64* exécutant un système d'exploitation de type Gnu Linux, alors que la cible de la compilation possède une architecture *ARMv8-A* et ne s'exécute en collaboration avec aucun système d'exploitation (d'où l'appellation *bare-metal*). Nous avons utilisé des binaires du compilateur *gcc* (dans sa version 5.3) et des outils *binutils* pré-compilés pour l'architecture *ARMv8-A* 64-bit [6] disponibles sur un dépôt Git de l'association *Linaro* [7]. En outre, certaines options de compilations doivent être utilisées afin que le binaire respecte le format adéquat à son exécution (cf listing A.3).

Par ailleurs, il est nécessaire de comprendre le comportement du logiciel chargé sur la plateforme à son démarrage afin d'en déduire la démarche et les modifications à effectuer pour s'insérer dans cette chaîne de démarrage. En effet, contrairement aux plateformes Intel, il est impossible d'accéder au niveau de privilège adéquat pour l'hébergement d'un hyperviseur après cette phase de démarrage.

Cette chaîne de démarrage se décompose en plusieurs étapes de chargement de *bootloaders* consécutifs, tous constituant un même *firmware*. Sur notre plateforme matérielle, c'est l'implémentation de *firmware* par défaut qui est utilisée : l'*ARM Trusted Firmware* pour processeurs applicatifs (TF-A) [8]. Ce firmware *TF-A* est chargé de fournir une implémentation de référence du moniteur de sécurité de l'environnement *TrustZone*, s'exécutant avec les privilèges de niveau *EL3*. C'est donc à lui qu'incombe la modification du niveau de sécurité du processeur pour donner la main à l'OS s'exécutant dans le *Secure World*. Ainsi, le logiciel du *Normal World* peut bénéficier des fonctionnalités offertes par ce dernier (cf. figure 1.1). Un tel OS, qui est facultatif, peut être chargé sur la plateforme via l'implémentation de référence nommée *OP-TEE* [9]. En outre, *TF-A* s'occupe de l'initialisation des différents composants et périphériques de la plateforme matérielle, de la gestion de leur alimentation, ainsi que du *hotpluging* des coeurs du CPU. Une fois celui-ci démarré, il donne la main à l'*Unified Extensible Firmware Interface* (UEFI), logiciel successeur du *BIOS* servant d'interface entre le *firmware* et l'OS, et ce au niveau de privilège *EL2*. Là encore, c'est l'implémentation de référence *Tianocore EDK2* qui est utilisée pour notre plateforme [10]. Enfin, l'*UEFI* poursuit cette séquence de démarrage en donnant la main au noyau Linux en mode *EL2*.

6. L'architecture *ARMv8-A* possède deux modes d'exécution : *AArch64* le mode 64 bit standard, et *AArch32* le mode 32 bit offrant une certaine rétro-compatibilité avec l'architecture *ARMv7-A*.

7. Dépôt consultable à l'adresse : `https://android-git.linaro.org/platform/prebuilts/gcc/linux-x86/aarch64/aarch64-linux-android-5.3-linaro.git/`

8. Anciennement appelée *ATF*, site de référence : `https://www.trustedfirmware.org/`

9. `https://www.op-tee.org/`

10. `https://www.tianocore.org/`

Pour que l'*IH* puisse jouer son rôle d'hyperviseur, il est donc nécessaire de prendre le contrôle du mode *EL2* avant que le noyau Linux ne soit chargé, et ce afin de protéger son démarrage. Dans ce but, nous avons choisi d'initialiser notre hyperviseur juste avant le chargement de l'*UEFI*. En effet, l'*UEFI* est configurée pour pouvoir s'exécuter avec les privilèges *EL2* ou *EL1* en fonction des choix de configuration de la plateforme [11]. De plus, s'insérer pendant l'exécution de l'*UEFI* aurait rendu notre hyperviseur dépendant de l'*UEFI* pour son chargement. Notre approche permet ainsi d'éviter cette dépendance en plus de s'assurer que l'*IH* soit chargé le plus tôt possible au démarrage, réduisant ainsi le risque qu'une quelconque compromission se produise avant sa mise en place. La figure 5.7 illustre le processus de démarrage normal et sa modification pour le chargement de notre hyperviseur.

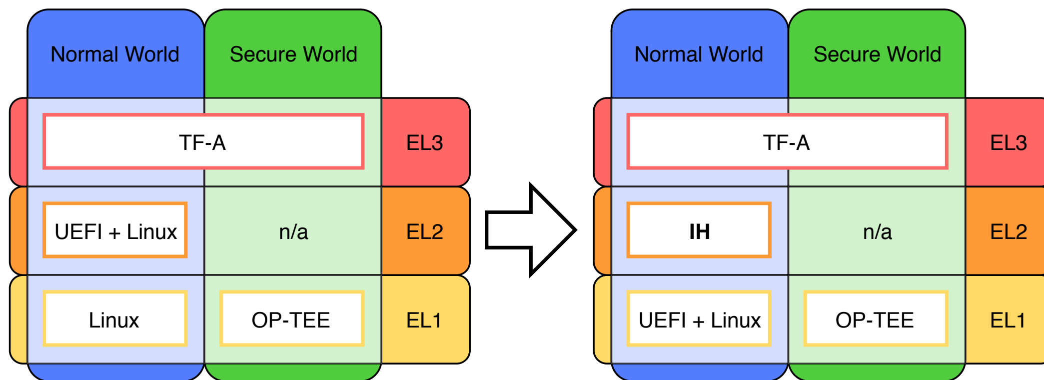


FIGURE 5.7 – Processus de démarrage et insertion de l'IH

5.2.5.2 Hyperviseur bare-metal pour l'architecture ARMv8-A

Comme nous l'avons précisé dans le chapitre 4, notre hyperviseur bare-metal doit être facilement auditable. Pour respecter cette propriété, nous avons donc choisi de n'inclure aucune dépendance externe, issues de bibliothèques par exemple. Ainsi le prototype de cet hyperviseur consiste en environ 1500 lignes de code, correspondant à la fois à la mise en œuvre d'un hyperviseur *bare-metal*, aux spécificités du composant *IH* (cf. chapitre 4), mais aussi une *libc* et un driver *uart* minimalistes utilisés pour le débogage. L'ensemble du code est ensuite compilé en un unique binaire. Les paragraphes suivants décrivent la configuration nécessaire du mode *EL2* pour accueillir un tel hyperviseur. En effet, cette initialisation doit être effectuée en plusieurs étapes.

Tout d'abord, il est nécessaire de définir les fonctions devant traiter les exceptions pouvant survenir à l'exécution. Ces exceptions peuvent être, par exemple issues d'interruptions logicielles ou matérielles, mais aussi résultant d'erreurs survenues à l'exécution. Plus spécifiquement, certaines de ces exceptions correspondent à la capture d'évènements en provenance du ou des systèmes d'exploitations virtualisés, ou

11. De nombreuses plateforme verrouillent l'utilisation du mode *EL2* depuis leurs firmwares.

de leurs applications. De telles exceptions peuvent être déclenchées explicitement par le logiciel virtualisé, via l'exécution d'instructions *hypercall*, utilisées notamment dans le cadre de la *paravirtualisation* [12]. Par ailleurs, un hyperviseur peut paramétrer une liste d'actions susceptibles de déclencher de telles exceptions. En voici une liste non exhaustive :

- Le non respect des permissions des pages mémoire de la MMU virtuelle.
- La lecture/écriture de registres système.
- L'exécution d'instructions spécifiques.

Notre hyperviseur doit donc, d'une part, définir les évènements pour lesquels il souhaite prendre la main sur le logiciel virtualisé ; et d'autre part, spécifier les fonctions de traitements pour chacun de ces évènements. Cette configuration passe par l'utilisation de plusieurs registres :

- `VBAR_EL2` (*Vector Base Address Register*) stocke l'adresse à laquelle sont définies les fonctions de gestion des exceptions pour le niveau de privilège *EL2*.
- `HCR_EL2` (*Hypervisor Configuration Register*) permet de configurer les paramètres associés au système d'exploitation virtualisé, dont les instructions et les accès à certains registres système capturés par l'hyperviseur.
- `CPTR_EL2` (*Architectural Feature Trap Register*) configure la capacité du système d'exploitation virtualisé à pouvoir accéder à certains registres liés à des fonctionnalités spécifiques du processeur (unité de *tracing*, de calculs en virgule flottante et *SIMD*).

Deuxièmement, il est nécessaire d'initialiser la MMU et de paramétrer les caches pour le mode *EL2* afin que l'hyperviseur puisse bénéficier de performances améliorées. Dans ce sens, la MMU est configurée pour que les adresses virtuelles correspondent à l'identique aux adresses physiques (*identity mapping*).

Enfin, notre hyperviseur doit protéger l'espace mémoire qu'il utilise pour son code, ses données et sa pile, des accès en provenance du logiciel virtualisé. Pour ce faire, il utilise la virtualisation de la MMU en initialisant le deuxième niveau de traduction d'adresse en *identity mapping* tout en enlevant les droits de lecture/écriture/exécution des pages mémoires à protéger. Il en est de même pour la mémoire associée au périphérique *uart*, utilisée par notre hyperviseur à des fins de débogage, sauf que celui-ci peut aussi être utilisé par le système d'exploitation virtualisé. Par conséquent, l'hyperviseur se charge également d'émuler les accès à ce périphérique.

5.2.5.3 Adaptations requises pour la plateforme matérielle

Comme pour tout logiciel bas niveau, un hyperviseur nécessite quelques adaptations afin d'être conforme et de pouvoir s'exécuter correctement sur une plateforme

12. On parle de *paravirtualisation* lorsque le logiciel sous-jacent a conscience d'être virtualisé, et peut ainsi invoquer directement certaines fonctionnalités implémentées par l'hyperviseur.

matérielle cible. Dans notre cas, ces adaptations ont dû être effectuées pour : 1) s'insérer correctement dans la chaîne de démarrage ; 2) Réserver une plage mémoire pour l'hyperviseur ; 3) Faciliter le prototypage et le débogage.

Comme expliqué précédemment, nous avons choisi d'effectuer le chargement de notre hyperviseur juste avant celui de l'*UEFI*. Dans ce but, nous avons mis au point un programme permettant de concaténer le code de notre hyperviseur à celui de l'*UEFI*, ainsi que de corriger les premières instructions du binaire *UEFI* afin que le code de l'hyperviseur soit exécuté avant celui de l'*UEFI*. Cette méthode permet de ne pas avoir à modifier le processus de démarrage : le *firmware* chargera à la fois notre hyperviseur et l'*UEFI* et ce sans aucune modification de son code ou de sa configuration.

Concernant le deuxième point, nous avons choisi de réserver une plage de mémoire physique comprise entre les adresses `0x3D00_0000` et `0x3DFF_FFFF` pour l'hyperviseur. Cependant il est nécessaire d'indiquer au logiciel virtualisé que cette zone mémoire n'est pas disponible pour son utilisation, faute de quoi il serait légitime de sa part de chercher à y accéder. Pour ce faire, nous avons modifié la liste des zones mémoires réservées de notre plateforme matérielle. Celles-ci sont déclarées, d'une part dans les fichiers de configuration de l'*UEFI*, et d'autre part dans le fichier *device tree* généré lors de la compilation du noyau Linux ; ce dernier étant un fichier binaire décrivant les paramètres matériels [13] associées à la plateforme cible de la compilation.

Enfin, pour éviter d'avoir à continuellement réécrire l'image *UEFI* en *rom*, nous avons créé un logiciel « trampoline » chargeant le code de notre hyperviseur depuis l'*UART*. Ce code est envoyé depuis la machine de compilation avec l'aide d'un programme python basé sur *miniterm.py* [14]. De cette manière nous avons pu gagner du temps lors du prototypage, tout en minimisant l'usure de la flash servant de *rom* sur notre plateforme. Par ailleurs le programme python nous est aussi utile pour afficher le texte de débogage de notre hyperviseur et du noyau Linux.

5.2.6 Limitations du prototype

Notre prototype étant avant tout une preuve de concept, il possède quelques limitations par rapport à l'ensemble des fonctionnalités présentées dans le chapitre 4, mais celles-ci n'impactent pas la validation d'une telle architecture.

Concernant le *PHS*, nous n'avons pas créé de permission système dédiée à protéger l'utilisation de l'API, exposée au *PDA*, de façon à ce qu'elle ne puisse être utilisée par d'autres applications. Dans le cadre d'une utilisation de notre solution dépassant la validation effectuée dans ce chapitre, il serait néanmoins nécessaire de développer cette fonctionnalité afin que seul le *PDA* soit autorisé à communiquer avec le *PHS* par cette API. Cependant, cette dernière n'étant pas connue des autres applications installées sur les plateformes de tests, cela n'impacte pas la validation

13. E.g. caractéristiques du CPU, périphériques embarqués et drivers associés, adresses mémoire des registres de ces périphériques, organisation globale de la mémoire ...

14. `https://github.com/pyserial/pyserial/blob/master/serial/tools/miniterm.py`

de notre prototype.

Le *SCIH*, quant à lui, n'est pas intégré à l'image du noyau mais est chargé après le démarrage par le *PHS*. De cette manière, il n'est pas nécessaire de compiler et de flasher l'image du noyau à chaque modification introduite, ce qui nous a permis de gagner beaucoup de temps lors de la création du prototype.

La deuxième limitation du *SCIH* repose sur le fait qu'il ne prend pas en compte les zones mémoires générées par le driver *ashmem*. Celles-ci peuvent en effet être partagées entre plusieurs applications en faisant transiter leurs descripteurs dans de simples transactions binder. Ainsi, lorsque la politique de sécurité interdit la communication entre un processus et un BN, seules les transactions binder subséquentes sont bloquées, toutefois les zones de mémoires partagées précédemment transmises restent utilisables. Afin de contrôler ce vecteur de communication, il aurait été nécessaire que le *SCIH* effectue une introspection dans le driver *ashmem* de manière similaire à ce qui a été fait pour le driver du binder.

Dernière limitation du *SCIH*, les règles qu'il met en œuvre pour filtrer l'accès aux ressources du binder opèrent à la granularité des BN. Or, il serait intéressant de pouvoir opérer à un niveau de granularité plus faible afin de filtrer l'utilisation de certaines fonctions associés à ces BN. De plus, cet ajout n'aurait probablement pas d'impact sur les performances, car il suffirait uniquement de consulter le champ `code` de la structure `binder_transaction_data` qui est déjà désencapsulée par le *SCIH*. Par ailleurs, l'absence de cette fonctionnalité nous empêche de bloquer efficacement l'accès à certaines ressources. C'est le cas par exemple pour la ressource « Affichage superposé » qui est implémentée dans le service système *WindowManager*. Or ce dernier est aussi responsable des fonctions d'affichages employées par les activités. Par conséquent, une restriction d'accès à ce service empêcherait tout affichage graphique de l'application.

Enfin l'*IH* n'implémente pas à l'heure actuelle les divers contrôles d'intégrité présentés dans la section 4.4. Cependant la faisabilité des différentes techniques présentées a été testée indépendemment du prototype. Il aurait été néanmoins intéressant de pouvoir mesurer précisément l'impact sur les performances de telles mesures. Toutefois, dans la suite de ce chapitre, les mesures de performances ont été effectuées sans *IH* activé.

5.3 Mesures de performances

Cette section est dédiée à la validation de l'efficacité de notre solution, un des objectifs de sa conception, à travers diverses mesures de performances réalisées sur notre prototype. Dans ce but, nous avons réalisé trois types de tests présentés dans les sous-sections suivantes. Ces tests de performance sont présentés par ordre croissant de niveau d'observation, en commençant par se concentrer sur les actions atomiques de notre prototype, puis en observant chaque communication applicative, et en finissant par examiner le système dans son ensemble.

5.3.1 Performances des actions atomiques de notre prototype

Premièrement, l'exécution de trois types de tests de performances est destiné à mesurer l'influence des différentes actions coûteuses de notre solution, soulignant ainsi l'impact que peut induire chaque processus élémentaire sur les performances du système. Pour chacun de ces tests, seules les applications utilisateur nécessaires aux tests sont préalablement lancées. Les mesures présentées sont obtenues par l'instrumentation du composant *PHS* de notre prototype.

Un premier ensemble de mesures recense le temps moyen de traitement des *intents* par le *PHS* afin de déterminer si ceux-ci sont conformes, ou non, aux règles actives de la politique de sécurité. Ce temps de traitement moyen dépend néanmoins du nombre de règles actives qu'il est nécessaire de parcourir pour analyser la validité de l'*intent* analysé. Ainsi, la figure 5.8 présente le coût moyen d'un tel algorithme, en fonction du nombre de règles actives. De plus, un intervalle de confiance (I_c) à 95%, en vert sur la figure, est associé à chacun des points du graphe afin de borner cette valeur moyenne en fonction des paramètres statistiques de l'échantillon de test. Les *intents* traités sont envoyés depuis une application de test invoquant une de ses propres activités. Les règles synthétiques chargées sont associées à plusieurs applications (de un à quatre), et on force la vérification de l'ensemble des règles pour chaque *intent* reçu. En outre, le nombre de règles testées est volontairement exagéré afin de montrer la capacité de mise à l'échelle de notre système pour les cas les plus extrêmes. Ainsi, lorsqu'on considère la mise en œuvre de 1000 règles, correspondant

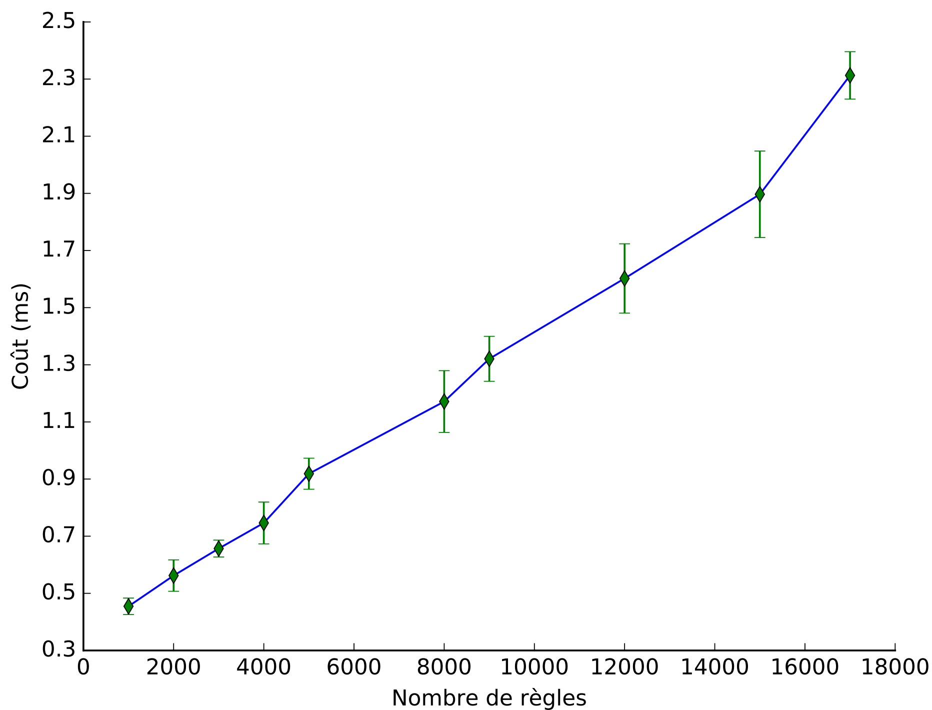


FIGURE 5.8 – Coût de l'application des règles à l'interception d'*intents*

par exemple à 50 règles pour 20 applications actives, le coût de la procédure est en moyenne compris entre 426 et 483μs avec un intervalle de confiance à 95%. Pour le cas extrême de 17000 règles actives, nous obtenons une moyenne comprise entre 2,230 et 2,395ms. Par ailleurs, ces mesures ont été effectuées sur l'émulateur Android, mettant en œuvre un ensemble de règle générées pour l'occasion, et pour lequel la durée moyenne d'exécution d'un *intent* est de 32ms. Le surcoût induit par notre solution est donc très faible sur l'interception des *intents*. Dernier point, nous n'avons pas fait apparaître ici, ni mesuré précisément, le temps de lecture de chaque *Secure Manifest* dont sont issues ces règles. En effet, celui-ci impacte surtout le temps de démarrage (de l'ordre d'une ou plusieurs secondes), et n'influe que très peu sur le système par la suite (uniquement lorsque l'utilisateur modifie un *Secure Manifest* via le PDA).

Un deuxième groupe de mesures concerne le coût de la vérification périodique permettant au *PHS* de détecter la fermeture d'applications en observant l'évolution de l'état de leurs composants. Cette tâche, lancée en arrière plan toutes les 500ms, effectue une lecture des `ProcessRecord` associés à chaque application en cours d'exécution. La durée moyenne d'un tel processus, en fonction du nombre d'applications actives, est présentée dans la figure 5.9. Il est important de noter qu'une dizaine d'applications système sont actives en permanence et tournent en arrière plan pour fournir divers services, c'est pourquoi la figure recense les résultats à partir de 18 applications (système et utilisateur confondues). De plus, comme sur la figure précédente, un intervalle de confiance à 95% permet de borner chaque mesure afin

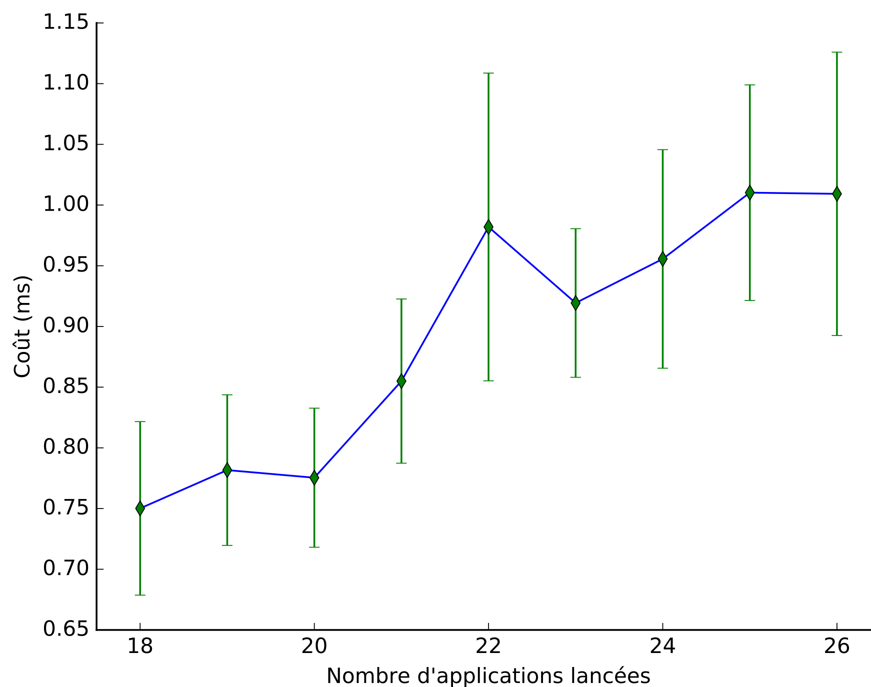


FIGURE 5.9 – Coût de la vérification périodique de la fermeture des applications

d'estimer plus correctement celui-ci en fonction des paramètres de l'échantillon recueilli. Par ailleurs, le nombre d'applications pouvant être lancées simultanément dépend énormément des capacités en mémoire vive de l'appareil et de l'agressivité du système *LMK* de celui-ci. Toutefois, on peut estimer par interpolation que ce surcoût ne devrait pas dépasser les 2,5ms pour 50 applications utilisateur actives. En outre, ces résultats sont issus de mesures effectuées sur l'émulateur Android exécutant un ensemble d'applications développées par nos soins pour l'occasion.

Enfin, un troisième lot de mesures évalue le surcoût associé à l'interception des appels systèmes par le *SCIH*. Celles-ci ont été effectuées sur la plateforme matérielle *hikey* en quantifiant la durée moyenne de l'exécution des appels systèmes, et ce avec ou sans l'interception active. Plus précisément, le temps mesuré correspond à l'envoi de 1 ou 10 appels systèmes `ioctl` depuis le code Java du *PHS*. Ces appels systèmes ciblent une fonction du pilote en mode caractère implémenté par le *SCIH* qui affiche une chaîne de caractères dans le log du noyau. Les résultats issus de ces tests peuvent être visualisés dans le tableau 5.1. Par ailleurs, nous voulons nous assurer que le calcul de surcoût, issu de la différence entre le temps nominal et celui obtenu avec notre solution, ait du sens d'un point de vue statistique, et qu'il ne soit pas le résultat d'une erreur d'estimation. Ainsi, nous avons voulu majorer cette différence en prenant en compte la marge d'erreur associée aux tests effectués. Plus précisément, nous avons calculé l'intervalle de confiance (I_c) à 95% associé à la moyenne de chaque test ; Puis, nous avons évalué la différence entre les deux moyennes (système actif (a) ou inactif (i)) en prenant en compte cet intervalle de confiance, nous donnant la formule :

$$\Delta I_c = (moyenne_a + I_{ca}) - (moyenne_i - I_{ci})$$

Par conséquent, si on calcule de cette manière la différence entre les résultats avec et sans l'interception, on obtient :

- Un surcoût de 0,30% pour 1 appel système émis.
- Un surcoût de 0,02% pour 10 appels systèmes émis.

Ces résultats montrent donc que l'interception effectuée par le *SCIH* a un impact négligeable sur la rapidité d'exécution des appels systèmes.

Nombre d'appels systèmes	1		10	
État de l'interception	Inactive	Active	Inactive	Active
Moyenne (ms)	6,8553	6,8621	58,2429	58.2214
I_c à 95% (μs)	6,8	6,7	17,1	16,3
ΔI_c	20μs $\leftrightarrow$ 0,30%		12μs $\leftrightarrow$ 0,02%	

TABLE 5.1 – Surcoût de l'interception des appels système

5.3.2 Microbenchmarks

Un second groupe de résultats vise à indiquer le surcoût global de notre solution sur les communications élémentaires effectuées par les applications. Pour rappel, celles-ci correspondent : soit aux communications de haut niveau (*intents*) transitant par l'*AMS*, soit aux communications binder directes entre applications.

Pour cela, nous avons développé plusieurs applications de tests et automatisé leur lancement à l'aide de la commande `am`[15] invocable depuis un shell *adb*. L'ensemble de ces tests spécifiques aux actions de communications élémentaires effectuées, aussi appelées *microbenchmarks*, ont été réalisés exclusivement sur la plateforme de développement *hikey*, et ce avec et sans la présence de notre solution. Pour chaque expérimentation, les mesures présentées sont obtenues à partir de l'exécution de 100 communications (*intent* ou accès aux ressources) par les applications de test.

Une première mesure présente le surcoût de notre solution relatif à l'envoi d'un *intent* ciblant une application en cours d'exécution[16]. Cet *intent* cible une simple activité appartenant à une application de test. La durée de réalisation de celui-ci est reportée dans le tableau 5.2.

État du système	Inactif	Actif
Moyenne (ms)	90,67	90,99
I_c à 95% (ms)	0,50	0,52
ΔI_c	1,34ms ↔ 1,48%	

Table 5.2 – Surcoût pour l'exécution d'un *intent*

Il est important de noter que cette mesure a été effectuée avec seulement un petit nombre de règles (10) chargées dans le système, l'influence de ces dernières ayant été évaluée plus tôt. En tout cas, le surcoût associé à la transmission des *intents* est minime (1,48%).

Une deuxième mesure s'intéresse quant à elle au surcoût du système sur une communication binder. Pour ce faire, nous avons développé une application interrogeant le service système *WifiManager*, service parmi tant d'autres. À partir de celle-ci nous avons relevé trois valeurs de temps d'exécution associées aux fonctions :

F1 : `getSystemService()` interrogeant le *ServiceManager* pour récupérer la référence du *WifiManager*.

F2 : `getWifiState()` fonction du *WifiManager* renvoyant un entier.

F3 : `getConnectionInfo()` fonction du *WifiManager* renvoyant un objet sérialisé.

Effectuer des mesures sur des fonctions différentes du service (F2 et F3) nous permet de vérifier que le type de données véhiculées n'influe pas sur le temps de traitement supplémentaire associé à notre système. L'ensemble de ces mesures sont

15. `https://developer.android.com/studio/command-line/adb.html#am`.
16. Une application arrêtée devant préalablement être lancée par le système pour recevoir l'*intent*.

répertoriées dans le tableau 5.3 en considérant le cas où notre système est actif ou non. De plus, nous évaluons également l'influence de règles, chargées dans le *SCIH*, et restreignant la liste des possibles *BN* destinataires de l'application de test.

État du sys.	Inactif			Actif				
Fonction	F1	F2	F3	F1	F2	F3	F2 (r)	F3 (r)
Moyenne (ms)	3,776	1,017	2,431	3,755	0,987	2,282	70,503	71,132
I_c à 95% (μs)	157	66	157	161	63	139	659	339
ΔI_c (μs)	$\emptyset$			297	99	147	70210	69197
ΔI_c (%)	$\emptyset$			7,86	9,78	6,06	6905	2847

Table 5.3 – Surcoût pour l'exécution d'une transaction binder

Ces nombreux résultats montrent que lorsqu'aucune règle n'est associée à l'application émettrice de l'appel système, le surcoût induit par notre système est assez contenu (<10%). Celui-ci correspond majoritairement au travail de désencapsulation de la transaction binder que le *SCIH* doit effectuer pour obtenir les informations nécessaires à son traitement. En revanche, les résultats obtenus par notre prototype lorsque des règles sont chargées sont nettement moins bons. Ils indiquent en effet un surcoût d'environ 70ms quelle que soit la fonction invoquée (soit un surcoût de 6905% pour F2 et 2847% pour F3). Cependant, on peut associer cet écart à l'introspection de la mémoire du driver du binder qui est effectué, dans notre prototype, à chaque transaction binder pour obtenir la référence globale du BN cible nécessaire à la prise de décision du *SCIH*. En effet, une telle introspection repose sur l'exécution de plusieurs actions assez coûteuses telles que l'obtention d'un mutex et le verrouillage de plusieurs spinlocks. Cependant, il serait possible de supprimer un tel surcoût dans une version plus aboutie du prototype en implémentant un cache de traduction des références locales pour chaque processus. Ainsi, le processus de traduction ne serait effectué que pour chaque première transaction binder, et les suivantes ne seraient pas ainsi ralenties.

5.3.3 Macrobenchmarks

Pour compléter les tests précédents, nous avons souhaité évaluer l'impact du système sur les performances globales de la plateforme de tests *hikey*. Dans ce but, nous avons utilisé trois applications de *macrobenchmarks* : Geekbench v4.3.2 [17], Vellamo Mobile Benchmark v3.2.6 [18] et CF-Bench v1.3 [19]. Ces applications ont pour but premier de tester la performance de l'appareil sur lequel elles s'exécutent, effectuant diverses opérations de calculs, d'accès mémoire, et d'accès disque. Ensuite en fonction des performances relevées par ces applications sur ces tâches, un score est

17. `https://play.google.com/store/apps/details?id=com.primatelabs.geekbench`
18. `https://web.archive.org/web/20170115041859/https://play.google.com/store/apps/details?id=com.quicinc.vellamo`
19. `https://play.google.com/store/apps/details?id=eu.chainfire.cfbench`

associé à l'appareil. Par conséquent, nous avons choisi d'évaluer les performances de calcul de notre plateforme de test *hikey* avec ces applications, et ce avec ou sans notre système actif. Comme la très grande majorité des actions effectuées au cours de ces tests consistent en calculs divers et variés, et qu'aucune interaction avec les autres applications ou les services système n'est effectuée (à notre connaissance), il ne nous a pas paru utile d'effectuer plusieurs tests en faisant varier le nombre de règles chargées. La moyenne des scores obtenus sur cinq tests, pour chacune des trois applications, est recensé dans le tableau 5.4.

Application de benchmark	Nom du test	État du système	
		Inactif	Actif
Geekbench 4	Single-Core	557	559
	Multi-Core	2290	2219
Vellamo	Metal	685	693
	Mutithread	1678	1674
CF-Bench	Native	51539	51238
	Java	2624835	2629584
	Overall	1595516	1598245

Table 5.4 – Résultats des macrobenchmarks

D'après les résultats obtenus, on peut observer que notre système n'entraîne pas d'impact significatif sur les résultats de ces *macrobenchmarks*. Les mesures associées au système actif sont même quelques fois meilleures que celles associées au système inactif mais ceci est simplement dû au fait que nous avons réalisé peu d'expérimentations. L'information importante à retenir est que l'impact est réellement négligeable.

5.4 Validation par scénario

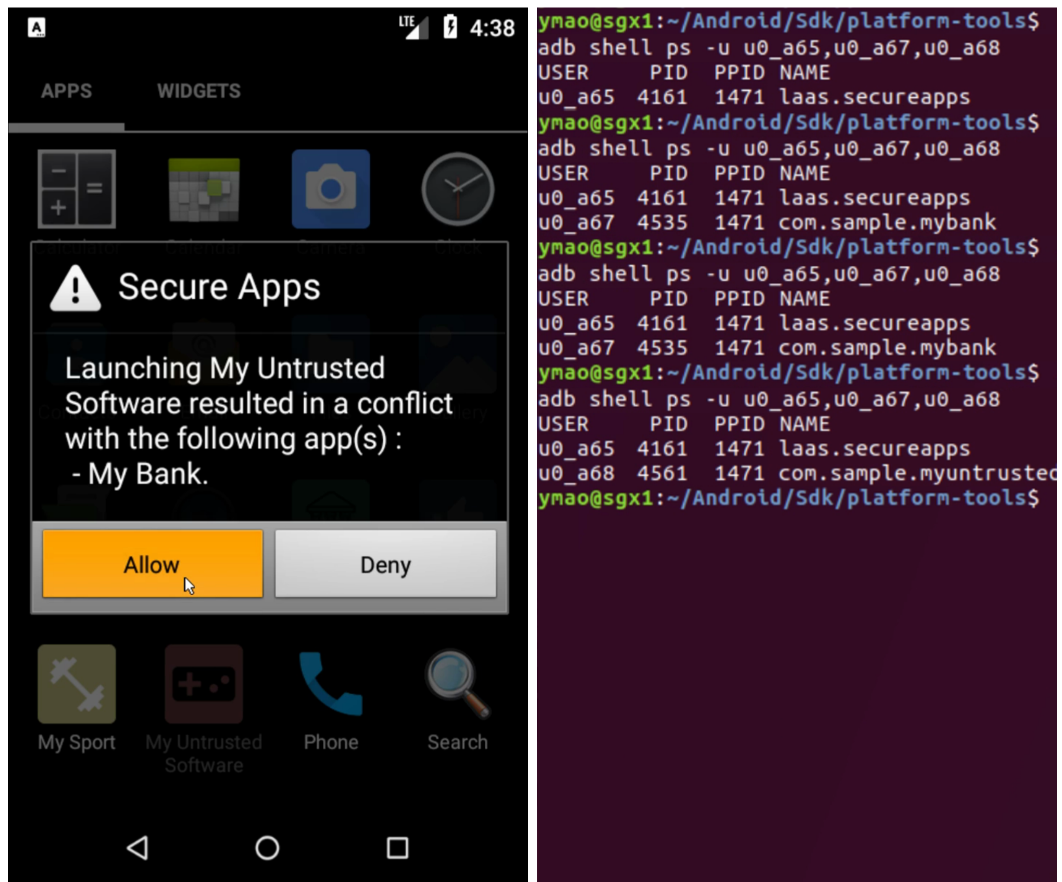
Dans cette section, nous souhaitons effectuer la validation de notre prototype à travers l'exécution de deux scénarios complémentaires mettant en lumière le bon fonctionnement de celui-ci. Le premier scénario concerne le blocage d'*intents* dans le cadre de l'application d'une règle d'interdiction d'exécution concurrente d'une application. Le second scénario représente, quant à lui, un exemple de blocage de communications binder issues de l'application d'une règle de restriction d'utilisation de ressource. La réalisation de ces scénarios, tous deux effectués sur l'émulateur Android et mettant en œuvre des applications génériques développées par nos soins, est décrite dans les paragraphes suivants.

5.4.1 Scénario n°1

Dans ce premier scénario, la politique de sécurité appliquée exprime la volonté de l'utilisateur de hiérarchiser la confiance qu'il possède envers les applications installées. L'exemple choisi considère ainsi deux applications : une application bancaire qui représente l'application critique aux yeux de l'utilisateur, et une application quelconque dont l'utilisateur est moins enclin à accorder sa confiance. L'utilisateur va ainsi configurer le *Secure Manifest* de l'application bancaire, présenté dans le listing 5, de manière à ne pas autoriser l'exécution concurrente de la seconde application.

```xml
<?xml version="1.0" encoding="utf-8"?>
<secure_manifest xmlns:secure="./secure.xsd">
    <pkg_info name="com.sample.mybank" pub_key="..." />
    <secure_context>
        <app_restriction>
            <app_list>
                <app name="com.sample.myuntrustedsoftware" />
            </app_list>
        </app_restriction>
    </secure_context>
</secure_manifest>
```

Listing 5 – *Secure Manifest* du scénario n°1



(a) Blocage de l'*intent* et popup de conflit (b) Applications lancées à chaque étape

FIGURE 5.10 – Scénario n°1

A partir d'un environnement où aucune des deux applications ne s'exécute initialement, le déroulement du scénario est ensuite le suivant :

- Ouverture de l'application bancaire « My Bank ».
- Tentative d'ouverture de la deuxième application « My Untrusted Software » depuis le *launcher*.
- L'*intent* issu de cette dernière action est intercepté et bloqué par le *PHS*.
- Une demande de résolution de conflits est envoyé au *PDA* qui affiche un popup à l'utilisateur (figure 5.10a).
- L'utilisateur autorise « My Untrusted Software » à s'exécuter et tente de la relancer.
- L'*intent* est cette fois-ci autorisé et entraîne la fermeture de « My Bank ».

Il est important de noter que si l'utilisateur n'avait pas autorisé l'exécution de l'application « My Untrusted Software », son lancement aurait à nouveau été bloqué à la dernière étape du scénario. Cependant, le cas opposé que nous considérons nous permet d'observer l'évolution qui doit être opérée par le système pour garantir le respect des règles. Ainsi, cette évolution peut être visualisée sur la figure 5.10b qui présente la liste des applications lancées à plusieurs étapes dans l'exécution du scénario :

- Initialement, seul le *PDA* s'exécute.
- Puis « My Bank » est lancée.
- La résolution du conflit n'a aucun impact sur cette liste.
- C'est seulement lorsque l'utilisateur souhaite relancer « My Untrusted Software » que l'application bancaire est fermée.

Ainsi la règle d'interdiction d'exécution concurrente des deux applications est respectée tout au long de l'exécution du scénario.

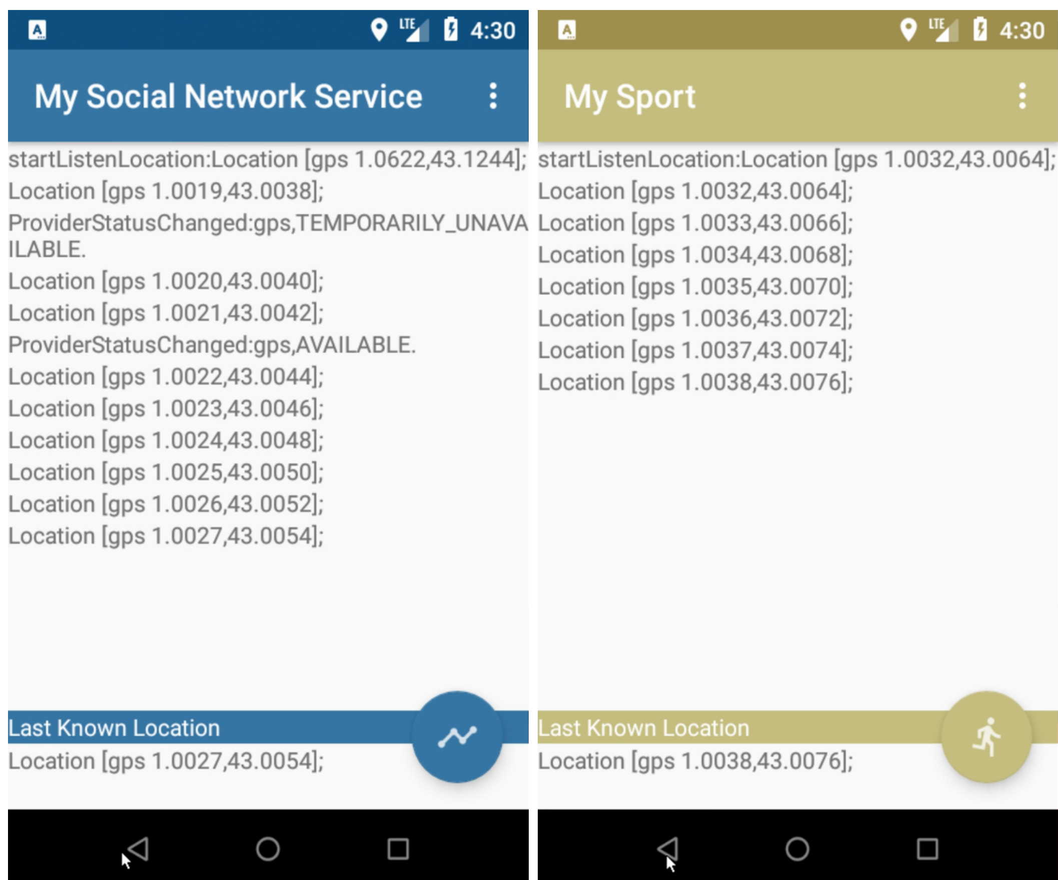
5.4.2 Scénario n°2

Dans ce second scénario, l'utilisateur choisit cette fois-ci de restreindre la liste des applications pouvant accéder à la ressource *GPS* en même temps que l'application de sport. De cette manière, celles-ci ne peuvent pas surveiller les déplacements de l'utilisateur lorsqu'il va faire son footing, par exemple. Nous avons choisi, pour ce scénario, de n'appliquer cette limitation qu'à l'application de réseau social. Ainsi, le *Secure Manifest* utilisé pour mettre en place cette restriction est présenté dans le listing 6.

Nous exécutons le scénario à partir d'un environnement vierge, c'est-à-dire ne possédant aucune application utilisateur, autre que le *PDA*, préalablement démarré. L'application de réseau social ne possède qu'une activité, tandis-que l'application de sport lance automatiquement un service en arrière plan dès son démarrage. Le déroulement du scénario est le suivant :

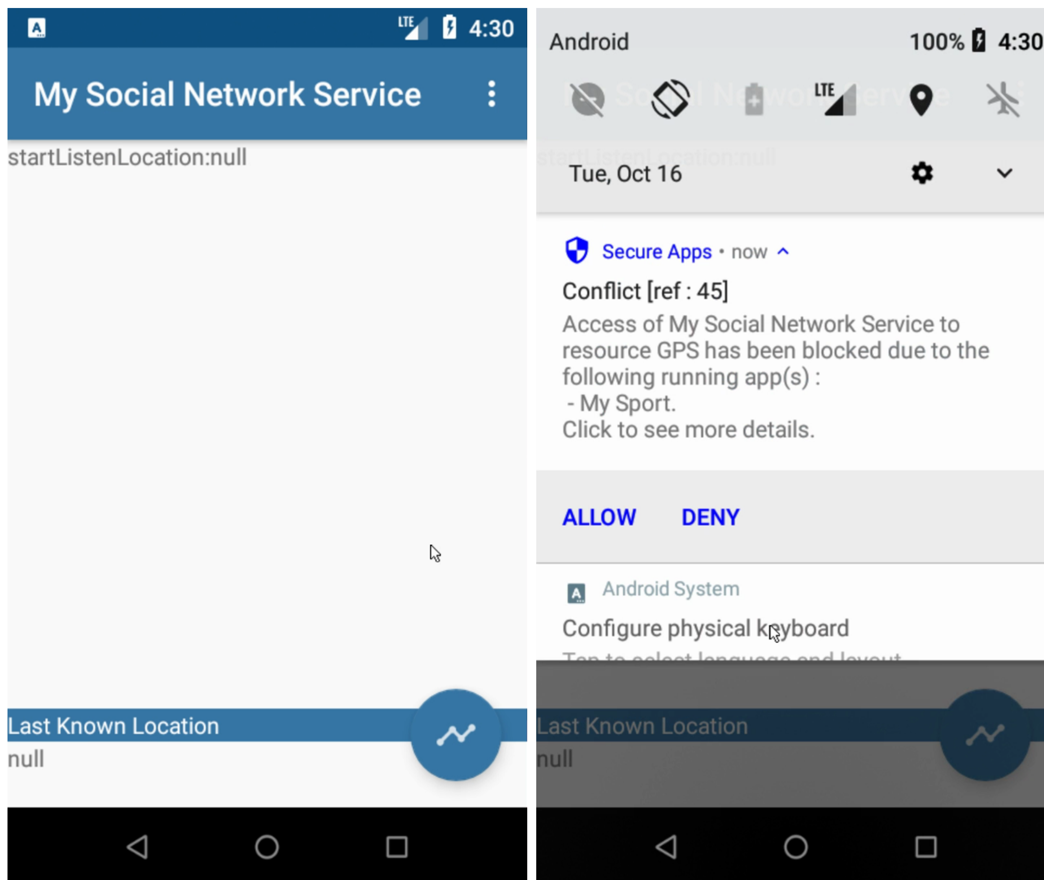
```xml
<?xml version="1.0" encoding="utf-8"?>
<secure_manifest xmlns:secure="./secure.xsd">
    <pkg_info name="com.sample.mysport" pub_key="..." />
    <secure_context>
        <app_restriction>
            <app_list>
                <app name="com.sample.mysocialnetwork" />
            </app_list>
            <resource_restriction>
                <resource name="gps" />
            </resource_restriction>
        </app_restriction>
    </secure_context>
</secure_manifest>
```

Listing 6 – *Secure Manifest* du scénario n°2



(a) Application de réseau social (b) Application de sport

FIGURE 5.11 – Scénario n°2 - Lancement des deux applications



(a) Accès au service GPS bloqué (b) Notification de conflit

FIGURE 5.12 – Scénario n°2 - Mise en œuvre des règles du *Secure Manifest*

- Ouverture de l'application de réseau social sur une activité affichant la position *GPS* en temps réel (figure 5.11a).
- Ouverture de l'application de sport sur une activité similaire (figure 5.11b) ; l'application de réseau social est fermée automatiquement suite à une résolution de conflit automatique [20].
- Ré-ouverture de l'application de réseau social ; lorsque celui-ci cherche à accéder à la ressource *GPS*, il voit sa transaction binder interceptée et bloquée par le *SCIH* (figure 5.12a).
- Suite à cette interception, le *SCIH* avertit le *PHS* qui détecte le conflit et informe l'utilisateur par le biais du *PDA* (figure 5.12b).

L'accès à la ressource *GPS* est donc effectivement bloqué pour l'application de réseau social jusqu'à ce que l'application de sport soit fermée, soit directement par l'utilisateur, soit indirectement par le *PHS* suite au choix effectué par l'utilisateur concernant le conflit.

20. Aucun composant de cette application n'étant actif.

5.4.3 Mise en relation avec les attaques récentes

Les deux scénarios présentés ci-dessus peuvent, par ailleurs, être mis en relation avec les attaques que nous avons relevés dans le chapitre 1 pour motiver nos travaux. Par exemple, si on considère l'attaque *Cloak and Dagger* [Fratantonio 2017] permettant notamment d'enregistrer les frappes clavier en utilisant l'affichage d'overlays ou les services d'accessibilité, il serait possible de la mitiger pour une application critique :

- En interdisant l'exécution simultanée de cette application comme pour le scénario n°1, l'utilisateur exprimant ainsi différents niveaux de confiance pour ces deux applications.
- En interdisant l'utilisation des ressources « affichage superposé » et « services d'automatisation » lorsque l'application critique s'exécute, de manière similaire au scénario n°2.

Autre exemple, certaines attaques permettent de détourner des communications Bluetooth [Naveed 2014] ou BLE [Sivakumaran 2018] avec un appareil connecté. Grâce à notre système, celles-ci peuvent être évitées par l'utilisation de règles restreignant l'accès simultané à ces ressources lorsque l'application source de la communication est utilisée ; ce type de règle étant illustré dans le scénario n°2.

Enfin, les attaques ayant pour but d'altérer la sécurité du système en lui-même, et ce quant bien même celles-ci utiliseraient des vulnérabilités dans le logiciel bas niveau comme *Dirty COW* [21], ou des vulnérabilités matérielles telles que les attaques *DMA*, seraient détectées par le composant *IH* de notre architecture. Malheureusement, l'implémentation de notre prototype n'ayant pas encore été finalisée, il ne nous a pas été possible de vérifier en pratique l'efficacité de cette contre-mesure.

5.5 Conclusion

Dans ce dernier chapitre, dédié à la présentation du prototype de notre solution, nous avons premièrement présenté les deux plateformes que nous avons employées pour le développement de celui-ci. Nous avons notamment utilisé la plateforme matérielle *hikey* dont les composants se rapprochent de ceux que l'on peut retrouver dans un smartphone moderne. Nous avons ensuite détaillé l'implémentation logicielle de chaque composant de notre architecture pour le prototype. Cette implémentation a par ailleurs nécessité un travail d'intégration conséquent, à la fois avec l'environnement Android et les deux plateformes de tests, qui est explicité. En outre, afin d'évaluer la pertinence du prototype, nous avons confronté ce dernier à un ensemble de tests de performance et de scénarios synthétiques. Les mesures de performances ont pu mettre en évidence que le prototype, malgré certaines limitations, n'impacte que très faiblement l'efficacité globale de l'appareil, n'occasionnant pas d'impact pouvant être ressenti par l'utilisateur. De plus, la bonne exécution des scénarios valide de manière synthétique la mise en œuvre adéquate des règles de

21. `https://nvd.nist.gov/vuln/detail/CVE-2016-5195`

notre politique par le prototype. Ces scénarios peuvent par ailleurs être transposés à des attaques réelles et représentatives de notre modèle d'attaque.

Conclusion

Les smartphones ont aujourd'hui véritablement envahi notre quotidien professionnel et familial. Ils offrent des services de plus en plus riches et variés en échange d'une manipulation de données de plus en plus importante, relevant bien souvent de la vie privée des utilisateurs. Par ailleurs, l'utilisateur d'un smartphone installe aujourd'hui de multiples applications dans lesquelles il est difficile d'avoir confiance, compte tenu du nombre important d'applications disponibles et du peu de contrôles de sécurité effectués sur ces applications, même si de nombreux efforts récents dans ce domaine commencent à porter leurs fruits [Rahul Mishra 2019]. Cette remarque est d'autant plus importante pour les utilisateurs n'utilisant pas le magasin d'application par défaut [Wang 2018]. Il est donc fondamental que les utilisateurs puissent pouvoir restreindre les privilèges des applications qu'ils utilisent et de préciser aussi finement que possible les données qu'elles peuvent manipuler, les périphériques qu'elles peuvent utiliser ou les interactions possibles avec les autres applications. Cette thèse se consacre à ce sujet et a proposé un modèle de gestion des droits d'accès dynamique pour smartphone équipés du système Android. Chaque application est dotée d'un « Secure Manifest », s'ajoutant au *manifeste* habituel des applications sous Android, dans lequel des règles de contrôle d'accès sont définies. Ces règles permettent de spécifier les conditions dans lesquelles l'application peut s'exécuter, en fonction du contexte d'exécution courant du smartphone, c'est-à-dire des autres applications qui s'exécutent et des ressources qu'elles manipulent.

Les principales contributions de cette thèse ont été les suivantes. Nous avons tout d'abord présenté cette nouvelle politique de sécurité, les éléments qui la composent, la structure des « Secure Manifest » et illustré son utilisation au travers de quelques exemples simples. Nous avons ensuite proposé une architecture logicielle permettant de mettre en œuvre les mécanismes de contrôle d'accès implémentant cette politique. Cette architecture logicielle est bâtie autour de trois composants principaux. Les deux premiers constituent des modifications réalisées au sein du framework Android et du noyau Linux, afin d'implanter et de vérifier les règles de contrôle d'accès qui sont spécifiées dans les « Secure Manifest ». Le troisième composant est un hyperviseur de sécurité *bare-metal* spécifiquement développé pour vérifier l'intégrité des mécanismes de contrôle d'accès implantés dans les couches logicielles supérieures. Enfin, nous avons décrit le développement d'un prototype partiel de cette architecture de sécurité, ainsi que des tests de performance associés et montré que notre architecture de sécurité est efficace sans pour autant dégrader les performances du système.

Concernant les perspectives d'amélioration des trois contributions présentées dans cette thèse, notre politique de sécurité pourrait voir ses fonctionnalités augmentées pour répondre à des problématiques parallèles. Il est possible d'envisager par exemple une liste plus fine des ressources considérées, ou bien d'ajouter des restrictions environnementales (spatiales ou temporelles) aux conditions de contexte

limitant la portée des règles comme de nombreuses solutions de l'état de l'art. Nous avons aussi abordé, dans le chapitre 3, l'utilité que pourrait avoir un site de *crowd-sourcing* pour la distribution des *Secure Manifests*. Toutefois la mise en place d'un tel site et l'implication de chacun de ses utilisateurs dans la conception des *Secure Manifests* posent de nombreuses questions qui devront être étudiées dans des travaux futurs. Une de ces questions nécessite d'étudier le problème de consensus résultant des informations saisies par les utilisateurs du site pour les intégrer aux *Secure Manifests*. De plus, notre architecture de sécurité pourrait facilement être adaptée pour répondre à ces nouveaux besoins. À propos de cette architecture, celle-ci pourrait intégrer un modèle prédictif permettant de résoudre automatiquement certains conflits afin de limiter le nombre d'interactions nécessaires avec l'utilisateur. Enfin, le prototype que nous avons développé n'implémentant pas l'ensemble des fonctionnalités de l'architecture, il serait intéressant de le compléter de ce point de vue, notamment pour tester l'efficacité des contrôles d'intégrité de l'*IH* face à des attaques réelles.

Par ailleurs, le système Android est en constante évolution, et notre solution mériterait d'être adaptée pour la nouvelle version, Android Q, qui devrait paraître dans le courant du troisième trimestre 2019. Il est envisagé que cette nouvelle version apporte de nouveaux changements en termes de sécurité et de gestion des permissions, sur la base des pré-versions bêta publiées depuis février 2019 [Jack Price 2019]. Il est notamment question de permettre à l'utilisateur de restreindre l'accès à la géolocalisation aux applications n'étant pas en cours d'utilisation. De plus, cette nouvelle version introduirait l'attribution de rôles aux applications leur conférant un certain nombre de permissions par défaut. En outre, certaines API seraient dépréciées comme celle permettant actuellement de s'afficher en surimpression d'autres applications. Ces changements sont bienvenus et sont complémentaires face à notre politique de sécurité qui devra être légèrement adaptée mais dont l'utilité restera entière. De plus, il est question d'une refonte de l'interface de contrôle des permissions qui serait adjointe à un système d'alerte d'utilisation de certaines ressources et à un écran de surveillance de l'utilisation des permissions pour chaque application [Mishaal Rahman 2019]. Cette nouvelle fonctionnalité serait ainsi dans la même veine que l'application servant à encourager l'utilisateur à paramétrer la politique présentée dans la section 3.4. Enfin, de nouveaux mécanismes renforçant la sécurité du système d'exploitation Android devraient voir le jour, comme l'utilisation de CFI pour vérifier l'intégrité de certaines parties du noyau et du framework. Ces mécanismes supplémentaires pourront ainsi voir leur bonne configuration et activation surveillée par notre hyperviseur dans le cadre de la vérification d'intégrité dont il est le garant.

Pour finir, l'architecture de sécurité présentée dans cette thèse pourrait être utilisée avantageusement pour servir d'autres buts. En effet, l'interception des communications qu'elle fournit est assez générique pour être adoptée dans différents autres cas d'applications. Par exemple, cette interception est susceptible d'être employée pour implémenter d'autres types de règles de contrôle d'accès, en ajoutant

si nécessaire des mécanismes d'introspection supplémentaires. Même en dehors du domaine de la sécurité, l'interception des communications pourrait servir à effectuer des statistiques d'utilisation de l'API Android dans son ensemble. En outre, notre hyperviseur de sécurité bénéficie lui aussi d'une conception assez générique pour être utilisé de manière individuelle ou au sein d'autres projets. En effet, même si les contrôles d'intégrité de notre architecture n'ont pas été implémentés dans leur ensemble, celui-ci peut servir de support à d'autres fonctions critiques requérant un tel niveau de privilège. Sa généricité est aussi un atout lui permettant d'être facilement porté sur d'autres plateformes, telles que les serveurs basse consommation qui utilisent de plus en plus des clusters de processeurs ARM.

Annexe A

Exemples de code

A.1 Modification des droits en écriture d'une page de la MMU pour l'architecture *ARMv8*

```
/**
 * Modification des droits d'accès en écriture (@writable)
 * de la page MMU associée à l'adresse @addr
 */
static int set_mem_write_access(unsigned long addr, int writable)
{
    uint64_t tcr, ttbr, table_entry;
    uint64_t *table;
    int i, j, TnSZ;
    int is_high_addr = (addr >> 63);

    // 1) Vérifier que 25 <= TnSZ <= 33
    tcr = TCR_EL1_read();
    TnSZ = (is_high_addr) ? (tcr >> 16) & 0x3f : tcr & 0x3f;
    if (TnSZ < 25 || TnSZ > 33) {
        error("TnSZ value not supported, aborting\n");
        return 1;
    }

    // 2) Récupérer le bon ttbr et l'adresse de la première table
    ttbr = (is_high_addr) ? TTBR1_EL1_read() : TTBR0_EL1_read();

    // 3) Effectuer un page walk jusqu'à l'entrée de la 3ème table
    table_entry = ttbr & ((1ull << 48) - 1);
    for (i = 0, j = 30; i < 3; i++, j -= 9) {
        table = (uint64_t *) phys_to_virt((table_entry & 0x0000FFFFFFFFF000));
        table_entry = table[(addr >> j) & 0x1ff];

        if ((table_entry & 0x3) != 0x3) {
            error("Page walk : invalid entry or block, aborting\n");
            return 1;
        }
    }

    // 4) Changer les droits en écriture de la page :
    // passer le bit 7 de la table entry à 0 -> rw / 1 -> ro
    if (writable) {
        table_entry &= ~(1ull << 7);
    } else {
        table_entry |= (1ull << 7);
    }
```

```
42      table[(addr >> 12) & 0x1ff] = table_entry;
43
44      // 6) Flush des tlb pour prendre en compte les changements
45      flush_tlb_all();
46      return 0;
47  }
```

A.2 Introspection de la mémoire du driver du binder

```
1   /**
2    * Récupération des infos (pid et global handle) d'un binder_node connu
3    * par un processus (pid @src_pid) sous le handle local @local_handle
4    */
5   static void get_handle_infos(pid_t src_pid, uint32_t local_handle,
6                                struct handle_infos *infos)
7   {
8       struct binder_proc *proc;
9       struct binder_ref *ref;
10      struct rb_node *n;
11      int done = 0;
12      struct mutex *binder_procs_lock =
13          (struct mutex *) kallsyms_lookup_name("binder_procs_lock");
14      struct hlist_head *binder_procs =
15          (struct hlist_head *) kallsyms_lookup_name("binder_procs");
16
17      infos->pid = -1;
18      infos->global_handle = -1;
19
20      if (binder_procs_lock == NULL || binder_procs == NULL) {
21          error("Binder is not loaded or kallsyms_lookup_name failed\n");
22          return;
23      }
24      // Parcours des structures d'infos binder par processus
25      mutex_lock(binder_procs_lock);
26      hlist_for_each_entry(proc, binder_procs, proc_node) {
27          // Exploration des infos pour le PID donné
28          if (proc->pid == src_pid) {
29              debug("proc %d\n", proc->pid);
30              // Parcours en profondeur du rb tree stockant
31              // les références connues (binder_node) du processus
32              spin_lock(&(proc->outer_lock));
33              for (n = rb_first(&proc->refs_by_desc); !done && n != NULL;
34                   n = rb_next(n)) {
35                  ref = rb_entry(n, struct binder_ref, rb_node_desc);
36                  // Recherche d'un binder_node correpondant au handle local
37                  spin_lock(&(ref->node->lock));
38                  if (ref->data.desc == local_handle) {
39                      if (ref->node->proc != NULL) {
40                          // Récupération des infos sur le processus associé
41                          spin_lock(&(ref->node->proc->inner_lock));
42                          debug("\thandle:%d -> node:%d (proc:%d)\n",
43                                ref->data.desc, ref->node->debug_id,
44                                ref->node->proc->pid);
```

```
                    infos->pid = ref->node->proc->pid;
                    spin_unlock(&(ref->node->proc->inner_lock));
                } else {
                    // Cas où le processus est en train d'être tué
                    debug("\thandle:%d -> node:%d (dead_node)\n",
                          ref->data.desc, ref->node->debug_id);
                }
                // Récupération du handle global
                infos->global_handle = ref->node->debug_id;
                done = 1;
            }
            spin_unlock(&(ref->node->lock));
        }
        spin_unlock(&(proc->outer_lock));
        if (done)
            break;
    }
  }
  mutex_unlock(binder_procs_lock);
}

/**
 * Récupération du handle global d'un binder_node chargé en mémoire à
 * l'adresse @ptr, et appartenant au processus identifié par @src_pid
 */
uint32_t binder_get_node_handle(pid_t src_pid, uintptr_t ptr)
{
    uint32_t handle = -1;
    struct binder_proc *proc;
    struct binder_node *node;
    struct rb_node *n;
    struct mutex *binder_procs_lock =
        (struct mutex *) kallsyms_lookup_name("binder_procs_lock");
    struct hlist_head *binder_procs =
        (struct hlist_head *) kallsyms_lookup_name("binder_procs");

    if (binder_procs_lock == NULL || binder_procs == NULL) {
        error("Binder is not loaded or kallsyms_lookup_name failed\n");
        return -1;
    }
    // Parcours des structures d'infos binder par processus
    mutex_lock(binder_procs_lock);
    hlist_for_each_entry(proc, binder_procs, proc_node) {
        // Exploration des infos pour le PID donné
        if (proc->pid == src_pid) {
            debug("proc %d\n", proc->pid);
            // Recherche dans les nodes appartenant au processus
            spin_lock(&(proc->inner_lock));
            n = proc->nodes.rb_node;
            while (handle == -1 && n) {
                node = rb_entry(n, struct binder_node, rb_node);
                // Dans le rb_tree, les nodes sont triés par adresses
                if (node != NULL) {
                    if (ptr < node->ptr)
```

```
99                          n = n->rb_left;
100                     else if (ptr > node->ptr)
101                         n = n->rb_right;
102                     else {
103                         handle = node->debug_id;
104                         debug("ptr:0x%llx -> node:%u\n", node->ptr,
105                               node->debug_id);
106                     }
107                 } else {
108                     error("NULL node ????\n");
109                 }
110             }
111             spin_unlock(&(proc->inner_lock));
112             if (handle != -1)
113                 break;
114         }
115     }
116     mutex_unlock(binder_procs_lock);
117 
118     return handle;
119 }
```

A.3 Makefile de l'*IH*

```
1  sourceDir   := src/hyp_boot
2  buildDir    := build
3  commonDir   := src/common
4  SRC_files   := $(shell find -L $(sourceDir) -name '*.[csS]')
5  
6  prefix      := aarch64-linux-android-
7  cc          := $(prefix)gcc
8  objcopy     := $(prefix)objcopy
9  cflags      := -mcpu=cortex-a53+nofp+nocrc+nosimd+nocrypto -mstrict-align
10 cflags      := $(cflags) -O2 -fmodulo-sched -fmodulo-sched-allow-regmoves
11 cflags      := $(cflags) -ffunction-sections -fdata-sections -fsection-anchors
12 cflags      := $(cflags) -I$(commonDir) -I$(commonDir)/libc -ffreestanding
13 cflags      := $(cflags) -std=gnu99 -pipe -Wall -Wextra -fdiagnostics-color=always
14 cflags      := $(cflags) -flto -ffat-lto-objects -fpic -tno-android-cc
15 ldscript    := $(sourceDir)/ldscript
16 ldflags     := -lgcc -nostdlib -T $(ldscript) -tno-android-ld
17 ldflags     := $(ldflags) -Wl,--sort-section=alignment,--gc-sections
18 
19 # determine objects
20 OBJS        := $(patsubst $(sourceDir)/%, $(buildDir)/%, $(SRC_files))
21 OBJS        := $(basename $(OBJS))
22 OBJS        := $(addsuffix .o, $(OBJS))
23 
24 # default target
25 all: $(buildDir)/image.bin
26 
27 # link
28 $(buildDir)/image.bin: $(buildDir)/image.elf
29     @echo "BIN $@"
```

```
30      @$(objcopy) $^ -O binary $@
31  
32  $(buildDir)/image.elf: $(OBJS)
33      @echo "ELF $@"
34      @$(cc) $(cflags) $^ -o $@ $(ldflags)
35  
36  # include dependency info for existing .o files
37  -include $(OBJS:.o=.o.d)
38  
39  # compile and generate dependency info
40  $(buildDir)/%.o: $(sourceDir)/%.c
41      @echo "CC $@"
42      @mkdir -p $(dir $@)
43      @$(cc) -MMD -MT $@ -MF $@.d $(cflags) -c $< -o $@
44  
45  $(buildDir)/%.o: $(sourceDir)/%.s
46      @echo "AS $@"
47      @mkdir -p $(dir $@)
48      @$(cc) $(cflags) -c $< -o $@
49  
50  # remove compilation products
51  clean:
52      @rm -Rf $(buildDir)/*
```

A.4 Point d'entrée de l'*IH*

```
1   .section .init
2   
3   .global _start
4   _start:
5       b real_start
6   
7   previous_instr: .word 0x0
8   
9   real_start:
10      /* set up stack and save context */
11      adr x29, stack /* x29 = fp */
12      mov sp, x29
13      /* save x0-x7 */
14      stp x0, x1, [sp, #-0x10]!
15      stp x2, x3, [sp, #-0x10]!
16      stp x4, x5, [sp, #-0x10]!
17      stp x6, x7, [sp, #-0x10]!
18  
19      /* correct lr(x30) to point to the previous instruction */
20      sub x30, x30, #4
21      /* restore the instruction present before the uefi binpatch */
22      ldr w0, previous_instr
23      str w0, [x30]
24      /* save lr (x30) */
25      str x30, [sp, #-0x8]!
26  
27      /* call main */
```

```
28     bl notmain
29 
30     /* restore context (x0-x7 + x30) */
31     ldr x30, [sp], #0x8
32     ldp x6, x7, [sp], #0x10
33     ldp x4, x5, [sp], #0x10
34     ldp x2, x3, [sp], #0x10
35     ldp x0, x1, [sp], #0x10
36     /* return to uefi first instruction unpatched */
37     ret
38 
39     /* In case anything bad happens */
40 hang: wfi
41     b hang
42 
43 
44 .section .data
45 
46 stack_up:
47 .space 8*500 /* stack for 500 dword */
48 .space 8*9   /* save x0-x7 + x30 */
49 stack:
50 .word 0xffffffff
```

Bibliographie

[Abdella 2017] J. Abdella, M. Özuysal et E. Tomur. *CA-ARBAC : Privacy Preserving Using Context-Aware Role-Based Access Control on Android Permission System.* Security and Communication Networks, vol. 9, no. 18, pages 5977–5995, janvier 2017. (Cité en pages 38 et 40.)

[Acar 2016] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel et M. Smith. *SoK : Lessons Learned from Android Security Research for Appified Software Platforms.* Dans 2016 IEEE Symposium on Security and Privacy (SP), pages 433–451, mai 2016. (Cité en page 19.)

[Almuhimedi 2015] Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor et Yuvraj Agarwal. *Your Location Has Been Shared 5,398 Times ! : A Field Study on Mobile App Privacy Nudging.* Dans Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI '15, pages 787–796, New York, NY, USA, 2015. ACM. (Cité en page 57.)

[Andriatsimandefitra 2015] Radoniaina Andriatsimandefitra et Valérie Viet Triem Tong. *Detection and Identification of Android Malware Based on Information Flow Monitoring.* Dans The 2nd IEEE International Conference on Cyber Security and Cloud Computing (CSCloud 2015), New York, United States, novembre 2015. (Cité en page 31.)

[Arena 2013] V. Arena, V. Catania, G. L. Torre, S. Monteleone et F. Ricciato. *SecureDroid : An Android Security Framework Extension for Context-Aware Policy Enforcement.* Dans 2013 International Conference on Privacy and Security in Mobile Systems (PRISMS), pages 1–8, juin 2013. (Cité en pages 37 et 40.)

[Averlant 2017a] G. Averlant, B. Morgan, É Alata, V. Nicomette et M. Kaâniche. *An Abstraction Model and a Comparative Analysis of Intel and ARM Hardware Isolation Mechanisms.* Dans 2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC), pages 245–254, janvier 2017. (Cité en pages 6, 27 et 80.)

[Averlant 2017b] Guillaume Averlant. *Multi-Level Isolation for Android Applications.* Dans Software Reliability Engineering Workshops (ISSREW), 2017 IEEE International Symposium On, pages 128–131. IEEE, octobre 2017. (Cité en page 61.)

[Averlant 2018] Guillaume Averlant, Eric Alata, Mohamad Kaaniche, Vincent Nicomette et Yuxiao Mao. *SAAC : Secure Android Application Context a Runtime Based Policy and Its Architecture.* Dans 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), pages 1–5, Cambridge, MA, novembre 2018. IEEE. (Cité en pages 43 et 61.)

[Backes 2014] Michael Backes, Sven Bugiel, Sebastian Gerling et Philipp von Styp-Rekowsky. *Android Security Framework : Extensible Multi-Layered Access Control on Android.* Dans Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC '14, pages 46–55, New York, NY, USA, 2014. ACM. (Cité en page 31.)

[Backes 2015] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz et Philipp von Styp-Rekowsky. *Boxify : Full-Fledged App Sandboxing for Stock Android.* Dans 24th USENIX Security Symposium (USENIX Security 15), pages 691–706, Washington, D.C., 2015. USENIX Association. (Cité en page 34.)

[Backes 2017] M. Backes, S. Bugiel, O. Schranz, P. v Styp-Rekowsky et S. Weisgerber. *ARTist : The Android Runtime Instrumentation and Security Toolkit.* Dans 2017 IEEE European Symposium on Security and Privacy (EuroS P), pages 481–495, avril 2017. (Cité en page 34.)

[Bai 2010] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo et Xiangqun Chen. *Context-Aware Usage Control for Android.* Dans Security and Privacy in Communication Networks, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 326–343. Springer, Berlin, Heidelberg, septembre 2010. (Cité en pages 37 et 40.)

[Bell 1976] D Elliott Bell et Leonard J La Padula. *Secure Computer System : Unified Exposition and Multics Interpretation.* Rapport technique, MITRE CORP BEDFORD MA, 1976. (Cité en page 26.)

[Bianchi 2015] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel et Giovanni Vigna. *NJAS : Sandboxing Unmodified Applications in Non-Rooted Devices Running Stock Android.* Dans Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '15, pages 27–38, New York, NY, USA, 2015. ACM. (Cité en page 35.)

[Biba 1977] Kenneth J Biba. *Integrity Considerations for Secure Computer Systems.* Rapport technique, MITRE CORP BEDFORD MA, 1977. (Cité en page 26.)

[Bonné 2017] Bram Bonné, Sai Teja Peddinti, Igor Bilogrevic et Nina Taft. *Exploring Decision Making with Android's Runtime Permission Dialogs Using in-Context Surveys.* Dans Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017), pages 195–210, Santa Clara, CA, 2017. USENIX Association. (Cité en page 57.)

[Bugiel 2013] Sven Bugiel, Stephen Heuser et Ahmad-Reza Sadeghi. *Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies.* Dans Presented as Part of the 22nd USENIX Security Symposium (USENIX Security 13), pages 131–146, Washington, D.C., 2013. USENIX. (Cité en pages 31, 39 et 40.)

[Canbek 2017] G. Canbek, N. Baykal et S. Sagiroglu. *Clustering and Visualization of Mobile Application Permissions for End Users and Malware Analysts.* Dans 2017 5th International Symposium on Digital Forensic and Security (ISDFS), pages 1–10, avril 2017. (Cité en page 57.)

[Cho 2016] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim et Yunheung Paek. *Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices.* Dans 2016 USENIX Annual Technical Conference (USENIX ATC 16), pages 565–578, Denver, CO, 2016. USENIX Association. (Cité en page 28.)

[Conti 2012] M. Conti, B. Crispo, E. Fernandes et Y. Zhauniarovich. *CRêPE : A System for Enforcing Fine-Grained Context-Related Policies on Android.* IEEE Transactions on Information Forensics and Security, vol. 7, no. 5, pages 1426–1438, octobre 2012. (Cité en pages 37 et 40.)

[Enck 2014] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel et Anmol N. Sheth. *TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones.* ACM Trans. Comput. Syst., vol. 32, no. 2, pages 5 :1–5 :29, juin 2014. (Cité en page 33.)

[European Commission 1992] European Commission, Telecommunications Directorate-General XIII Information Market and Exploitation of Research, Commission Européenne, Direction Générale XIII, European Commission et Telecommunications Directorate-General XIII Information Market and Exploitation of Research. Critères d'évaluation de la sécurité des systèmes informatiques (ITSEC) : critères harmonisés provisoires : juin 1991. Office des Publications Officielles des Communautés Européennes, Luxembourg, 1992. OCLC : 636381964. (Cité en page 23.)

[Fratantonio 2017] Yanick Fratantonio, Chenxiong Qian, Simon Chung et Wenke Lee. *Cloak and Dagger : From Two Permissions to Complete Control of the UI Feedback Loop.* Dans Proceedings of the IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, mai 2017. (Cité en pages 17, 51 et 113.)

[Fu 2017] Jiaojiao Fu, Yangfan Zhou, Huan Liu, Yu Kang et Xin Wang. *Perman : Fine-Grained Permission Management for Android Applications.* Dans Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium On, pages 250–259. IEEE, 2017. (Cité en page 36.)

[Guo 2013] Tao Guo, Puhan Zhang, Hongliang Liang et Shuai Shao. *Enforcing Multiple Security Policies for Android System.* Dans 2nd International Symposium on Computer, Communication, Control and Automation. Atlantis Press, avril 2013. (Cité en page 37.)

[Harrison 1976] Michael A. Harrison, Walter L. Ruzzo et Jeffrey D. Ullman. *Protection in Operating Systems.* Commun. ACM, vol. 19, no. 8, pages 461–471, août 1976. (Cité en page 26.)

[Heuser 2014] Stephan Heuser, Adwait Nadkarni, William Enck et Ahmad-Reza Sadeghi. *ASM : A Programmable Interface for Extending Android Security.* Dans 23rd USENIX Security Symposium (USENIX Security 14), pages 1005–1019, San Diego, CA, 2014. USENIX Association. (Cité en pages 31 et 40.)

[Horsch 2015] J. Horsch et S. Wessel. *Transparent Page-Based Kernel and User Space Execution Tracing from a Custom Minimal ARM Hypervisor.* Dans 2015 IEEE Trustcom/BigDataSE/ISPA, volume 1, pages 408–417, août 2015. (Cité en page 28.)

[Hu 2014] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller et Karen Scarfone. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations.* Rapport technique NIST SP 800-162, National Institute of Standards and Technology, janvier 2014. (Cité en page 25.)

[Hu 2016] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena et Zhenkai Liang. *Data-Oriented Programming : On the Expressiveness of Non-Control Data Attacks.* Dans 2016 IEEE Symposium on Security and Privacy (SP), pages 969–986, San Jose, CA, mai 2016. IEEE. (Cité en page 77.)

[Huber 2016] Manuel Huber, Julian Horsch, Michael Velten, Michael Weiss et Sascha Wessel. *A Secure Architecture for Operating System-Level Virtualization on Mobile Devices.* Dans Dongdai Lin, XiaoFeng Wang et Moti Yung, editeurs, Information Security and Cryptology, Lecture Notes in Computer Science, pages 430–450. Springer International Publishing, 2016. (Cité en page 30.)

[IDC 2018] IDC. *Smartphone OS Market Share.* https ://www.idc.com/promo/smartphone-market-share, 2018. (Cité en page 1.)

[Jack Price 2019] Jack Price. *Android Q : All the New Security and Privacy Features Coming to Android 10.* https ://www.xda-developers.com/android-q-security-privacy-features/, mai 2019. (Cité en page 116.)

[Jeon 2012] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster et Todd Millstein. *Dr. Android and Mr. Hide : Fine-Grained Permissions in Android Applications.* Dans Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '12, pages 3–14, New York, NY, USA, 2012. ACM. (Cité en page 34.)

[Jones 1976] A. K. Jones, R. J. Lipton et L. Snyder. *A Linear Time Algorithm for Deciding Security.* Dans Proceedings of the 17th Annual Symposium on Foundations of Computer Science, pages 33–41. IEEE Computer Society, octobre 1976. (Cité en page 26.)

[Lacombe 2009] Eric Lacombe. *Sécurité des noyaux de systèmes d'exploitation*. PhD thesis, INSA de Toulouse, décembre 2009. (Cité en page 79.)

[Laprie 1996] Jean-Claude Laprie, Jean Arlat, Jean-Paul Blanquart, Alain Costes, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Hubert Guillermain, Mohamed Kaâniche, Karama Kanoun, Corinne Mazet, David Powell, Christophe Rabéjac et Pascale Thévenod. Guide de la sûreté de fonctionnement. Cépaduès, Toulouse (France), 1996. OCLC : 35123685. (Cité en page 20.)

[Latini 2014] Tommaso Latini. *Jarvis : Bridging the Semantic Gap between Android APIs and System Calls*. PhD thesis, Università di Pisa, 2014. (Cité en page 14.)

[Lengyel 2014] T. K. Lengyel, T. Kittel, J. Pfoh et C. Eckert. *Multi-Tiered Security Architecture for ARM via the Virtualization and Security Extensions*. Dans 2014 25th International Workshop on Database and Expert Systems Applications, pages 308–312, septembre 2014. (Cité en pages 28 et 29.)

[Lorch 2003] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura et Sumit Shah. *First Experiences Using XACML for Access Control in Distributed Systems*. Dans Proceedings of the 2003 ACM Workshop on XML Security, XMLSEC '03, pages 25–37, New York, NY, USA, 2003. ACM. (Cité en page 26.)

[Mayrhofer 2019] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker et Nick Kralevich. *The Android Platform Security Model*. arXiv :1904.05572 [cs], avril 2019. (Cité en page 16.)

[Miettinen 2014] Markus Miettinen, Stephan Heuser, Wiebke Kronz, Ahmad-Reza Sadeghi et N. Asokan. *ConXsense : Automated Context Classification for Context-Aware Access Control*. Dans Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, pages 293–304, New York, NY, USA, 2014. ACM. (Cité en pages 39 et 40.)

[Mishaal Rahman 2019] Mishaal Rahman. *How Android Q Improves Privacy and Permission Controls over Android Pie*. https ://www.xda-developers.com/android-q-privacy-permission-controls/, février 2019. (Cité en page 116.)

[Moreira 2017] João Moreira, Sandro Rigo, Michalis Polychronakis et Vasileios P Kemerlis. *DROP THE ROP : Fine-Grained Control-Flow Integrity for the Linux Kernel*. Dans Black Hat ASIA 2017, mars 2017. (Cité en page 78.)

[Morgan 2015a] B. Morgan, É Alata, V. Nicomette, M. Kâaniche et G. Averlant. *Design and Implementation of a Hardware Assisted Security Architecture for Software Integrity Monitoring*. Dans 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC), pages 189–198, 2015. (Cité en page 62.)

[Morgan 2015b] Benoît Morgan, Eric Alata, Vincent Nicomette et Guillaume Averlant. *Abyme : Un Voyage Au Coeur Des Hyperviseurs Récursifs*. Dans

Symposium Sur La Sécurité Des Technologies de l'Information et Des Communications (SSTIC), Actes de La Conférence SSTIC 2015 (Symposium Sur La Sécurité Des Technologies de l'Information et Des Communications, page 29p., Rennes, France, juin 2015. (Non cité.)

[Morgan 2016] Benoît Morgan, Guillaume Averlant, Eric Alata et Vincent Nicomette. *Bypassing DMA Remapping with DMA*. Dans Symposium Sur Le Sécurité Des Technologies de l'Informatino et Des Communications, Actes Du Colloque SSTIC 2016 (Symposium Sur Le Sécurité Des Technologies de l'Informatino et Des Communications), page 9p., Rennes, France, juin 2016. (Non cité.)

[Nadkarni 2016] Adwait Nadkarni, Benjamin Andow, William Enck et Somesh Jha. *Practical DIFC Enforcement on Android*. Dans 25th USENIX Security Symposium (USENIX Security 16), pages 1119–1136, Austin, TX, 2016. USENIX Association. (Cité en page 31.)

[Naveed 2014] Muhammad Naveed, Xiaoyong Zhou, Soteris Demetriou, XiaoFeng Wang et Carl A Gunter. *Inside Job : Understanding and Mitigating the Threat of External Device Mis-Bonding on Android*. Dans Proceedings 2014 Network and Distributed System Security Symposium, San Diego, CA, 2014. Internet Society. (Cité en pages 17, 44 et 113.)

[Nordholz 2015] Jan Nordholz, Julian Vetter, Michael Peter, Matthias Junker-Petschick et Janis Danisevskis. *XNPro : Low-Impact Hypervisor-Based Execution Prevention on ARM*. Dans Proceedings of the 5th International Workshop on Trustworthy Embedded Devices, TrustED '15, pages 55–64, New York, NY, USA, 2015. ACM. (Cité en page 77.)

[Olejnik 2017] Katarzyna Olejnik, Italo Dacosta, Joana Soares Machado, Kévin Huguenin, Mohammad Emtiyaz Khan et Jean-Pierre Hubaux. *SmarPer : Context-Aware and Automatic Runtime-Permissions for Mobile Devices*. Dans 38th IEEE Symposium on Security and Privacy (S&P), pages 1058–1076, San Jose, CA, United States, mai 2017. IEEE. (Cité en pages 39 et 40.)

[PaX Team 2015] PaX Team. *Rap : Rip Rop*. Dans Hackers 2 Hackers Conference (H2HC), octobre 2015. (Cité en page 78.)

[Rahul Mishra 2019] Rahul Mishra et Tom Watkins. *Google Play Protect in 2018 : New Updates to Keep Android Users Secure*. https ://android-developers.googleblog.com/2019/02/google-play-protect-in-2018-new-updates.html, février 2019. (Cité en page 115.)

[Ratazzi 2015] Paul Ratazzi, Ashok Bommisetti, Nian Ji et Wenliang Du. *PINPOINT : Efficient and Effective Resource Isolation for Mobile Security and Privacy*. Dans Proceedings of the SPW Workshop on Mobile Security Technologies (MoST), 2015. (Cité en pages 32 et 66.)

[Rohrer 2013] Felix Rohrer, Yuting Zhang, Lou Chitkushev et Tanya Zlateva. *DR BACA : Dynamic Role Based Access Control for Android*. Dans Proceedings

of the 29th Annual Computer Security Applications Conference, ACSAC '13, pages 299–308, New York, NY, USA, 2013. ACM. (Cité en pages 38 et 40.)

[Seo 2016] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim et Insik Shin. *FLEXDROID : Enforcing In-App Privilege Separation in Android.* Dans Proceedings 2016 Network and Distributed System Security Symposium, San Diego, CA, 2016. Internet Society. (Cité en page 36.)

[Shen 2018] Dong Shen, Zhoujun Li, Xiaojing Su, Jinxin Ma et Robert Deng. *Tiny-Visor : An Extensible Secure Framework on Android Platforms.* Computers & Security, vol. 72, pages 145–162, janvier 2018. (Cité en page 29.)

[Sivakumaran 2018] Pallavi Sivakumaran et Jorge Blasco. *Attacks Against BLE Devices by Co-Located Mobile Applications.* ArXiv e-prints, août 2018. (Cité en pages 17, 44, 59 et 113.)

[Smalley 2013] Stephen Smalley et Robert Craig. *Security Enhanced (SE) Android : Bringing Flexible MAC to Android.* Dans NDSS, volume 310, pages 20–38, 2013. (Cité en pages 30 et 37.)

[Song 2016] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim et Wenke Lee. *Enforcing Kernel Security Invariants with Data Flow Integrity.* Dans Proceedings 2016 Network and Distributed System Security Symposium, San Diego, CA, 2016. Internet Society. (Cité en page 78.)

[Sun 2015] H. Sun, K. Sun, Y. Wang, J. Jing et H. Wang. *TrustICE : Hardware-Assisted Isolated Computing Environments on Mobile Devices.* Dans 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 367–378, juin 2015. (Cité en page 28.)

[Sun 2016] Mingshen Sun, Tao Wei et John C.S. Lui. *TaintART : A Practical Multi-Level Information-Flow Tracking System for Android RunTime.* Dans Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, pages 331–342, New York, NY, USA, 2016. ACM. (Cité en page 34.)

[Wang 2015] Xueqiang Wang, Kun Sun, Yuewu Wang et Jiwu Jing. *DeepDroid : Dynamically Enforcing Enterprise Policy on Android Devices.* Dans Network and Distributed System Security (NDSS) Symposium 2015. Internet Society, 2015. (Cité en page 33.)

[Wang 2018] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao et Guoai Xu. *Beyond Google Play : A Large-Scale Comparative Study of Chinese Android App Markets.* arXiv :1810.07780 [cs], septembre 2018. (Cité en page 115.)

[Wu 2014] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang et Xuxian Jiang. *AirBag : Boosting Smartphone Resistance to Malware Infection.* Dans Proceedings 2014 Network and Distributed System Security Symposium, San Diego, CA, 2014. Internet Society. (Cité en page 30.)

[Xu 2015] L. Xu, G. Li, C. Li, W. Sun, W. Chen et Z. Wang. *Condroid : A Container-Based Virtualization Solution Adapted for Android Devices.* Dans

2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, pages 81–88, mars 2015. (Cité en page 29.)

[Yagemann 2016] Carter Yagemann et Wenliang Du. *Intentio Ex Machina : Android Intent Access Control via an Extensible Application Hook.* Dans Computer Security – ESORICS 2016, Lecture Notes in Computer Science, pages 383–400. Springer, Cham, septembre 2016. (Cité en pages 33 et 92.)

Résumé : Cette dernière décennie a vu croître l'utilisation des smartphones au quotidien. Leur usage est désormais ancré dans les habitudes de consommation du numérique et remplace de plus en plus souvent l'utilisation des ordinateurs classiques. Le smartphone est d'ores et déjà utilisé comme une plateforme privilégiée pour les interactions sociales, mais aussi avec les différents services numériques ayant fleuri depuis l'avènement du « cloud computing ». Ainsi, la prise en charge de multiples activités de la vie de tous les jours leur confère une place prépondérante au quotidien. Cet état de fait est renforcé par la forte capacité de connectivité des smartphones, assurée par de multiples périphériques de communications, qui leur permet d'interagir avec un environnement de plus en plus connecté.

Ainsi, l'ensemble des fonctionnalités offertes par un smartphone oblige celui-ci à devoir gérer une grande quantité de données personnelles ou professionnelles sensibles. On peut considérer comme exemple les données relatives aux applications des réseaux sociaux, aux e-mails, ainsi qu'aux applications bancaires. Par conséquent, un smartphone est une cible très attractive pour de nombreux d'attaquants qui souhaitent mettre en péril la sécurité de cet appareil, des données qu'il héberge, des services numériques qu'il utilise, ou des objets avec lesquels il interagit.

Dans cette thèse, nous nous intéressons spécifiquement à l'environnement Android. Nous avons en effet relevé un manque dans les capacités d'expression du modèle de permission d'Android vis-à-vis d'un certain nombre de menaces émergentes. Pour répondre à ces dernières, nous proposons une politique de sécurité venant compléter le système de permissions actuel d'Android. Celle-ci a pour but de restreindre dynamiquement, i.e. en fonction du contexte courant d'exécution du smartphone, les droits d'exécution et la capacité d'accès aux ressources du smartphone, pour chaque application installée. Les capacités d'expression de cette politique permettent, par exemple, à un utilisateur de restreindre la liste des applications pouvant s'exécuter de manière concurrente à une application critique, ou bien d'accorder à une application un accès exclusif à un périphérique. Outre les bénéfices en termes de sécurité, l'utilisation d'une telle politique permet à un utilisateur d'avoir un contrôle plus fin sur l'accès des applications aux données qui relèvent de la vie privée.

Cette politique de sécurité constitue la première contribution de cette thèse. Elle est adjointe à la conception d'une architecture de sécurité implémentant les mécanismes de contrôle d'accès nécessaires à sa mise en œuvre. Cette deuxième contribution se repose sur une architecture multi-niveau, i.e. comprenant plusieurs composants de différents niveaux de privilèges. Plus précisément, ceux-ci ont été implantés à la fois dans le framework Android, dans le noyau Linux, et dans un hyperviseur. La dernière contribution de cette thèse consiste en la réalisation d'un prototype de cette architecture sur une carte de développement, associée à la présentation de tests permettant de montrer l'efficacité et la pertinence de l'approche.

Mots clés : Smartphones, Android, Politique d'autorisation, Architecture de sécurité multi-niveau, Hyperviseur, Sécurité