TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ACC	Accuracy
ACL	Access Control List
ADB	Android Debug Bridge
ADS	Anomaly Detection System
AHP	Analyse Hiérarchique des Procédés
AOSP	Open Source Project
API	Application Programming Interface
AUC	Area Under Curve
CART	Classification And Regression Tree
CPU	Central Processing Unit (processeur)
CSER	Consortium for Software Engineering Research
CS	Chi-square
DAC	Discretionary Access Control
DoS	Deny of Service
FN	False Negative
FP	False Positive
FSM	Finite-state machine
GID	Group Identifier
GUI	Graphic User Interface
ICC	Inter-Component Communication
ID	Identifiant
IDS	Intrusion Detection Systems
IG	Information Gain
iOS	iPhone Operating System
IPC	Inter-Process Communication
LTTng	Linux Trace Toolkit Next Generation

MAC	Mandatory Access Control
MMU	Memory Management Unit
OS	Operating System (système d'exploitation)
RAM	Random Access Memory
RF	Random Forest
ROC	Receiver Operating Characteristic
RDDR	Recherche et Développement pour la Défense Canada
RPC	Remote Procedure Call
SD	Secure Digital (carte mémoire amovible de stockage)
SMS	Short Message Service
SVM	Support Vector Machines
TP	True Positif
TN	True Negative
UID	User Identifier
VLN	Varied Length N-grams

LISTE DES SYMBOLES ET UNITÉS DE MESURE

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

INTRODUCTION

Mise en contexte

Les systèmes embarqués à petite échelle sont en rapide évolution, résultant dans plusieurs familles de microprocesseurs qui répondent aux besoins en performance, puissance et coûts pour presque tous les marchés de l'application. Par exemple, la carte embarquée « Beagleboard », sortie en 2008, dispose d'un processeur simple cœur cadencé à 600 Mégahertz (MHz) et 128 de mémoire vive RAM (*Random Access Memory*) (Frazier, 2008), tandis que la carte MatrixTBS2910, le puissant système sur puce publié en 2014, dispose d'un processeur quadricœur cadencé à 1 GHz par cœur, de 2 Go de RAM et de16 gigaoctet (Go) de stockage embarqué (Joly, 2014).

Les téléphones mobiles sont l'un des systèmes embarqués à petite échelle, les plus populaires d'aujourd'hui. Cet univers est en plein bouillonnement, les industriels s'efforcent d'innover pour prendre position sur le marché. Android, l'OS (*Operating System*) de Google, est l'une des principales plateformes adoptées par l'industrie des téléphones intelligents. Il présente une part de marché mondial de l'ordre de 81.5 % (FRAMINGHAM, 2015) en 2014, par rapport à l'OS de son concurrent Apple, l'iOS (*iPhone Operating System*), qui continue progressivement de chuter, avec une part de marché de 14,8 %, contre 18.7 % en 2012.

Android a été initialement introduit pour les téléphones intelligents et les tablettes seulement. En 2014, plus d'un milliard d'appareils Android ont été livrés dans le monde (Richter, 2015). L'ensemble de la pile logicielle et les outils de développement ont été conçus et mis au point pour ces dispositifs. Les fonctions sophistiquées d'Android (par exemple, l'interface utilisateur, la communication sécurisée, la pile réseau...) l'ont rendu lucratif pour des appareils autres que ceux mobiles, comme les dispositifs d'application de télécommunication, médicale, automobile et d'autres dispositifs embarqués, ce qui encourage les organisations à l'adopter dans leurs solutions d'entreprise.

En effet, le développement Android pour les systèmes embarqués, offre de nombreux avantages qui peuvent aider à améliorer les solutions d'entreprise: permet de créer des interfaces utilisateurs personnalisées hautement différenciées et adéquates au type d'appareil spécifique, personnaliser et porter des composants clés du cadre Android comme *WebKit* et *OpenGL*, apporter la vidéoconférence Skype sur l'appareil Android et accélérer le délai de mise sur le marché.

Malheureusement, la popularité de ces dispositifs fait d'eux des cibles attrayantes pour les logiciels malveillants comme « Cabir », qui a infecté les téléphones Symbian en 2004 (Millard, 2004) et le malware de SMS qui a infecté environ 1,2 million des dispositifs Android en 2014 (ABHISHEK, 2014). L'ouverture (Husted, Saïdi et Gehani, 2011) et la programmabilité de la plateforme Android, la rend la cible la plus populaire et la plus vulnérable à des diverses attaques malveillantes avec près de 2000 échantillons de logiciels malveillants Android découverts chaque jour (Sophos, 2014). Ces menaces deviennent de plus en plus dangereuses, car il y a encore un manque d'outils de sécurité, appropriés, pour protéger efficacement la plateforme Android.

Toutefois, lorsque les organisations commencent à intégrer ces systèmes à leurs solutions d'entreprise, ils font face aux nombreux problèmes de configuration et de sécurité. C'est ainsi que la sécurité logicielle des systèmes embarqués, basés en particulier sur la plateforme Android, est devenue une préoccupation de plus en plus importante pour les entreprises.

Nombreuses sont les contre-mesures et les approches adoptées pour assurer la sécurité de ces dispositifs. Cependant, on est toujours contraints aux performances limitées de ces appareils à savoir la mémoire, le processeur, la batterie, etc. Donc, la nécessité de mettre en place une solution de sécurité optimisée en termes de mémoire et de temps d'exécution devient de nos jours un enjeu important.

Problématique

Depuis son lancement, Android a capté l'intérêt des entreprises, des développeurs et du public en général. Il a été constamment amélioré soit en termes de fonctionnalités ou de matériels supportés, en plus de l'extension de sa portabilité aux nouveaux types de dispositifs, différents des périphériques mobiles. Cependant, le système Android, reste l'OS le plus insécurisé, avec environs de 95 % des attaques qui ciblent cette plateforme (Manekari, 2013). Pour cela, il est important pour qu'Android soit adopté, une stratégie de sécurité claire et efficace doit être mise en place.

Une large gamme de produits de sécurité est apparue sur le marché en réponse à la question de la sécurité des systèmes embarqués à petite échelle tels que les téléphones intelligents. Ils ont mis en œuvre des solutions anti-malware embarquées, dont la majorité sont des adaptations de produits de sécurité traditionnels, qui sont déployés sur les ordinateurs de bureaux et qui n'utilisent pas les ressources limitées économiquement. Par exemple, les sociétés de sécurité « McAfee », « Symantec » et « Kaspersky », fournissent des versions mobiles de leur logiciel antivirus commercial. Ces solutions sont généralement similaires à leurs homologues de bureau et entraînent un coût important lorsqu'elles sont exécutées sur un dispositif à ressources limitées comme un téléphone intelligent. En fait, la puissance de calcul, la capacité de mémoire et la puissance d'énergie des téléphones intelligents ne font pas le poids si on les compare aux ordinateurs de bureau (Perrucci, Fitzek et Widmer, 2011). Cependant, les solutions de sécurité pour PC ne seront pas applicables sur tels systèmes embarqués à ressources limitées.

La plupart de ces solutions sont basées sur la technique de détection statique, basée sur les signatures. Cette technique cherche des modèles prédéfinis ou signatures correspondant aux attaques connues répertoriées dans une base de données sur un serveur distant, par conséquent elle ne peut pas détecter les nouvelles attaques. De plus, la localisation de la base des signatures en dehors de l'appareil, exige une connexion réseau rapide, continue et

sécurisée. Autrement, l'augmentation du délai de transfert de données va augmenter la latence, ce qui affecte le temps de détection. Aussi, sous un réseau non fiable, les données peuvent être volées ou perdues et le dispositif peut même être contrôlé par un tiers. Une autre limitation de cette technique est le fait que l'outil de sécurité dépend toujours de la base de données distante et ne peut pas donc tourner efficacement sur l'appareil sans connexion internet. En outre, l'évolution rapide des malwares génère une base de données des signatures énorme, qui dépasse la mémoire des systèmes à petite échelle.

Des recherches récentes ont proposé, également, des mécanismes de détection pour se défendre contre la menace croissante des logiciels malveillants mobiles (Arp et al., 2014; Enck et al., 2014; Min, Mingshen et Lui, 2014), bien que certaines ont entraîné une surcharge importante des ressources. Par exemple, AntiMalDroid (Zhao et al., 2011) et DroidAPIMiner (Aafer, Du et Yin, 2013) sont basés sur une autre technique de détection, dite détection d'anomalies. Cette technique, crée des profils de comportements normaux en formant un modèle normal sur des ensembles de données bénignes profilées sur une activité normale du système. Cette approche permet de détecter les logiciels malveillants connus et inconnus et elle ne dépend pas d'une base de données de signatures, ce qui permet son déploiement sur l'appareil. Elle permet aussi de détecter un type d'attaque important ciblant les Smartphone Android et qui n'est pas détectable par l'approche basée sur les signatures. Cette attaque consiste à reconditionner des applications légitimes pour injecter des activités malveillantes. Cependant la détection d'anomalies utilise, généralement, des algorithmes de détection complexes conçus pour augmenter le taux de détection et de précision. Les anciennes approches de détections de malwares utilisent l'apprentissage machine, qui nécessite une grande puissance de calcul (Zhao et al., 2011), tandis que les dernières approches utilisent le « Data-Mining, », qui implique une gestion extensive de données (Aafer, Du et Yin, 2013).

En outre, une tendance récente d'effectuer des contrôles de sécurité sur des serveurs distants (Amos, Turner et White, 2013; Burguera, Zurutuza et Nadjm-Tehrani, 2011), introduit également des coûts importants d'énergie en raison de téléchargement de données.

Ces approches améliorent la précision de détection, cependant, elles ne tiennent pas en compte les ressources limitées des appareils embarqués à petite échelle, ce qui rend difficile leurs déploiements sur les dispositifs de petite taille. Par exemple, Overo FE COM (Gumstix.com) et APC8750 (apc.io, 2014) ont des ressources limitées, avec un maximum de 512 Mo de RAM et 800 MHz de CPU (*Central Processing Unit*). Même le nouveau Samsung Galaxy S5, qui vient avec 16 Go de stockage à bord , possède moins de 8 Go de stockage utilisable (Ng, 2014).

Ainsi, les défis que nous rencontrons consistent à trouver le meilleur compromis entre la sécurité et la facilité d'utilisation et de pouvoir gérer efficacement le processus de détection d'anomalies sur les appareils mobiles en tenant compte de leurs ressources limitées.

Donc, dans ce présent travail, nous abordons la question de la recherche qui consiste à : comment concevoir et développer un cadre de détection et de profilage global, pour les appareils mobiles, qui assure une détection d'anomalie pertinente en utilisant efficacement les ressources limitées?

Objectifs et réalisations

Notre objectif principal est de trouver le meilleur compromis entre la sécurité et la convivialité dans les systèmes embarqués à ressources limitées sous la plateforme Android. Cet objectif peut être divisé en plusieurs sous objectifs comme suit :

- Étudier et implémenter les différentes solutions possibles qui nous permettent d'adapter les algorithmes de détection d'anomalies à la disponibilité des ressources du système.
- Mettre en place un cadre de détection sur l'appareil, qui ne dépend pas d'un serveur externe.

- Concevoir un cadre dynamique, permettant d'appliquer plus d'un algorithme de détection lorsque nous avons suffisamment de ressources.

Pour atteindre ces objectifs, nous utilisons l'approche comportementale de détection d'anomalies. Dans une phase initiale d'apprentissage, nous établissons un modèle pour un comportement normal. Ensuite, dans la phase de détection, si le comportement est trop différent du comportement normal, alors il sera considéré comme anormal. Cette technique nous permet de détecter les malwares connus et inconnus sans avoir besoin d'une base de données de signature énorme ou de la disponibilité continue du réseau. Ainsi, nous adoptons cette technique pour le système de détection d'intrusion basée sur l'hôte à ressources limité.

Nous construisons un modèle de comportement normal en utilisant des traces d'exécutions de l'application normale. Pendant le fonctionnement, le système de détection d'anomalie tente de détecter les événements qui s'écartent sensiblement du profil normal établi. Bien que ces écarts soient considérés comme des événements anormaux, ils peuvent générer un grand nombre de fausses alarmes si l'algorithme ne possède pas des seuils appropriés. Ce problème se produit lorsque le modèle normal n'est pas représentatif du comportement du système de logiciel normal.

Après avoir établi un algorithme de détection approprié, nous tentons de répondre aux questions soulevées à garantir à la fois la sécurité et la facilité d'utilisation pour les dispositifs à ressources limitées. Nous proposons un système de détection de malware efficace qui gère ce compromis en tenant compte de plusieurs types de ressources.

L'objectif est d'assurer la protection d'un dispositif même avec des ressources limitées, ce qui signifie établir une solution de détection adaptative basée sur des résultats du profilage. Nous avons expérimenté quatre algorithmes de détection d'anomalies. Pour chaque algorithme, nous avons créé un modèle de comportement normal en formant sur certaines traces normales. Ensuite, nous avons analysé l'effet de plusieurs facteurs, dont le seuil de

détection, la taille et la longueur des séquences de motifs, sur la performance de chaque algorithme et sur les ressources.

Les principales contributions de ce présent travail sont :

- Une approche de détection d'anomalie générique pour surveiller les mises à jour malicieuses de la totalité des applications Android installées;
- L'expérimentation de différents algorithmes à base de n-grams (courtes séquences d'appels système) et l'étude de leur précision de détection.
- La caractérisation d'un profil de consommation de ressources des algorithmes expérimentés.

Ces contributions ouvrent la porte à une détection d'anomalie adaptative pour les systèmes à petite échelle où les algorithmes de détection peuvent être choisis en temps réel en se basant sur un compromis entre la précision et des frais généraux des ressources.

Méthodologie suivie

La Figure 0.1 illustre une représentation graphique de notre méthodologie de recherche.

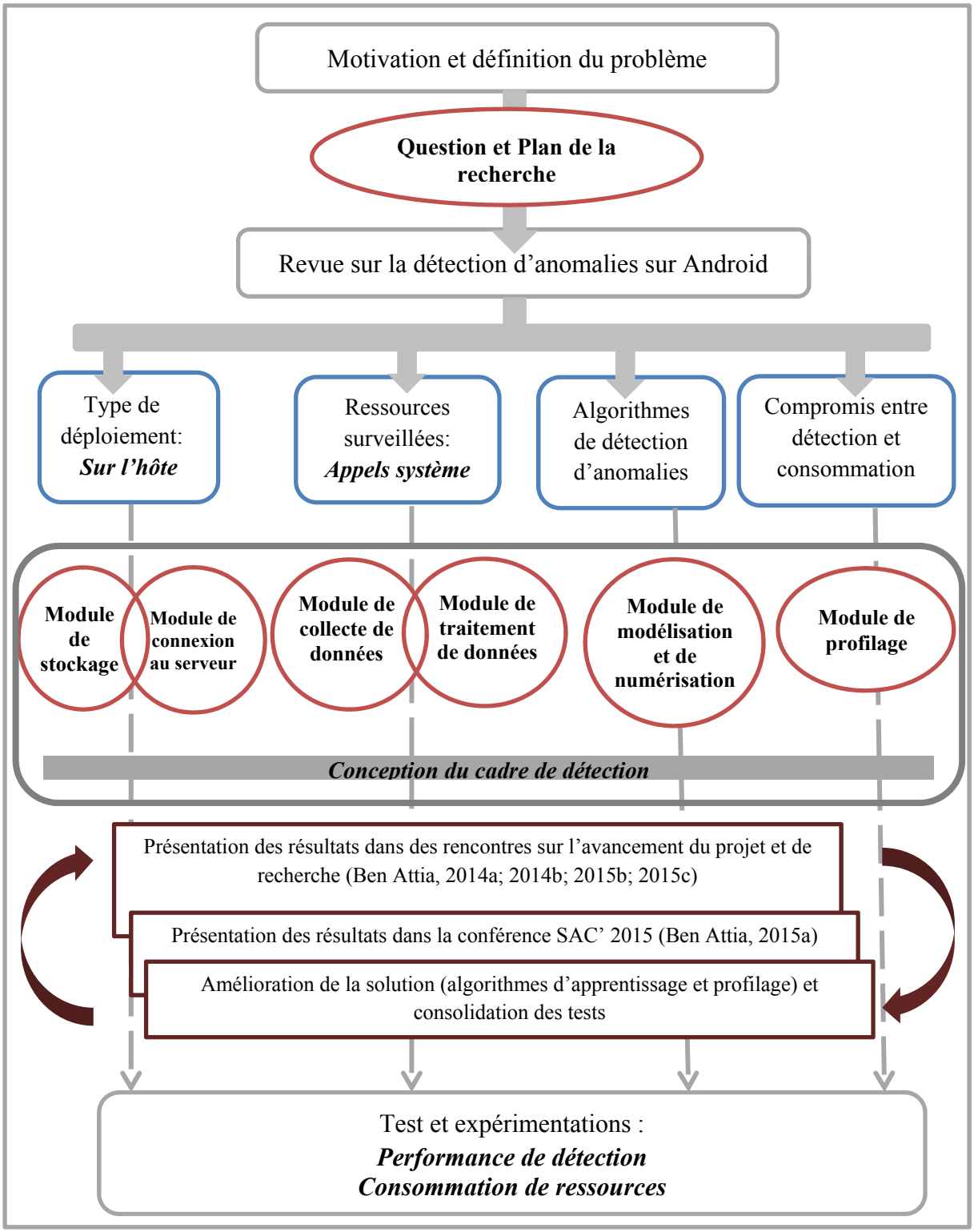

Motivation et définition du problème
Question et Plan de la recherche
Revue sur la détection d'anomalies sur Android
Type de déploiement: Sur l'hôte
Ressources surveillées: Appels système
Algorithmes de détection d'anomalies
Compromis entre détection et consommation
Module de stockage
Module de connexion au serveur
Module de collecte de données
Module de traitement de données
Module de modélisation et de numérisation
Module de profilage
Conception du cadre de détection
Présentation des résultats dans des rencontres sur l'avancement du projet et de recherche (Ben Attia, 2014a; 2014b; 2015b; 2015c)
Présentation des résultats dans la conférence SAC' 2015 (Ben Attia, 2015a)
Amélioration de la solution (algorithmes d'apprentissage et profilage) et consolidation des tests
Test et expérimentations :
Performance de détection
Consommation de ressources

Figure 0.1 Représentation graphique de la méthodologie de la recherche

Dans une première phase, nous avons commencé par une étude sur les mécanismes et les vulnérabilités de sécurités sur Android, cette étude nous a permis de bien définir notre problématique et plan de recherche.

Puis en nous basant sur une revue de littérature des techniques de détection d'anomalies sur la plateforme Android, nous avons pu identifier les différentes caractéristiques de notre solution (présentées par des rectangles bleus dans figure 0.1). Ces caractéristiques constituent le fondement de notre cadre de détection qui comprend plusieurs modules permettant de répondre aux questions de la recherche.

Ensuite, nous avons expérimenté notre solution avec plusieurs tests de détection et de consommations. Une première analyse des résultats a été le sujet d'une communication (Ben Attia, 2015a) à la conférence « *30th Annual ACM/SIGAPP Symposium on Applied Computing (SAC'2015)* » en 2015.

La dernière étape était l'amélioration de la solution avec des expérimentations plus robustes, une base de données de test plus élargie et la conception d'autres algorithmes de détection d'anomalies.

Chaque réalisation (présenté par des cercles rouges dans figure 0.1) dans notre recherche a était présentée et validée à travers des livrables et des rencontres sur l'avancement du projet qui est en collaboration avec le centre de recherche et développement pour la défense Canada (RDDR) (Ben Attia, 2014a; 2014b; 2015b) et des rencontres dans le cadre du programme de consortium de recherche en génie logiciel (CSER : *Consortium for Software Engineering Research*) (Ben Attia, 2015c), et ceci durant toute la durée de la recherche.

L'organisation du rapport

L'organisation de ce rapport sera comme suit : nous allons tout d'abord présenter les concepts de base de la plateforme Android. Ce chapitre présente un survol sur l'architecture, les mécanismes de base et les failles de sécurité de cette plateforme.

Puis, dans le chapitre 2, nous allons examiner les systèmes de détection d'anomalies sur la plateforme Android, en étudiant et analysant les travaux connexes.

Ensuite, dans le troisième chapitre, nous allons présenter le cadre de sécurité proposé. Ce chapitre décrit les différents modules mises en place et détaille les différents algorithmes déployés.

Le chapitre 4 contient les ensembles de données, les protocoles expérimentaux et les paramètres d'évaluation utilisés dans nos expériences, ce chapitre comporte les différents tests de détection et de surconsommations, suivies d'une discussion des résultats.

Enfin, le dernier chapitre sera consacré à la description de la réalisation de ce travail. Ce chapitre énumère les travaux futurs et conclut le rapport.

Diffusion des résultats

Les résultats de ce travail ont été publiés dans un article de conférence (Ben Attia, 2015a) présenté en mai 2015 à la 30 th Annual ACM/SIGAPP Symposium on Applied Computing (SAC' 2015) (*Voir* ANNEXE III) et en mars 2015 dans le cadre du programme CSER (Ben Attia, 2015c). De plus, nous prévoyons de soumettre un article journal sur la totalité du travail.

CHAPITRE 1

SÉCURITÉ SOUS ANDROID

1.1 Introduction

Ce chapitre présente une introduction à la sécurité sous la plateforme Android. Il commence par une description générale du système Android et de ses composants, suivis d'un aperçu de ses mécanismes de sécurités. Il se concentre ensuite sur ses principales vulnérabilités et ses failles de sécurité. Enfin, ce chapitre se termine par une motivation des attaques contre Android.

1.2 Le système Android

Android est un système d'exploitation mobile développé par Google, il est basé sur une version modifiée du noyau Linux. Il a été publié en tant qu'Android AOSP (*Open Source Project)* en 2007. Ce système supporte la plupart des plateformes embarquées telles que ARM, MIPS, Power, x86 et est connue comme « la première plateforme mobile complète, ouverte et libre » (webappers, 2008).

Android a été conçu pour intégrer au mieux des applications existantes de Google (Gmail, cartographie, google Maps, Google Agenda, Google Talk, YouTube) et il bénéficie d'une architecture en couche complète faisant de lui une plateforme riche. Il peut donc se subdiviser en quatre couches principales selon la figure 1.1.

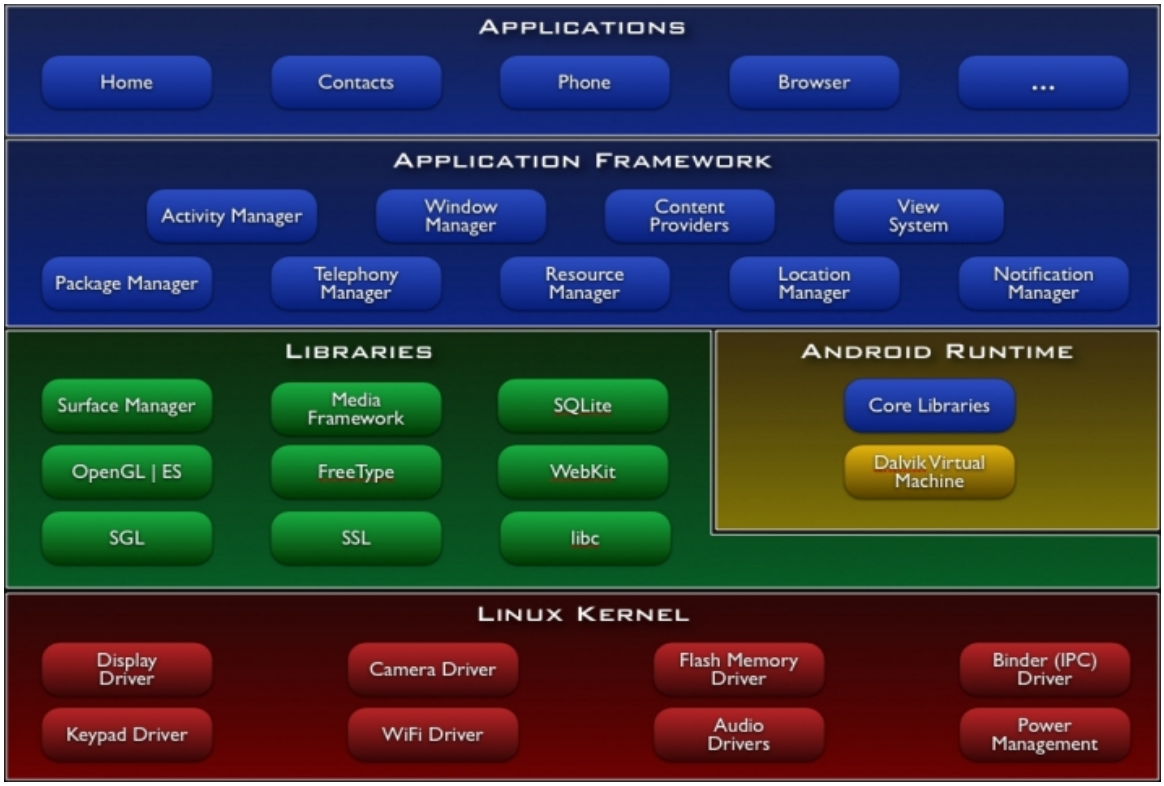


Figure 1.1 Architecture du système Android.
Tirée de nikhilsalunkedev[1](2014)

Le noyau Linux offre des services système de base tels que la sécurité, la gestion d'énergie, la gestion de mémoire, gestion de processus, etc.

La couche des bibliothèques natives est un ensemble de librairies, code ouvert, écrit en C/C++ jouant le rôle d'un logiciel médiateur. C'est sur cette couche que se greffe « *Android Runtime* », un moteur d'exécution et d'hébergement des applications Android, comprenant la machine virtuelle « *Dalvik* » et ses bibliothèques de base fournissant les fonctionnalités spécifiques à Android. Ce moteur est conçu avec un souci de compacité et d'efficacité pour un usage sur des appareils mobiles.

---

[1] nikhilsalunkedev. 2014." What is Android?". En ligne. <https://nikhilsalunkedev.wordpress.com/2014/03/21/what-is-android/>. Consulté le 12 mai 2014.

La couche de plateforme logicielle, nommée aussi cadre de développement. Cette couche, comme celle juste au-dessus, est écrite entièrement en Java. Elle permet de mutualiser les ressources entre les applications Java et offre aux développeurs la possibilité de produire des applications diverses et innovantes à travers un ensemble d'API (*Application Programming Interface*). Toutes les applications utilisent le même cadre de développement, peu importe la nature de l'application.

La couche Applications contient les logiciels installés sur l'appareil sous forme de paquets .apk, écrits par l'équipe Android ainsi que tout logiciel tiers de provenances variées : *Android-Market*, site Internet et, bien sûr, depuis votre ordinateur.

1.3 Modèle de sécurité d'Android

Le modèle de sécurité d'Android est mis en œuvre sur deux couches ; au niveau applicatif et au niveau du noyau.

1.3.1 Au niveau du noyau Linux

Le modèle de permissions, appliqué par l'intergiciel Android, contrôle l'accès aux composants de l'application (telle que la possibilité d'invoquer un service fourni par une autre application) et l'accès aux ressources du système (telle que la possibilité d'accéder à la caméra ou le réseau).

Ce modèle est directement exposé aux développeurs d'applications Android, qui doivent préciser l'ensemble des autorisations requises à travers le fichier manifeste de leur application et aux utilisateurs finaux, qui doivent approuver l'ensemble des autorisations jugées dangereuses avant d'installer une application.

1.3.2 Au niveau des applications

Au niveau du noyau, le mécanisme de « *sandboxing* » et d'isolation des applications est nécessairement appliqué par le noyau Linux depuis la machine virtuelle *Dalvik*.

1.4 Mécanismes de sécurités

Android implémente nativement quelques mécanismes qui offrent un certain niveau de sécurité.

1.4.1 Le contrôle d'accès discrétionnaire et le sandboxing

Le mécanisme DAC (*Discretionary Access Control*) permet le contrôle d'accès des utilisateurs aux fichiers et aux répertoires. Il fonctionne d'une manière invisible pour les développeurs d'applications et les utilisateurs.

DAC permet d'isoler les applications des ressources systèmes. En effet, il est utilisé pour autoriser ou non les applications à accéder aux ressources système, par exemple la possibilité d'activer le Bluetooth ou des sockets réseau et la capacité d'accéder au système de fichiers sur la carte mémoire SD (*Secure Digital*).

Le DAC prend donc la forme de liste de contrôle d'accès ACL (*Access Control List*), qui permettent aux propriétaires de spécifier des permissions distinctes pour des utilisateurs et des groupes spécifiques.

DAC permet également d'isoler les applications les unes des autres. À l'installation, chaque application est attribuée à un utilisateur et un groupe unique (UID et GID, respectivement) et cette paire (UID, GID) est attribuée à des processus et des fichiers de données privées associées à l'application.

Cette approche est conçue pour empêcher une application d'accéder directement à l'état du processus ou des fichiers d'une autre application via les interfaces du noyau (les applications signées par la même clé peuvent éventuellement être exécutées dans la même paire [UID, GID] s'ils souhaitent partager pleinement les ressources sans restriction).

Le DAC prend donc la forme des bits d'autorisation UNIX, qui définissent la protection en lecture, écriture et exécution en fonction du statut de l'utilisateur (propriétaire, groupe et autres utilisateurs). Par la suite, les fichiers systèmes appartiennent soit au « système » soit au « root », tandis que les autres applications ont leurs propres identificateurs uniques.

1.4.2 Mécanisme de permission (fichier Manifest)

Ce mécanisme est fourni par la couche logicielle intermédiaire d'Android. Un moniteur de référence applique le contrôle d'accès obligatoire MAC (*Mandatory Access Control*) sur les appels ICC (*Inter-Component Communication*).

Les interfaces de sécurité sensibles sont protégées par des autorisations Android standard, telles que les appels téléphoniques, Internet, envoyer des SMS...

En outre, les applications peuvent déclarer des autorisations personnalisées afin d'empêcher l'accès aux interfaces propres. Ces autorisations sont déclarées dans un fichier *Manifest* et sont approuvées lors de l'installation par un contrôle contre les signatures des applications, une déclaration de ces autorisations et une confirmation de l'utilisateur.

1.4.3 L'encapsulation de composants

Les composants privés sont accessibles uniquement par les composants au sein de la même application. Les composants publics sont accessibles par d'autres applications, cependant,

l'accès complet peut être limité en exigeant des applications appelantes d'avoir les autorisations spécifiques.

1.4.4 Unité de gestion mémoire

Linux utilise une unité de gestion d'accès mémoire, MMU (*Memory Management Unit),* permettant la séparation des processus dans des espaces mémoire différents, ce qui empêche une application d'accéder à l'espace mémoire d'une autre application.

1.4.5 Signature des applications

Android utilise des signatures cryptographiques pour vérifier l'origine des applications et pour établir des relations de confiance entre elles (un code modifié invalide automatiquement la signature). Par conséquent, les développeurs doivent signer le code de l'application. Cela permet d'activer les autorisations en se basant sur la signature et permet aussi aux applications, signées par le même développeur, de partager le même ID (identifiant) utilisateur.

1.4.6 Langage Java

Le développement d'applications Android est principalement réalisé en Java, langage fortement typé. Ceci nous permet de nous protéger automatiquement de certaines erreurs de programmation pouvant entraîner, entre autres, des débordements de tableaux « *buffers overflows* » et donc de l'exécution de code arbitraire.

1.4.7 Services et Internet

Pour pirater un système connecté à Internet, on se base principalement sur les vulnérabilités de services qui sont en écoute sur le réseau. Par défaut, les services qui fonctionnent sur le système Android ne sont pas à l'écoute de connexions entrantes sur le téléphone. Par la suite,

la probabilité d'une attaque en provenance du système lui-même est donc beaucoup plus importante.

1.4.8 Suppression et mise à jour d'applications à distance

Google a évidemment la possibilité de retirer des applications du marché Android, mais peut également supprimer des applications directement installées sur les téléphones sans confirmation de la part de l'utilisateur.

En effet, si un logiciel de connexion présente des risques, il serait détecté par Google, en plus ça va empêcher les nouveaux utilisateurs de l'installer, il est possible en peu de temps de supprimer l'application de tous les appareils. Le système Android peut également recevoir des mises à jour sans que l'utilisateur s'en aperçoive.

1.4.9 Bluetooth

La connexion, le transfert du fichier et le pairage ne peuvent être établis que par l'autorisation de l'utilisateur et l'installation de l'application se fait manuellement grâce à un explorateur de fichiers. De plus l'appareil ne reste détectable que durant deux minutes. Donc, la probabilité d'installer une application malveillante via Bluetooth est faible.

1.5 Les vulnérabilités

Dans cette section nous allons citer les principales vulnérabilités du système Android.

1.5.1 Utilisateurs inexpérimentés

Le manque de formation de l'utilisateur est une vulnérabilité importante. L'utilisateur a un faux sentiment de sécurité sur son téléphone mobile. Alors qu'il a probablement été sensibilisé aux problèmes les plus importants pouvant survenir sur un ordinateur, il

n'applique pas le même modèle aux téléphones et n'aura donc pas la méfiance nécessaire pour éviter certaines menaces.

1.5.2 Marché d'applications non fiable

Les marchés d'applications ne sont pas forcément fiables dans la mesure où ni les applications ni l'identité du développeur ne sont vérifiées.

Un développeur peut donc se faire passer pour une banque et publier une fausse application de « *e-banking* ». De plus, le développeur est libre de donner deux noms différents à la même application, un nom pour l'application sur « *Android Market* », l'autre pour l'application sur le téléphone. Ceci permet de faire passer une application pour une autre auprès d'un utilisateur non attentif.

1.5.3 Accès temporaire au terminal

Si le téléphone est perdu ou oublié, l'absence de verrouillage ou un verrouillage trop lent peut laisser le temps à une personne tierce d'accéder à des informations sensibles ou d'effectuer des actions critiques.

1.5.4 Environnement domestique

Le téléphone peut être contrôlé lorsqu'il est connecté à l'ordinateur (à condition d'avoir activé le mode débogage). En encourageant la victime à activer ce mode et à connecter son téléphone à l'ordinateur, on peut installer un malware sur le téléphone sans affichage des permissions (grâce à ADB : *Android Debug Bridge*).

1.5.5 Environnement ouvert

Certaines applications peuvent abuser de leurs permissions. En effet, une application alternative d'envoi de SMS a besoin d'une permission pour faire son travail, mais elle peut aussi, sans l'autorisation de l'utilisateur, envoyer des SMS surtaxés par exemple.

1.6 Les attaques contre Android

1.6.1 Drive-by download

C'est l'infection d'un appareil par un logiciel malveillant lors d'une simple visite d'un site Web. Cette attaque exploite les vulnérabilités des navigateurs web et du manque d'attention de l'internaute. Parmi les scénarios d'une attaque *drive-by-download* sur les appareils mobiles, on trouve l'envoi à la victime des messages et des courriels qui comportent des liens vers des sites web malicieux. Il y a aussi le scénario où la victime visite un site web de l'attaquant qui comporte du code malveillant.

1.6.2 Les botnets

C'est l'infection par un réseau de « machines *zombies* ». Il suffit d'une simple consultation sur un site web comportant du code malveillant pour être infecté. Les « *botnets* » malveillants servent en général à réaliser des opérations de « *phishing* », identifier et infecter d'autres machines par diffusion de virus et de programmes malveillants (malwares).

1.6.3 Hameçonnage en modifiant la configuration du serveur mandataire

Cette faille a été découverte en 2013, c'est une application qui permet aux applications bénéficiant d'un accès complet à internet de reconfigurer le mandataire d'un terminal pour conduire à des abus de « *phishing* », ce qui provoque ainsi, l'atteinte aux données personnelles. Pour que cette faille ne soit pas exploitée dans le futur, l'équipe de Google a

pris ses dispositions en bloquant l'application « *AdBlock Plus* », qui modifie automatiquement les paramètres de proxy d'un périphérique lorsqu'il est activé, sous Android 4.2.2.

1.6.4 La faille Exynos

Cette faille touche les *Samsung Galaxy* équipés d'un processeur *Exynos*. Elle permet facilement de rooter un appareil et d'accéder donc aux données personnelles. Cette faille a été corrigée avec une mise à jour en ligne mise en place par Samsung en janvier 2013. Mais cette solution reste limitée puisqu'elle provoque une baisse significative de l'autonomie.

1.6.5 L'injection du code

Chaque jour, un grand nombre d'applications est ajouté dans le marché Android. Ceci limite la popularité des malwares. Pour cela, les auteurs de logiciels malveillants téléchargent des applications légitimes populaires, tel que Skype et Angry Birds(Avenger, 2013; Danchev, 2013; sectechno, 2014), les décortiquent (en utilisant, l'ingénierie inverse) pour obtenir le code source, puis ils insèrent du code malveillant et téléchargent la nouvelle application au marché Android.

Avec cette méthode, il est beaucoup plus probable que les utilisateurs téléchargent ces applications reconditionnées par rapport à une nouvelle application malicieuse (Datta, Bonnet et Nikaein, 2012). Récemment, les chercheurs(Viennot, Garcia et Nieh, 2014) ont constaté que le quart de toutes les applications Google Play libres sont des clones.

En outre, toute application Android peut télécharger des exécutables contenants du code natif afin de l'exécuter. Ainsi les logiciels malveillants qui peuvent sembler à l'origine légitimes, peuvent télécharger du code malveillant, en cours d'exécution sans être détectés par les logiciels anti-malware. Cette lacune est considérée comme l'un des plus grands problèmes

qui subsistent en matière de sécurité Android (Fedler, Schütte et Kulicke, 2013) et elle engendre deux scénarios d'attaques.

Le premier scénario implique les applications légitimes qui téléchargent, pour des raisons bénignes, du code supplémentaire qui peut être remplacé par un code malicieux par l'attaquant. Dans ce scénario, plusieurs applications sont basées sur le même cadriciel, qui est installé sur le périphérique comme une application séparée. Ces applications contiennent du « stub code » qui leur permet de charger le code du cadriciel (Poeplau et al., 2014b). Alors, si un attaquant est capable d'installer une application qui prétend fournir le cadre commun, les applications basées sur ce cadre vont charger le code de l'attaquant au lieu celui du cadriciel légitime.

Le deuxième scénario implique les applications malveillantes, qui ne contiennent initialement aucun code malveillant, mais elles téléchargent du code malicieux supplémentaire après avoir été installées sur le périphérique (Poeplau et al., 2014b). Certains cadriciels mettent en œuvre la fonctionnalité d'auto-mise à jour, le téléchargement et l'exécution du code à partir des ordinateurs distants.

1.6.6 Les techniques pour surmonter les analyseurs

Une des caractéristiques de sécurité qui caractérisent des plateformes mobiles modernes est l'utilisation de l'environnement de bac à sable (*Sandboxing*) qui permet de freiner l'activité non désirée.

Le Sandboxing est généralement considéré comme une technique de détection d'intrusion souhaitable, alors qu'il possède des limites. Il est généralement moins efficace dans la détection d'attaques non génériques, par exemple, les logiciels malveillants conçus pour être activés sur une action spécifique de l'utilisateur ou pour déclencher un comportement malveillant après une période d'activité normale.

Le Sandboxing est aussi largement inefficace contre la pratique de charger une fonctionnalité malveillante dans une application légitime, sans modifier le code légitime existant. Le bytecode original reste donc intact, ce qui permettra d'échapper du système de détection.

Poeplau et al. (2014a) ont défini plusieurs techniques pour charger le code externe sur un dispositif à l'aide de chargeurs de classes. Ces derniers permettent d'extraire les classes à partir de fichiers dans des emplacements arbitraires, à travers le contexte des applications Android. Ce contexte est une classe abstraite dont la mise en œuvre est assurée par le système Android. Il permet l'accès aux ressources et aux classes spécifiques d'autres applications, par l'utilisation de code natif (à l'aide de la méthode d'exécution *exec* qui donne accès à un *Shell* système) et à travers l'installation moins furtive des fichiers *.apk* demandées par un fichier *.apk* principal.

Maier, Muller et Protsenko (2014), ont étudié un type d'attaques, « *Divide-and-Conquer* ». Ce type d'attaque permet de contourner les analyseurs dynamiques des appareils mobiles, tels que le bac à sable, les antivirus et l'analyseur « Google Bouncer », en utilisant une combinaison des empreintes digitales et de chargement dynamique de code (fichiers Dex ou librairies) qui peut être téléchargé de l'internet, copié depuis la carte SD, chargé depuis d'autres packages, emballé dans les ressources des applications ou crypté dans l'application.

Cette attaque divise le logiciel malveillant en une partie bénigne et une partie malveillante qui sera masquée dans l'analyse statique et dynamique. Le malware se comporte donc malicieusement sur les appareils réels, alors qu'il cache sa partie malicieuse sur l'environnement d'analyse.

Les auteurs ont démontré que l'attaque « Divide-and-Conquer » peut contourner facilement les bacs à sable mobiles, en implémentant un outil, « Sand-Finger », qui génère des empreintes digitales des systèmes d'analyse basés sur Android, pour les envoyer à un serveur

distant pour les analyser. Selon l'empreinte digitale, l'outil décide d'activer ou non le code malveillant. Les auteurs ont analysé les empreintes digitales des dix environnements d'analyse différents et ils ont conclu qu'ils peuvent facilement distinguer un environnement virtuel d'un dispositif réel.

Cependant, l'analyseur dynamique « Google Bouncer » de Google playStore bloque la connexion internet, ce qui empêche l'envoi de son empreinte digitale au serveur pour l'analyser. Pour échapper à l'analyse dynamique « Google Bouncer », le malware n'exécute pas son code malicieux sur une connexion internet manquante. Les auteurs ont démontré leur approche, en implémentant une application Android bénigne avec un « root exploit » compressé qui s'exécute lorsque l'application reçoit certaines commandes de serveur. Cette application a été bien mise dans « Play Store », ce qui montre qu'elle a dépassé facilement les contrôles de sécurité de « Google Bouncer ».

Les auteurs ont montré aussi que les échantillons de logiciels malveillants connus peuvent dépasser « Google Bouncer » en les modifiant légèrement. Ils ont créé 3 variantes du malware « AndroRAT ».

L'attaque « Divide-and-Conquer » peut aussi dépasser l'analyse statique, puisque cette technique ne peut pas détecter le chargement dynamique du code. En effet, le chargement du code est une pratique courante dans plusieurs applications Android bénignes.

1.7 Sécurité des systèmes embarqués

Au cours des dernières années, il y a eu un progrès énorme dans le domaine de la sécurité des logiciels. Plusieurs techniques ont été mises en place, tels que la génération automatique de signatures, l'évaluation des vulnérabilités et la détection des comportements malveillants.

Cependant, toutes ces techniques ne sont pas directement applicables dans le contexte des systèmes embarqués pour les raisons suivantes :

- Les systèmes embarqués sont généralement déployés dans des environnements hautement dynamiques et configurables.

- Les exigences fonctionnelles d'un système embarqué varient au cours du temps (exigences fonctionnelles d'un système embarqué déployé dans un changement défendu, selon la mission de l'opération).

- Généralement, un système embarqué est un réseau complexe de composants. Par conséquent, une faille ou une erreur dans un composant peut entraîner une cascade complexe d'événements dans le réseau.

- Les systèmes embarqués sont souvent déployés dans des applications critiques, où les conséquences de défaillances peuvent être désastreuses. Par conséquent, la récupération d'échecs est extrêmement importante dans le contexte des systèmes embarqués.

1.8 Conclusion

Ce chapitre présente un aperçu du système Android, ses mécanismes de sécurité et ses vulnérabilités. Il décrit aussi les principales attaques contre cette plateforme et en particulier l'attaque par injection du code qui engendre des mises à jour malicieuses qui, généralement, ne peuvent être détectées que par une analyse comportementale de l'application durant son exécution sur l'appareil. Le chapitre suivant présente une revue de la littérature des techniques et méthodes de détection d'anomalies sur les plateformes mobiles et en particulier sur Android.

CHAPITRE 2

REVUE DE LITTÉRATURE DES TECHNIQUES DE DÉTECTION DES ANOMALIES SUR ANDROID

2.1 Introduction

Ce chapitre présente une étude des travaux trouvés dans la littérature qui sont appropriés pour la détection d'anomalies sur Android.

Il commence par une description générale d'un système de détection d'intrusion IDS (*Intrusion Detection Systems)* et de ses composants, suivie d'un aperçu des techniques de détection d'intrusion communes pour les plateformes mobiles, qui seront présentées dans la section 2.2. Ces travaux vont être discutés et critiqués en les regroupant en trois catégories; l'analyse sur l'appareil mobile et l'analyse à distance, qui seront discutés à la section 2.3, les types de ressources surveillées dans la détection d'anomalies sur les environnements mobiles, qui seront discutés à la section 2.4 et le compromis entre la sécurité et la consommation des ressources, qui seront examinées dans la section 2.5. La section 2.6 va établir une comparaison entre les différents travaux connexes et notre approche. Enfin, la section 2.7 va conclure ce chapitre.

2.2 La détection d'intrusion

Un domaine de recherche actif et pratique est la détection d'intrusion qui est le processus d'observer le comportement du système pour identifier une activité inappropriée (Sabahi et Movaghar, 2008).

La définition de ce qui est inapproprié peut varier, mais elle est généralement destinée à décrire l'utilisation illégale. La sécurité du système et du réseau dépend fortement de l'efficacité des systèmes de détection d'intrusion qui émettent des alertes lorsque certains motifs prédéfinis d'attaque (IDS basé sur les signatures) ou de comportements potentiellement dangereux (IDS basé sur les anomalies) ont été détectés.

De nombreux systèmes de détection ont été présentés dans la littérature et plusieurs outils sont bien utilisés dans la pratique. Les premiers travaux sur la détection d'intrusion étaient dus à *Anderson* (Anderson, 1980) et *Denning* (Denning, 1987). Depuis lors, il est devenu un domaine très actif.

La plupart des systèmes de détection d'intrusion peuvent être classés en deux catégories: la détection d'anomalie et la détection basée sur les signatures. Aussi selon l'endroit où ces IDS sont positionnés, nous avons deux classes de systèmes de détection d'intrusion; basés sur le réseau et sur l'hôte.

2.2.1 Détection basée sur les signatures

L'approche basée sur les signatures (approche statique) (Shabtai et al., 2009) (Christodorescu et Jha, 2003), surtout utilisée par les compagnies d'antivirus, se repose sur une base de données des signatures (empreintes digitales) qui caractérisent l'activité de chaque malware. La détection consiste à chercher des éléments ou des motifs connus des attaques répertoriées dans la base de données des signatures des malwares (Griffin et al., 2009), comme l'indique la figure 2.1.

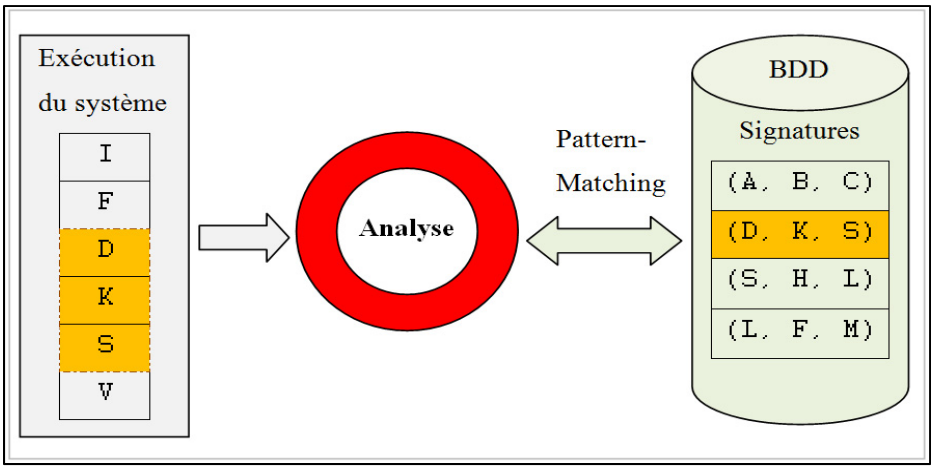


Figure 2.1 Détection de malwares basée sur les signatures

Cette technique peut être à son tour soit statique ou comportementale, selon la nature des signatures des malwares sur lesquelles elle repose. L'approche statique de détection d'intrusion basée sur les signatures s'appuie sur des propriétés structurelles du programme (par exemple des séquences d'octets ou des fonctions de hachage), alors qu'une approche dynamique s'appuiera sur les informations d'exécution (par exemple les systèmes vus sur la pile d'exécution) du programme.

En général, une approche statique tente de détecter les logiciels malveillants avant que le programme surveillé s'exécute. En revanche, une approche dynamique tente de détecter les comportements malveillants lors ou après l'exécution du programme. Il existe des techniques hybrides qui combinent les deux approches (Rabek et al., 2003). Dans ce cas, les informations statiques et dynamiques sont utilisées pour détecter les logiciels malveillants.

Fuchs, Chaudhuri et Foster (2009) proposent un cadre de détection, « ScanDroid », qui extrait des paramètres de sécurité à partir des fichiers « Manifest » des applications en cours d'exécution. Les caractéristiques extraites sont comparées au modèle généré pour identifier les malwares. Desnos et Gueguen (2011) proposent une méthode basée sur les signatures qui utilise les propriétés d'autorisation de chaque application. Ils construisent le graphe de flot de

contrôle grâce à ces caractéristiques collectées pour détecter les logiciels Android malveillants.
Tout en étant rapide et efficace, l'approche statique est réactive et pose plusieurs contraintes. En effet elle ne permet pas de détecter que les attaques répertoriées dont on possède les empreintes digitales. Ainsi des mises à jour fréquentes de la base de données des signatures sont requises pour que la détection soit plus précise. En outre, cette technique est limitée, principalement due au fait que les différentes techniques d'obfuscation et de cryptage peuvent être utilisées pour y échapper et rendre ainsi leur capacité à faire face aux logiciels malveillants polymorphes (qui changeant leur signature à chaque fois) limitée (Moser, Kruegel et Kirda, 2007).

2.2.2 Détection d'anomalies

En revanche, les systèmes de détection d'anomalies ADS (*Anomaly Detection System*) (détection heuristique ou détection basée sur le comportement) sont basés sur des règles qui sont soit déterminées par des experts ou par des techniques d'apprentissage machine qui définissent le comportement normal ou anormal de chaque programme surveillé (Jacob, Debar et Filiol, 2008).

Cette approche comprend deux phases ; une phase d'apprentissage et d'une phase de détection, comme le montre la figure 2.2. La première phase consiste à créer, automatiquement ou manuellement, un profil pour le programme surveillé (un modèle normal). La façon dont le profil est créé n'est pas importante tant que ce profil définit avec précision les caractéristiques du programme surveillé. Puis, dans la phase de détection, si un comportement s'écarte trop du profil enregistré, le système génère une alarme indiquant que l'activité du programme est malicieuse.

L'avantage principal de cette technique est qu'elle est capable d'identifier des attaques inconnues. En définissant le comportement normal prévu, toute anomalie peut-être détectée,

si elle fait partie d'une attaque connue ou non. Cependant, le comportement normal peut changer au fil du temps, résultant en une génération élevée de fausses alarmes, d'où la nécessité de reconstruire périodiquement le modèle normal.

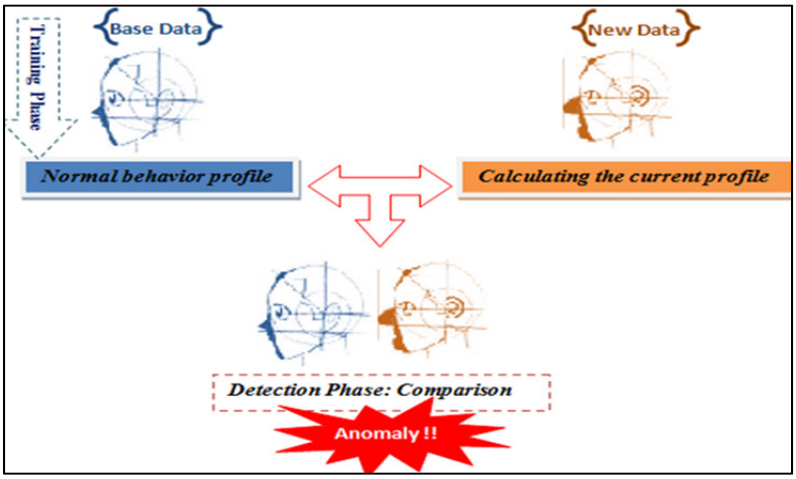


Figure 2.2 Détection de malwares
basée sur le comportement

Récemment, des algorithmes de classification ont été utilisés pour automatiser et étendre l'idée des méthodes comportementales. Dans ces méthodes, le code binaire d'un fichier ou le comportement de l'application en cours d'exécution est représenté et des classificateurs sont utilisés pour apprendre des modèles afin de classer de nouvelles applications (inconnues) comme malveillantes ou bénignes (Moskovitch, Elovici et Rokach, 2008).

Dans le travail de Xie et al. (2010), les auteurs utilisent une approche probabiliste grâce à la corrélation des entrées de l'utilisateur avec les appels système pour construire les profils de comportement sur les téléphones intelligents et détecter les activités anormales. Leur solution « *pBMDS* » utilise le modèle de Markov caché pour apprendre le comportement de l'application. La caractéristique la plus distinctive de la solution proposée est que sa capacité de détection de malware se concentre sur la reconnaissance de comportement non humain au lieu de compter sur les signatures d'attaque connues pour identifier les logiciels malveillants.

Bose et al. (2008) proposent un modèle qui retrace les comportements normaux des applications et utilise les *Support Vector Machines* (SVM) pour classer les comportements anormaux.

Le Tableau 2-1 présente une comparaison entre l'approche de détection basée sur les signatures et la détection d'anomalies.

Tableau 2-1
Comparaison entre la détection basée sur les signatures et la détection d'anomalies

Détection	Types d'attaques	Fausses alarmes	Temps de mise en place/ détection	Mise à jour	Taille occupée	Difficulté
Signatures	Connues	Peu	Immédiatement	Oui	Énorme	Familles de malwares
Anomalies	Connues Inconnues	Plusieurs	Période de formation	Oui	Moyenne	Dans l'analyse

2.3 Déploiement de la détection d'anomalies sur Android

De nos jours, plusieurs techniques d'analyse et de détection des logiciels malveillants basées sur le comportement ont été proposées pour les appareils mobiles et en particulier pour le système Android. On trouve principalement l'analyse comportementale basée sur l'hôte et l'analyse renforcée par un serveur distant ou basée sur lui.

2.3.1 Analyse basée sur l'hôte

L'analyse basée sur l'hôte est totalement indépendante de toute autre source d'analyse externe.

Sanz et al. (2013b) ont présenté une méthode de détection d'anomalie sur la plateforme Android. Les auteurs créent un modèle normal à partir des caractéristiques textuelles de l'application, représentées sous forme d'un modèle vectoriel. La déviation du modèle normale est calculée en utilisant la valeur moyenne, la valeur maximale et la valeur minimale des trois distances ; la distance de Manhattan, la distance euclidienne et la similarité cosinus. Les auteurs ont effectué cinq validations croisées sur 333 applications légitimes et ils ont testé leur approche avec différents seuils sur 333 applications malicieuses recueillies auprès de la société *VirusTotal*. *VirusTotal* offre une série de services appelés « *Virus-Total Malware Intelligence Services »,* qui permet aux chercheurs d'obtenir des échantillons de leurs bases de données. Cette approche a fourni les meilleurs résultats avec une précision de 83,51 % en utilisant la règle de la combinaison moyenne des distances.

Un outil d'analyse dynamique « *DroidTrace* » est proposé par Min, Mingshen et Lui (2014). Cet outil utilise ptrace (manpagesfr, 2008) pour surveiller les appels système sélectionnés. Dans leur solution, les auteurs se sont concentrés sur la détection de chargement dynamique qui permet à une application de charger une bibliothèque dynamique dans la mémoire pendant qu'elle s'exécute. Leur capacité d'exécution à l'avance permet de déclencher automatiquement presque tous les comportements de chargement dynamiques dans l'application. DroidTrace peut fonctionner sur l'appareil lui-même.

Le travail de Shabtai et al. (2014) définit un système dynamique pour détecter les écarts significatifs de comportements des applications par rapport à leurs comportements normaux en se basant sur le trafic réseau. Ils ont ciblé des applications populaires injectées par du code malveillant ainsi que les logiciels malveillants avec des capacités d'auto-mise à jour. Les auteurs ont utilisé un algorithme d'apprentissage semi-supervisé pour créer les comportements normaux et pour mesurer l'écart par rapport à ces derniers. Si cet écart dépasse un seuil prédéfini, l'application sera considérée comme un malware. Leur système proposé se compose de six composantes principales: Une interface graphique (GUI) qui communique avec l'utilisateur et affiche des alertes, une alerte maître qui génère des alertes

et traite les réponses des utilisateurs, un module d'extraction des caractéristiques, un module d'agrégation des caractéristiques extraites, un module d'apprentissage local qui crée les modèles normaux, un détecteur d'anomalie qui analyse le comportement du réseau pour les applications et détecte les écarts par rapport à un comportement normal.
L'algorithme d'apprentissage, dans ce travail est basée sur l'approche de l'analyse des caractéristiques croisées (*cross-feature analysis*) qui transforme un problème d'apprentissage semi-supervisé en un ensemble de problèmes d'apprentissage supervisé pour lesquels il existe des solutions bien établies (Huang et al., 2003).

En outre, les processus de détection et d'apprentissage ont eu lieu localement sur le périphérique lui-même. Les auteurs ont mené plusieurs séries d'expériences en utilisant le trafic des applications dans les dispositifs de huit utilisateurs volontaires sur une période de temps définie. Ils ont également utilisé 10 malwares auto-écrits et 5 malwares réels pour leurs tests. Leurs expériences montrent que la modélisation du trafic d'une application en utilisant uniquement des fonctionnalités au niveau applicatif est possible, aussi ils ont déduit 7 caractéristiques discriminatoires du réseau au niveau applicatif.

Pour l'analyse des caractéristiques croisées, les auteurs utilisaient l'arbre de décision C4.5 pour les fonctions catégorielles et plusieurs techniques d'apprentissage automatique pour les fonctions numériques, y compris les classificateurs basés sur la régression, table de décision, SVM et l'arbre de décision/régression.

Les auteurs ont mesuré les taux de TP (*True Positive*), de FP (*False Positive*), d'exactitude et le temps de détection. Leurs résultats ont montré que l'arbre de régression et le tableau de décision sont les plus performants en terme de mesures. Les auteurs ont fait valoir l'efficacité de leur solution en termes de la mémoire et CPU, ce qui encouragerait son déploiement sur les périphériques directement.

Dans le travail de Arp et al. (2014), « *DREBIN* », un système léger pour la détection de logiciels malveillants sur Android a été présenté. Ce système combine les concepts de l'analyse statique et d'apprentissage machine et peut fonctionner sur l'appareil lui-même. L'analyse statique a été utilisée pour recueillir autant de fonctionnalités que possible des fichiers manifeste et « .dex ». L'ensemble des caractéristiques extraites comprenait les composants matériels, les permissions, les composants d'application, les intentions filtrées, les appels d'API restreints et suspects, et les adresses MAC. Les différentes combinaisons de ces caractéristiques ont été mappées sur un espace vectoriel commun pour qu'elles puissent être analysées géométriquement. Ensuite, SVM a été appliqué pour séparer les comportements malveillants de ceux normaux. Les caractéristiques relatives à chaque application analysée ont été rapportées à l'utilisateur.

Les auteurs ont évalué leur solution de détection en utilisant 123 453 applications bénignes et 5560 applications malveillantes. Ils ont effectué trois séries d'expériences, dans la première, ils ont mesuré les taux de TP et de FP, et ont constaté que leur approche a surpassé les autres techniques d'analyse statique et que sa performance était comparables à celle des moteurs antivirus. Dans la deuxième expérience, ils ont étudié sa capacité à expliquer les résultats obtenus pour 4 familles de logiciels malveillants. Pour chaque échantillon de ces familles, ils ont identifié les caractéristiques qui contribuent le plus à la décision de classement et ils ont calculé la moyenne des résultats pour tous les cas d'une famille. Dans la troisième expérience, ils ont exécuté une instance *DREBIN* sur plusieurs téléphones intelligents et ont constaté que le temps d'exécution nécessaire pour analyser une demande était, en moyenne de 10 secondes.

2.3.2 Analyse à distance

L'analyse à distance permet d'exécuter le processus de détection en dehors de l'appareil, sur un serveur distant. Généralement, les techniques de détection qui sont traitées à distance, sont complexes et nécessitent une grande puissance. Ce type d'analyse est la solution la plus

simple est souvent la plus employée par les systèmes mobiles de détection d'intrusions et la plus adéquate pour les systèmes à ressources limitées puisque toute la charge de processus sera traitée à distance et non sur l'appareil. Cependant, le système de détection dépend toujours d'une connexion fiable et stable, sinon l'appareil sera pratiquement totalement vulnérable.

L'approche de Sahs et Khan (2012) présente un système d'apprentissage automatique pour la détection des logiciels malveillants sur les appareils Android. Le système extrait un certain nombre de caractéristiques statiques (les permissions du fichier « Manifest » d'Android avec des informations de graphe de flux de contrôle à partir le bytecode) à l'aide de l'outil « Androguard », puis il entraine une seule classe SVM sur un ensemble de 2081 applications bénignes, d'une manière hors-ligne (off-dispositif), afin de tirer parti de la puissance de calcul élevée d'un serveur ou d'un cluster de serveurs. Cette approche génère un taux de faux positifs près de 50 %, ce qui la rend intenable.

CrowDroid (Burguera, Zurutuza et Nadjm-Tehrani, 2011) est une application cliente, basée sur une approche antérieure qui analyse le comportement des applications. Les auteurs ont appliqué l'algorithme de clustering « K-means » avec k=2 pour détecter les malwares reconditionnés. Le Malware reconditionné est créé en prenant une application bénigne et la reconditionnant avec un code malveillant supplémentaire. L'ensemble du processus d'analyse est effectué sur un serveur distant, responsable de la gestion et de l'analyse de toutes les informations recueillies auprès des dispositifs clients *Crowdroid*. Les informations de base de dispositifs sont stockées dans une base de données centrale. Cette approche a été testée avec quatre malwares crées par les auteurs et deux véritables logiciels malveillants. *CrowDroid* a identifié avec succès tous les logiciels malveillants créés par les auteurs et il a produit un taux de faux positifs de 20 % sur l'un des deux véritables logiciels malveillants.

Bien que ces résultats soient satisfaisants, avec *CrowDroid*, on pourrait soumettre intentionnellement des données incorrectes au système en laissant l'ensemble de données corrompues. Aussi, la communication entre le client et le serveur est basée sur le protocole

FTP sans se focaliser sur la protection de la confidentialité des données transférées. Si une attaque renifle et manipule le trafic dans le processus de communication, elle peut conduire à des erreurs de classification. La mise en place des mécanismes de chiffrement pour assurer l'intégrité des données et l'authenticité de l'expéditeur sur l'appareil mobile pourrait avoir une charge supplémentaire pour le processeur, ce qui entraîne une décharge rapide de la batterie.

Un sandbox, *AASandbox*, destiné à être géré comme un service dans l'infonuagique pour analyser les applications Android, est créé par Bla et al. (2010). Contrairement à d'autres services de sandbox mobiles, *AASandbox* est une approximation hybride basée sur une analyse dynamique et statique. Dans la partie statique, l'outil décompresse les fichiers d'installation et désassemble les exécutables correspondants. Ceci peut être utilisé avant l'exécution pour une préanalyse rapide des motifs malveillants. La partie dynamique est basée sur l'analyse des logs pour les interactions de bas niveau obtenues lors de l'exécution. Pour valider leur approche, Bla et al. (2010) ont utilisé 150 applications bénignes et une application malicieuse auto-créée qui effectue une attaque de déni de service (Dos : *Deny of Service*).

Amos, Turner et White (2013) visent à réaliser une détection rapide, dynamique et à grande échelle des logiciels malveillants mobiles en utilisant des algorithmes d'apprentissage machine. Ils ont présenté leur cadre intitulé « STREAM » qui est un profileur d'applications distribués. Il comprend un serveur maître qui distribue les tâches de profilage à un ensemble de nœuds. Ces nœuds à leur tour, distribuent les tâches entre les dispositifs et /ou les émulateurs pour exécuter les algorithmes de détection en parallèle. Le maître STREAM collecte alors les vecteurs de caractéristiques à partir des dispositifs et des émulateurs et gère la formation des classificateurs et l'évaluation. Dans leur cadre, ils ont évalué *Random Forest*, les réseaux de Perceptron multicouches, les réseaux bayésiens naïfs, la régression logistique et les arbres de décision J48. Dans leurs expériences, ils ont utilisé une base de données des malwares connus afin d'évaluer leur approche. En outre, ils ont utilisé l'outil

« *MonkeyRunner* » pour explorer le comportement de l'application. Les auteurs ont mesuré le taux de TP, le taux de FP et le temps de détection, ainsi que le pourcentage des cas correctement classés. Leurs résultats ont montré que les réseaux bayésiens naïfs étaient les plus performants, tandis que l'algorithme de régression logistique était le moins performant en termes de mesures. En outre, ils ont formé leurs algorithmes sur 1330 applications malveillantes et 408 applications bénignes.

Une approche de collaboration pour l'analyse des applications et la détection de logiciels malveillants sur Android, est proposée par Schmidt et al. (2009). Le système proposé fournit essentiellement une fonctionnalité d'analyse sur l'appareil, de collaboration et d'analyse à distance en utilisant une architecture client-serveur. Le client recueille des données sur l'appareil pour la collaboration ou l'analyse à distance. Ces données peuvent être échangées entre deux clients mobiles dans un esprit de collaboration ou envoyées à un serveur distant lorsque la détection sur l'appareil n'est pas possible. Le serveur, à son tour, peut envoyer les résultats de détection au client. En outre, il peut envoyer des commandes de reconfiguration du client.

2.4 Type de ressources monitorées

Le type de ressources analysées est toujours utile et significatif pour la construction de modèle normale. Les paramètres extraits affectent les résultats de test. Par conséquent, dans cette section, nous présentons les caractéristiques étudiées dans les travaux antérieurs. Nous classons ces paramètres en deux catégories; statique et dynamique.

2.4.1 Paramètres statiques

L'analyse des paramètres statiques est une méthode populaire pour analyser les malwares. On peut atteindre l'objectif en analysant le contenu de chaque application. Cette méthode d'analyse peut réduire le coût et améliorer la performance. De nombreux chercheurs

suggèrent d'utiliser des paramètres statiques pour détecter les comportements malveillants sans réellement exécuter l'application. Parmi ces paramètres, on trouve le bytecode d'Android. En effet, la représentation des binaires Android est sémantiquement plus riche que les binaires bureautiques communes. Par exemple, l'extraction des chaînes dans les binaires des bureaux est complexe et, généralement, les auteurs de malware utilisent des techniques d'obscurcissement pour cacher des informations pertinentes. Alors que l'obscurcissement de chaînes dans les binaires Android est plus difficile, compte tenu de la structure interne des fichiers binaires dans cette plateforme.

Le travail de Sanz et al. (2013b) consiste à désassembler l'application Android avec l'outil « smali » et extraire le code opérationnel « const-chaîne » pour construire un modèle de détection d'anomalies qui mesure les écarts à la normalité (c'est à dire, les applications légitimes). Le bytecode contient aussi les appels API (Interfaces pour l'accès programmé aux applications) qui fournissent des moyens aux applications pour interagir avec l'appareil. L'inspection statique de ces appels donne donc des informations sur leurs activités d'exécution.

Un cadriciel d'analyse statique des applications Android, appelé *DroidMat*, est proposé par Dong-Jie et al. (2012). Cet outil permet d'extraire et d'analyser les informations de fichier manifeste, y compris les appels API et les permissions pour identifier le comportement de l'application. Puis, il applique un algorithme d'apprentissage k-means pour améliorer la capacité de modélisation des logiciels malveillants. La performance de détection a été évaluée sur un ensemble de données de 1500 applications bénignes et 238 applications malveillantes tout en la comparant à un autre travail connexe appelé *Androguard* (Desnos, 2011).

Les résultats des tests de DroidMat étaient prometteurs avec une précision d'environ 97 %. Cependant, le coût de l'analyse dépend de la taille et du nombre de fichiers et les ressources limitées de l'appareil ne sont pas prises en compte par cet outil. Aussi, DroidMat n'est pas un

modèle dynamique dans le sens où ses seuils restent constants alors que les algorithmes de détection sont en cours d'exécution.

L'approche de Bla et al. (2010) scanne aussi le bytecode avec les appels API et le fichier « Manifest » de l'application de test pour des motifs spéciaux. Leur approche AASandbox surveille l'utilisation de la JNI (Java Native Interface) qui peut être utilisée pour charger dynamiquement les bibliothèques natives, la méthode « Sytem.getRuntime().exec(..) » qui permet d'exécuter des commandes dans des processus séparés, les mécanismes de communications inter processus (IPC : *Inter-Process Communication*) et les permissions déclarées dans le fichier « Manifest » afin de déterminer quelles autorisations seront accordées lors de l'installation.

En effet, les permissions jouent un rôle important dans l'attribution des droits d'accès à certains éléments restreints du système d'exploitation Android. Ces éléments comprennent l'accès à divers matériels (par exemple, GPS, caméra), aux caractéristiques sensibles du système d'exploitation (par exemple, les contacts) et l'accès à certaines parties exposées d'autres applications. L'utilisateur doit accorder toutes les autorisations demandées par l'application au moment de l'installation. Diverses approches de détection des logiciels malveillants Android sont basées sur l'analyse des combinaisons de permissions demandées par des malwares.

Par exemple Aung et Zaw (2013) effectuent une analyse basée sur les autorisations requises par les applications Android du marché « Google Play ». Ils extraient les autorisations définies à partir des fichiers « AndroidManifest.xml » de 500 applications Android et les réduisent en sélectionnant celles les plus pertinentes en utilisant la méthode d'entropie relative. Puis ils appliquent les algorithmes de classification tels que l'arbre de décision, l'analyse d'arbre de classification et de régression CART (*Classification And Regression Tree*) et les forêts d'arbres décisionnels RF (*Random Forest*) sur l'ensemble de données réduit. Les performances ont été comparées en termes de taux de vrai positif, faux positif,

précision et rappel. Selon leurs résultats, l'arbre de décision et les forêts d'arbres décisionnels sont plus performants que l'analyse d'arbre de classification et de régression, en présentant des taux de précision plus élevés et des taux de faux positifs plus faibles.

L'approche de Sheen, Anitha et Natarajan (2015) utilise également les permissions et les appels API pour analyser les applications Android. Cette approche applique trois algorithmes : la loi du $\chi 2$ (CS : *Chi square*), Relief (RF) et la divergence de Kullback-Leibler (IG : *Information Gain*) pour sélectionner les appels API et les permissions les plus exigeants pour la détection des malwares. Par la suite, elle applique différents classificateurs (Naive bayes, J48, SVM et IBk) pour chaque groupe de données. Les classificateurs collaborent les uns avec les autres pour aboutir à une décision finale. Les auteurs ont évalué les performances de la méthode de collaboration sur les 20 principaux paramètres qui ont été sélectionnés à partir des données extraites de 1073 applications malicieuses et 904 applications bénignes. Les résultats montrent que cette approche offre une précision de 98,31 %.

Une méthode de détection des malwares sur Android, appelée *PUMA* est proposée par Sanz et al. (2013a). Cette approche est basée sur l'apprentissage machine qui utilise des informations obtenues à partir des permissions requises par l'application. Pour évaluer leur méthode, les auteurs ont recueilli 1811 applications bénignes de différentes catégories depuis l'Android Market et 249 applications malveillantes en utilisant « VirusTotal Malware Intelligence Services ». Les caractéristiques utilisées pour représenter chaque échantillon sont basées sur l'ensemble des permissions requis par l'application. Grâce à ces informations, les auteurs ont évalué huit algorithmes de classification disponibles dans le cadriciel WEKA. Pour chaque classificateur, ils ont réalisé 10 validations croisées et ont conclu que Random Forest a fourni les meilleurs résultats avec une précision de 86 %.

2.4.2 Paramètres dynamiques

Il y a plusieurs caractéristiques dynamiques utilisées par les travaux connexes, nous listons les caractéristiques dynamiques comme suit:

2.4.2.1 Les appels systèmes

La capture et l'analyse des appels fourniront des informations précises sur le comportement de l'application. Le but de détourner ces appels système, est de générer un fichier de sortie avec tous les événements générés par l'application Android. Pour cela, la surveillance des appels système est un moyen pratique pour détecter les logiciels malveillants. Cette technique d'analyse a été largement utilisée dans la littérature.

L'approche de Schmidt et al. (2009) extrait des librairies et des appels de fonction système depuis environ 240 logiciels malveillants qui ciblaient les systèmes Linux, et moins de 100 commandes système Linux à partir d'un appareil Android. Puis elle les compare aux logiciels malveillants exécutables pour classer les applications. Ils ont appliqué trois classificateurs (règle inductrice, le plus proche voisin, et arbre de décision) à quelques sous-ensembles de fonctionnalités. Toutes leurs configurations avaient un taux de vrais positifs égal ou supérieur à 80 % avec un taux de faux positifs supérieur à 10 %.

L'analyse dynamique de l'approche *AASandbox* de Bla et al. (2010), permet de journaliser le comportement de l'application au niveau du système. Elle consiste à collecter tous les appels système et leurs occurrences, y compris ceux qui sont effectués par des processus autres que l'application en cours d'analyse, afin de contrôler l'état complet du système. Les appels système pour chaque application analysée, sont stockés dans un fichier séparé.

L'approche de Burguera, Zurutuza et Nadjm-Tehrani (2011) utilise un outil disponible sous Linux appelé « Strace » pour recueillir les appels système et générer ensuite un fichier qui

contient des informations sur les fichiers ouverts et accessibles, l'horodatage d'exécution et le nombre de chaque appel système exécuté par l'application. Ce fichier a été utilisé pour représenter le comportement de chaque exécution d'application Android. Les auteurs ont utilisé un système central qui recueille les fréquences de plusieurs appels système à partir de plusieurs utilisateurs exécutant l'application sur différents appareils. L'inconvénient de cette approche est le fait que le système sépare toujours les vecteurs de données d'appel système en deux groupes, même s'il n'y a aucun malware. La cartographie des clusters changerait radicalement chaque fois qu'un vecteur d'exécution malveillante entre dans le jeu de données. Ce problème nécessite une certaine vérification manuelle ou une analyse automatique plus poussée.

Kevin Joshua Abela (2013) ont proposé une approche dynamique de détection d'anomalie « AMDA », qui fonctionne sur l'appareil Android lui-même. Ce système détermine le comportement malveillant de comportement bénin en surveillant et en capturant 15 appels système sur lesquels on applique l'apprentissage machine. Tout d'abord, « AMDA » collecte ces caractéristiques des applications bénignes et malveillantes pour les utiliser dans l'apprentissage machine afin de fournir une base de référence de comportements. Chaque application surveillée est passée ensuite à travers le module d'analyse comportementale qui va la classer comme malveillante ou bénigne. Les algorithmes utilisés pour ce module comprennent l'algorithme *Naïve Bayes*, arbres de décision, régression logistique et RF. Les auteurs ont comparé les performances de ces quatre algorithmes en se basant sur le pourcentage des cas correctement classés, les statistiques de Kapp, la précision et les taux de vrais positifs et de faux positifs. Les tests ont été faits sur 68 applications malicieuses et 36 applications bénignes et les résultats ont montré que l'algorithme « *Random Forest* » est le plus performant avec 71,15 % des cas correctement classés.

Parmi les travaux liés à la détection des intrusions en surveillant les appels système, on trouve les méthodes qui utilisent des fenêtres de taille fixe ou variables. Ces méthodes génèrent un ensemble de courtes séquences à partir d'une suite de données, en faisant glisser

une fenêtre coulissante à travers les traces d'exécution d'un programme. En se basant sur cette technique, le travail de Forrest et al. (1996) définit un comportement normal en tant qu'un ensemble de courtes séquences d'appels système d'un processus en cours d'exécution. Une base de données de ces courtes séquences est initialement construite en faisant glisser une fenêtre de longueur k sur la trace. Les anomalies sont détectées s'il y a suffisamment des sous-séquences de longueur k dans la séquence d'essai qui ne sont pas enregistrées dans la base normale.

L'approche *STIDE* de Hofmeyr, Forrest et Somayaji (1998), extrait aussi un ensemble de courtes séquences d'appels système en utilisant une fenêtre de longueur k, afin de construire la base de données des séquences normales. Chaque séquence de test se voit attribuer un score d'anomalie défini comme l'inverse de la distance de Hamming normalisé à la longueur de la fenêtre. Ce score indique si la séquence de test est anormale.

Une autre approche à base des fenêtres est proposée par Somayaji et Forrest (2000). Cette approche conserve deux profils; un profil qui contient des séquences normales d'appels systèmes sous forme de courtes séquences de longueur k et un autre qui contient un ensemble de séquences de test. L'approche retarde l'exécution des appels système qui dérivent du comportement normal, en utilisant une fonction exponentielle.

Des fenêtres de tailles variables, qui peuvent prédire le symbole suivant dans la trace, ont été explorées dans le travail de Guofei et al. (2007). Cette approche permet d'extraire des séquences d'appel système, de différentes tailles, conditionnées par un seuil de généralisation. Nous allons décrire cette méthode avec plus de détails dans le chapitre 3.

Parmi les travaux liés à la détection d'intrusions qui utilisent les appels système, on trouve également les méthodes basées sur les états, dont le travail de Michael et Ghosh (2002) fait partie. Les auteurs ont proposé une technique de génération automatique des machines à états finis (FSM : F*inite-state machine*). La FSM est construite à partir de données d'audit d'un

programme. Elle caractérise les courtes séquences d'appel systèmes qui sont collectées dans la phase d'apprentissage. Dans la phase de test, l'outil détecte les nouvelles séquences qui n'ont pas été vues au cours de la première phase. Les auteurs ont proposé aussi un modèle prédictif qui prédit la séquence suivante sur la base de la probabilité des séquences dans l'ensemble de données d'apprentissage. Pendant la formation, le système calcule les probabilités de transition d'un appel système à un autre, puis, dans la phase de détection, tout écart par rapport à cette probabilité estimée est considéré comme une anomalie.

2.4.2.2 La consommation de la batterie

L'utilisation de la batterie a été largement explorée par plusieurs études afin d'identifier les activités anormales dans l'appareil mobile. Les chercheurs ont essayé de créer des profils de comportement basés sur l'utilisation de la batterie engendrée par les applications légitimes ou par les activités malveillantes.

Comme l'application malveillante a tendance à utiliser plus la batterie, une méthodologie intéressante a été proposée en 2006 par Grant (2006), où les auteurs proposent un IDS basé sur la batterie. Cette solution identifie les comportements anormaux de l'énergie afin de détecter une variété d'attaques. Elle comprend deux modules: un module de détection d'intrusion qui mesure l'énergie consommée sur une période de temps (basé sur l'hôte), et un module d'analyse des signatures de puissance qui compare ces signatures avec une courte liste des signatures d'attaque connues.

En 2008, une étude similaire a été réalisée par Kim, Smith et Shin (2008). L'étude propose un cadre de détection des logiciels malveillants qui a une bonne connaissance des besoins en énergie d'une activité. L'idée est de suivre et d'analyser les menaces d'épuisement d'énergies inconnues. Un analyseur de données génère une signature pour l'utilisation de l'énergie produite à partir de l'histoire de la consommation d'énergie de l'appareil mobile et la compare à une activité malveillante détectée. Le cadre est composé d'un moniteur de

puissance, pour recueillir les échantillons de puissance et construire un historique de consommation d'énergie à partir des échantillons prélevés, et d'un analyseur de données, afin de générer une signature d'énergie à partir de l'historique déjà construit.

Une autre approche basée sur la surveillance de la consommation d'énergie est présentée par Liu et al. (2009) pour observer la consommation d'énergie et détecter les logiciels malveillants. Les auteurs ont proposé « *VirusMeter* » qui détecte un comportement anormal par une consommation anormale d'énergie. L'idée derrière cette approche est que toute activité malveillante consommerait plus de batterie. Un modèle de puissance centré sur l'utilisateur est construit en enregistrant et en caractérisant la consommation d'énergie de chaque activité légitime. « *VirusMeter* » surveille les activités dans le téléphone et utilise des API fournies par la plateforme mobile pour recueillir la capacité restante de la batterie. Sur la base des données recueillies, il calcule la quantité de batterie que l'application peut consommer et la compare avec le modèle de puissance. Si la différence dépasse le seuil, le système déclenche une alarme. Les résultats des expérimentations montrent que *VirusMeter* pourrait détecter les activités malveillantes avec un taux de détection moyen proche de 80 % pour les différents cas avec un taux de faux positifs allant de 4,3 % à 10 %.

Cependant, il est difficile de construire un modèle de puissance et de collecter la consommation d'énergie en temps réel. Aussi, leur système a été prototypé sur un appareil mobile relativement vieux et il n'est pas clair si leur approche fonctionne d'une manière efficace sur un téléphone intelligent moderne qui exécute plusieurs applications capables de vider rapidement la batterie d'un dispositif.

Ma et al. (2013) traite l'épuisement anormal de la batterie sur les appareils mobiles. Les auteurs construisent un outil nommé, « *eDoctor* », pour aider les utilisateurs à diagnostiquer et à réparer ces problèmes en capturant les comportements anormaux des applications, qui peuvent ensuite être utilisés pour identifier une application anormale.

La surveillance continue de ce paramètre et sa comparaison avec un modèle de consommation d'énergie normal permettent de détecter des anomalies. Cependant, ces techniques ne sont efficaces que contre les attaques qui visent l'épuisement de la batterie.

2.4.2.3 Utilisation de la mémoire et du processeur

L'approche de Hyo-Sik et Mi-Jung (2013) introduit un système de détection de logiciels malveillants sur Android, basé sur l'apprentissage machine. Le système surveille 32 paramètres classés en sept catégories, telles que le réseau, SMS, CPU, alimentation, processus, la mémoire et la mémoire virtuelle. Les auteurs ont également effectué une évaluation de la performance des différents classificateurs d'apprentissage machine telle que les forêts d'arbres décisionnels (Randorm Forest), le naïf bayésien, la régression logistique et les séparateurs à vaste marge (SVM) sur un ensemble de 30 applications bénignes et cinq logiciels malveillants. Leurs résultats expérimentaux ont montré que le rendement des forêts d'arbres décisionnels était le plus approprié dans l'aspect de TP/FP. Bien que les séparateurs à vaste marge ont montré de faibles performances.

Les travaux de Shabtai et al. (2012) et de Amos, Turner et White (2013) se basent sur l'extraction des caractéristiques dynamiques, y compris l'activité de la mémoire et la charge de la CPU afin de classer les applications Android. Les auteurs appliquent plusieurs classificateurs y compris les arbres de décision, bayésiens naïfs et les réseaux bayésiens. Les expériences de Shabtai et al. (2012) montrent aussi un taux de précision allant de 44 % à 100 % avec 10 % de dégradation des performances dans le pire scénario, c'est à dire, 8 classificateurs différents avec 30 caractéristiques. Cependant, le meilleur classificateur dans Andromaly a abouti à un taux de faux positifs de plus de 10,4 %. Aussi, il n'est pas clair comment, cette performance a été mesurée et dans quelles conditions. Leur approche est limitée aussi par le fait que leur évaluation a été effectuée seulement avec 4 logiciels malveillants auto-écrits, non réels.

L'approche de Amos, Turner et White (2013) a été évaluée sur un ensemble de données plus vaste: 1330 logiciels malveillants et 408 applications bénignes. Leurs expériences montrent un taux de précisant allant de 82 % à 95 %. Comme pour « Andromaly », leur méthode souffre d'un taux de faux positifs élevé (de plus de 15 % pour l'ensemble de leurs configurations).

2.4.2.4 Autres

Zhao et al. (2011) proposent « *AntiMalDroid* », un cadre de détection des logiciels malveillants basé sur le comportement, il utilise le classificateur *Support Vector Machine* (SVM) formé à partir de 100 applications normales et trois malwares. *AntiMalDroid* peut détecter les logiciels malveillants et ces variantes à l'exécution en surveillant les séquences de comportements journalisées et étendre dynamiquement la base des caractéristiques malveillantes. Les résultats expérimentaux indiquent que leur approche fournit un taux de détection élevé et un faible taux de faux négatifs et de faux positifs.

Gilbert et al. (2011) proposent « *AppInspector* », qui retrace deux flux de données explicites et implicites ainsi que les flux de contrôle pour trouver les violations de sécurité. *AppInspector* surveille l'utilisation d'informations sensibles par l'application et vérifie les comportements suspects, comme la suppression des données de l'utilisateur.

Enck et al. (2014) proposent « *TaintDroid* », une extension de la plateforme de téléphone mobile Android qui permet de suivre le flux des données sensibles. *TaintDroid* surveille en temps réel la façon avec laquelle les applications tierces accèdent et manipulent des données personnelles et confidentielles des utilisateurs.

Suivant ce bref survol, nous avons constaté qu'il y a de nombreuses approches pour détecter les logiciels malveillants. Nous avons considéré que la surveillance des appels système est l'une des techniques les plus précises pour déterminer le comportement des applications

Android, car elle fournit des informations de bas niveau plus détaillées. Nous sommes conscients que les techniques d'analyse des appels API, de suivi des flux d'informations ou de la surveillance du réseau peuvent contribuer à une analyse plus approfondie des logiciels malveillants, en fournissant des informations plus utiles sur le comportement des programmes malveillants et des résultats plus précis. D'autre part, une surveillance de plusieurs caractéristiques consommera plus de ressources systèmes.

2.5 Compromis entre la détection et la consommation des ressources

Malgré son efficacité, l'analyse comportementale basée sur l'hôte nécessite beaucoup de ressources en termes de processeur, mémoire et énergie, ce qui rend difficile son déploiement sur les périphériques mobiles à ressources limitées.

Il existe peu de travaux dans la littérature qui aient essayé de définir un compromis entre la détection et la consommation d'énergies.

Bickford et al. (2011) ont étudié ce compromis selon deux facteurs: la surface d'attaques que le détecteur de logiciels malveillants couvrira et la fréquence de contrôle. En effet, en surveillant une surface d'attaque plus petite, la vérification des structures de données sera plus fréquente, présentant ainsi une petite fenêtre de vulnérabilité aux attaquants, ce qui maximise donc la sécurité. Aussi, l'énergie dépensée au cours de la vérification d'une partie de données est nettement inférieure à celle dépensée pour la vérification de la totalité des données. De même, la fréquence de contrôle influe sur l'énergie dépensée et sur la sécurité, en effet, le parcours fréquent réduit les vulnérabilités du système d'exploitation (présente une petite fenêtre de vulnérabilité), alors qu'il consomme beaucoup plus d'énergie. Lors de leurs expérimentations, les auteurs ont utilisé deux types de détecteurs de Rootkits et un hyperviseur qui sépare le domaine d'utilisateur et le domaine de confiance dans lequel s'exécutent ces deux détecteurs de Rootkits.

Le compromis raisonnable entre l'efficacité énergétique et la fenêtre de vulnérabilité a été réalisé à travers le point d'intersection entre les courbes correspondantes (T=30 secondes). Leur stratégie, offre également l'avantage supplémentaire de ne pas lier l'appareil à la sécurité fixe en fonction des profils d'énergie et ceci en construisant des profils dont les utilisateurs finaux peuvent tirer parti pour prendre des décisions éclairées sur la meilleure façon de protéger leurs plates-formes mobiles (à travers la transition automatique entre les différents modes de sécurités et d'énergies).

Bien que leur technique de mesure soit efficace, les auteurs ont implémenté des mesures de sécurité coûteuses (détection des rootkits, vérifications d'intégrité des données) qui sont rarement utilisées, même dans des systèmes autonomes dus à l'augmentation de la charge de travail.
Aussi, un écart de traitement a été prouvé après l'ajout du module de sécurité, qui conduit à un temps de traitement supplémentaire autour de ~ 30-50% pour les fonctions standard effectuées par le dispositif et il peut atteindre même plus de 100 %.

Dans (Yuan et al., 2013), un système de détection d'intrusion léger est proposé. Ce système utilise le classificateur Naive Bayes, pour détecter les malwares. Ce classificateur est formé à partir de plusieurs caractéristiques du téléphone intelligent, y compris le trafic de réseau, la consommation de la batterie, l'utilisation du processeur et le nombre de processus en cours et le nombre de SMS. Leur système s'exécute sur l'appareil mobile, donc, l'utilisation des ressources est prise en compte. Les auteurs ont testé leur approche sur 45 applications normales et 15 applications malveillantes de l'Android Market. Leurs résultats montrent un taux de précision allant de 76 % à 88 %. Cependant, le taux de faux positifs n'a pas été calculé afin de vérifier si leur système est efficace ou non. Une autre limitation, est que leur système n'est pas dynamique et qu'il crée un profil de comportement normal pour le système Android et non par application. Aussi, bien que les auteurs ont affirmé que leur système est « léger », ils n'ont pas évalué son impact sur la performance de l'appareil mobile.

Wang et al. (2013) ont introduit les concepts d'architecture orientée services dans leur approche de détection des malwares. Ils ont proposé « *SmartMal* », une architecture client-serveur. Les clients collectent et envoient des caractéristiques du système vers un serveur distant. Les caractéristiques sont extraites toutes les 30 secondes, au niveau du noyau Linux (CPU, RAM, etc.), au niveau applicatif (messages, etc.) et au niveau de l'utilisateur. Le serveur contient une collection des serveurs distribués qui exécutent des techniques de détection d'anomalies en parallèle. « *SmartMal* » utilise, donc, une fusion d'informations afin de concaténer les résultats des différents serveurs.

Pour la détection d'anomalies, les auteurs n'ont pas utilisé un algorithme d'apprentissage machine, mais ils ont utilisé un processus d'analyse hiérarchique des procédés (AHP) pour construire un modèle de 3 couches afin de diviser le problème complexe de décision en différents sous problèmes.

Pour évaluer la performance de *SmartMal*, les auteurs ont construit un prototype pour profiler le comportement de 32 applications normales qui s'exécutent sur trois téléphones intelligents et ils ont simulé et détecté une attaque Dos. Leurs résultats ont conduit à une utilisation maximale de la CPU autour de 20 % et 24 %, une consommation de la RAM de 26Mo, une consommation de l'énergie de 9,7 % à 10,6 % par heure et à une vitesse de chargement et de téléchargement, respectivement de 9Ko et 4 Ko par 10 min. Alors que leur approche souffre d'un taux de faux positif de 20 % en utilisant trois téléphones intelligents. Ce taux s'améliore en augmentant le nombre des appareils pour atteindre 99 % si on utilise plus de 10 téléphones intelligents. Aussi, l'analyse sur l'appareil a été faite d'une manière fondée sur des règles statiques.

Dans (Salman et al., 2014), les auteurs ont proposé « *DAIDS* », un système de détection d'intrusion dynamique pour Android, basée sur l'hôte. Le système surveille en temps réel le comportement des applications sur l'appareil et permet l'utilisation de l'analyse dynamique

pour détecter les comportements malveillants. Il collecte et surveille les informations sur les applications installées sur l'appareil, leurs dates d'installation et de mises à jour, leurs permissions, leur consommation de la mémoire, de la CPU et de la bande passante, l'activité de réseau, les SMS, etc. « *DAIDS* » consomme en moyenne 9,9 % de la CPU avec des pics allant jusqu'à 90 %, 23 Mo de mémoire et 163 mW d'énergie. Cependant, le système ne forme pas un modèle et la performance de détection n'est pas considérée afin de vérifier la crédibilité du modèle proposé.

2.6 Comparaison

Un bref résumé des solutions les plus récentes (2013-2015), présentées dans cette revue est décrit dans le tableau 2.2. Le résumé met en évidence les principales différences entre ces travaux et notre approche proposée, en termes des sept caractéristiques suivantes.

- Extraire des informations dynamiques (I.D) sous la forme de fonctionnalités pour générer des données d'entrée à des fins d'apprentissage et de test.
- Appliquer une analyse comportementale (A.C) afin de détecter les malwares polymorphes et les attaques par injection du code malicieux.
- Appliquer des algorithmes d'apprentissage machine (ML) pour former le modèle normal (et peut-être anormal) à être utilisé comme un point de référence pour identifier les actions anormales et bénignes.
- Déployer un processus de détection d'anomalies sur l'appareil (H). Le dispositif mobile devrait être en mesure d'identifier les actions anormales sans transfert de données vers un serveur externe pour la détection des logiciels malveillants.
- Concevoir un modèle dynamique (M.D) pour établir des seuils de prise de décision tout en effectuant le processus de détection d'anomalies /de l'utilisation abusive.
- Déployer une analyse des performances de détection (P.D) pour éviter de générer de faux positifs élevés et d'améliorer la performance en modifiant les seuils et les paramètres liés à la décision.

- Déployer une analyse des performances du dispositif (P.A) en termes de batterie, la mémoire et l'utilisation du processeur.

Tableau 2-2
Comparaison des caractéristiques des travaux connexes récentes

Approches	**I.D**	**A.C**	**ML**	**H**	**M.D**	**P.D**	**P.A**
(Sanz et al., 2013b)	F	V	V	V	V	V	F
(Min, Mingshen et Lui, 2014)	V	F	-	V	F	F	F
(Shabtai et al., 2014)	V	V	V	V	F	V	V
(Arp et al., 2014)	F	V	V	V	F	V	F
(Amos, Turner et White, 2013)	V	V	V	F	F	V	F
(Aung et Zaw, 2013)	F	V	V	V	F	V	F
(Sheen, Anitha et Natarajan, 2015)	F	V	V	V	V	V	F
(Sanz et al., 2013a)	F	V	V	-	F	V	F
(Kevin Joshua Abela, 2013)	V	V	V	V	F	V	F
(Hyo-Sik et Mi-Jung, 2013)	V	V	V	V	F	V	F
(Yuan et al., 2013)	V	V	V	V	F	F	V
(Wang et al., 2013)	V	V	F	F	F	V	V
(Salman et al., 2014)	V	V	F	Y	Y	F	V
Notre Approche	V	V	V	V	V	V	V

Notre travail est complémentaire à ces efforts, dans le sens où il traite tous les problèmes susmentionnés et emploie différents algorithmes d'apprentissage machine pour modéliser le comportement normal de chaque application. Ces algorithmes s'exécutent sur l'appareil lui-même pour identifier les malwares d'une façon autonome, indépendamment d'un serveur externe.

Le modèle normal est construit à partir des courtes séquences des appels système (n-grams). Nous avons choisi de surveiller les appels systèmes, car puisqu'ils sont fournis par le noyau et utilisés par les programmes en cours d'exécution dans l'espace utilisateur, tout programme malicieux doit générer des appels système pour causer des dommages. De plus, surveiller plusieurs paramètres à la fois, consomme plus de ressources systèmes.

Notre approche est, également, dynamique, permettant d'établir des seuils de prise de décision tout en effectuant le processus de détection d'anomalies. Cette décision est établie en testant les performances de chaque algorithme utilisé, en termes de taux de détection et de surconsommations de ressources.

La limitation des ressources est surveillée en utilisant un procédé similaire à celui utilisé par Salman et al. (2014).

2.7 Conclusion

Ce chapitre présente une étude des techniques tirées de la littérature qui sont appropriées pour la sécurisation des appareils mobiles. Il s'agit notamment de la détection d'anomalies sur la plateforme Android.

Dans ce chapitre, ces techniques sont classées en fonction de la méthode de détection, le type de déploiement, les types de ressources monitorées et le degré d'adaptation aux ressources limitées du téléphone intelligent.

Bien que ces approches aient démontré une importante efficacité pour protéger les téléphones intelligents, la majorité d'entre elles apprennent des techniques traditionnelles de détection

d'anomalies sur le PC et la limitation des ressources du téléphone intelligent n'est pas pleinement prise en compte.

Nous proposons, donc, un cadre de détection dynamique qui utilise des algorithmes de détection d'anomalies basés sur les n-grams. La détection est totalement indépendante d'un serveur externe et peut tourner efficacement sur l'appareil lui-même en respectant la restriction des ressources et en maintenant un important taux de détection. Les algorithmes utilisés sont présentés en détail dans la section suivante.

CHAPITRE 3

ARCHITECTURE PROPOSÉE

3.1 Introduction

Ce chapitre présente l'approche proposée. Il comporte une description du cadre de détection et de ses différents modules. Il comprend aussi une analyse des algorithmes de détection d'anomalie utilisés, tout en décrivant leurs processus d'apprentissage et de détection.

3.2 Présentation du cadre de détection

Dans ce travail, nous proposons un cadre modulaire de détection d'anomalies pour les appareils mobiles Android. Ce cadre est basé sur l'hôte et utilise l'analyse comportementale pour identifier les comportements malicieux des applications et alerter l'utilisateur.

Le système proposé comprend différents algorithmes de détection permettant de surveiller l'exécution d'une application sur un périphérique. Pour chaque application, le cadre construit un modèle normal basé sur l'algorithme de détection d'anomalie sélectionné. Ceci est basé sur l'hypothèse que la première exécution de l'application est normale et que nous avons assez de bonnes données de formation qui nous permettent de construire notre modèle normal.

Ensuite, chaque exécution future de l'application est surveillée pour construire un modèle de détection qui est ensuite comparée au modèle normal. Si le modèle de détection s'écarte significativement du modèle normal, le cadre soulève une alerte sur les traces anormales.

La construction du modèle normal peut être faite localement ou à distance, ce qui permet de tirer profit de la puissance de calcul et de stockage à distance, lorsque le serveur est

accessible. Cependant, le moteur de détection sur l'appareil est autonome et peut fonctionner parfaitement sans compter sur le serveur distant. C'est une importante propriété qui distingue notre approche du travail connexe. Dans les deux cas, les traces sont soumises à un serveur distant qui peut construire des modèles de détection d'anomalies plus complexes et plus lourds.

En outre, les traces et les modèles sont stockés dans des formats compressés en utilisant une technique de compression sans perte permettant d'augmenter l'espace de stockage et de réduire le coût de transfert des données sur le réseau. Les traces sont sérialisées avant d'être exportées vers le serveur distant.

Comme le montre la figure 3.1, le cadre proposé se compose de six éléments principaux: (1) la collecte de données, (2) le traitement des données; (3) la gestion des scans et des modèles, (4) le profilage, (5) le stockage et (6) la connexion à la base de données et au serveur.

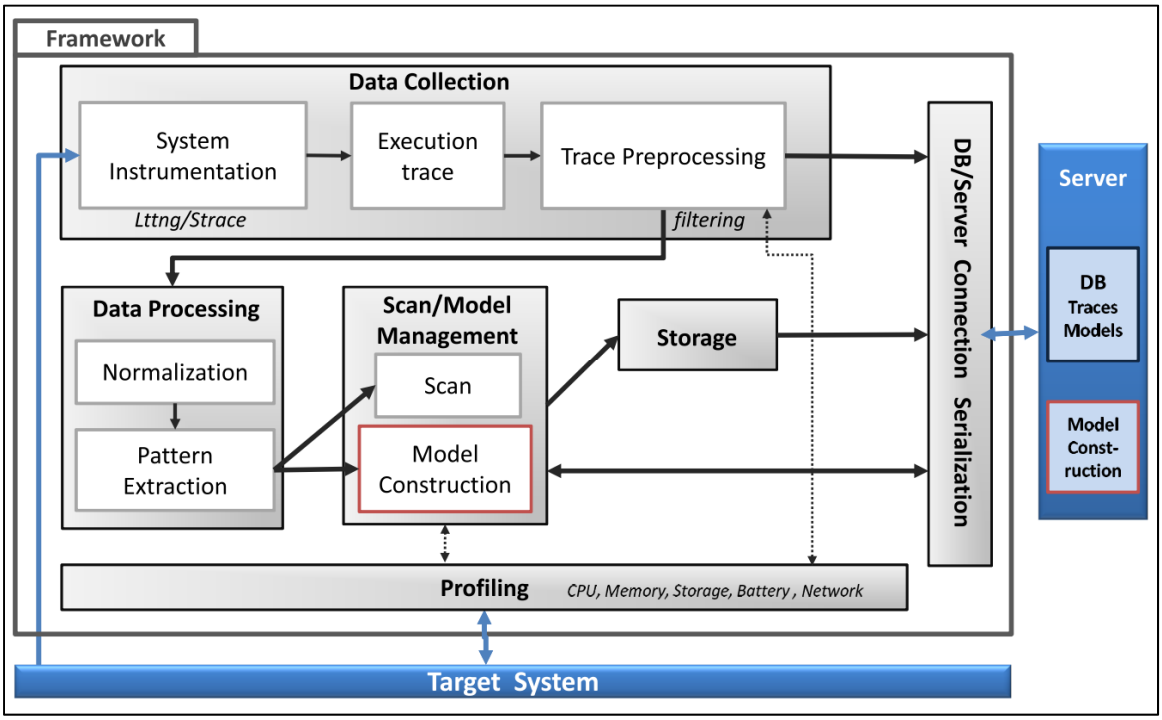


Figure 3.1 Architecture du cadre de détection proposé

3.2.1 Module de collecte de données

Pour construire un modèle représentatif, nous devons recueillir les données générées lorsque le système cible exécute les applications surveillées.

Dans ce travail, nous surveillons les appels système car ils sont fournis par le noyau et ils sont utilisés par les programmes en cours d'exécution dans l'espace utilisateur. En effet, toutes les requêtes des applications telles que la communication de réseau, la gestion des fichiers ou des opérations liées au processus doivent passer par le noyau en utilisant l'interface de l'appel système avant qu'ils ne soient exécutés. Ces données fournissent des informations précises sur le comportement d'une application.

Ceci est motivé par l'hypothèse qu'une fois compromise, les traces d'appels système d'une application seront différentes de ceux générés par sa version originale. Cette approche a été suivie par divers travaux antérieurs sur la détection d'anomalies dans les serveurs (Forrest, Hofmeyr et Somayaji, 2008).

La génération des traces d'exécution peut être faite à l'aide de différents outils d'instrumentation système tels que « *LTTng* » (*Linux Trace Toolkit Next Generation*) (Toupin, 2011) et « *strace* » (die.net). *LTTng*, est un outil de traçage avancé et complet permettant d'enregistrer, visualiser et analyser des données sensibles et critiques de différentes couches du système (noyau, métal nu, espace utilisateur). Cet outil est capable de tracer les appels système jusqu'à 172 fois plus rapide qu'avec l'outil *strace*, selon l'étude de Desnoyers et Dagenais (2006). Cependant, nous avons choisi de travailler avec l'outil *strace* suite à un problème de compatibilité de l'outil *LTTng* sur le système Android (au niveau du noyau) au moment du choix.

Strace est installé par défaut sur toutes les distributions Linux (sauf pour *Gentoo*). Une fois que les traces sont générées, elles seront prétraitées et converties en format texte pour être utilisées comme entrée pour le modèle choisi.
Ces traces peuvent être filtrées pour ne garder que les plus importants appels système afin de réduire la taille des traces et des modèles. En effet, dans la version 4.1.1 du système Android, il y a 444 appels système dans le noyau Linux 3.0.x, qui peuvent être invoquées par l'application (android.googlesource.com, 2012), alors qu'en réalité, une seule partie de ces appels sera invoquée par l'application. Par conséquent, le module de prétraitement se charge de filtrer les appels système, en choisissant ceux les plus susceptibles d'être significatifs, afin de pouvoir différencier les caractéristiques qui peuvent représenter le comportement malveillant et bénin.

Le choix du bon appel système est en lui-même une question de recherche à laquelle plusieurs études ont tenté de répondre comme (Dini et al., 2012; Isohara, Takemori et Kubota, 2011; Kevin Joshua Abela, 2013; Mas' ud et al., 2014).

Dans notre étude on se limite à concevoir un module de prétraitement, qui permet de filtrer les appels systèmes recueillis, sans spécifier réellement ceux qu'il faut garder et ceux qu'il faut supprimer.

3.2.2 Module de traitement de données

Les données recueillies seront envoyées, par la suite, à l'unité de traitement pour la normalisation. Afin de simplifier le processus d'analyse des données et de la création d'un modèle de comportement normal du système, nous traitons les données recueillies et nous découpons les traces générées en séquences de taille fixe. Cette taille influence sur la précision (comme nous le verrons dans le chapitre 4).

Pour chaque séquence de taille fixe, on extrait les caractéristiques qui sont utilisées par l'algorithme de détection d'anomalie. Les caractéristiques extraites sont de longueur fixe et variable, basées sur l'ordre temporel des appels système dans la séquence de trace, ainsi que sur la fréquence de chaque appel.
Ensuite, les traces générées sont converties au format approprié pour l'algorithme de détection d'anomalie choisi.

3.2.3 Module de modélisation et de numérisation

Ce module crée le modèle de comportement normal de chaque application. Le modèle normal est alors utilisé comme un point de référence, tout en testant une trace pour identifier les anomalies. À cette fin, plus le modèle représentatif est complet et les algorithmes de détection d'anomalies sont précis, plus on obtient de bons résultats en termes de taux de vrais positifs et de faux positifs (voir chapitre 4).

Nous avons choisi de définir le profil normal à l'aide d'un ensemble de courtes séquences d'appels système en utilisant des modèles de langage n-grammes. En effet, une vaste communauté de chercheurs a démontré l'efficacité de l'analyse des n-grammes d'appels système pour détecter l'utilisation malveillante des applications légitimes (Forrest, Hofmeyr et Somayaji, 2008; Hubballi, Biswas et Nandi, 2011; Jayasinghe, Shane Culpepper et Bertok, 2014; Rafiqul Islam, Saiful Islam et U. Chowdhury, 2011). Les algorithmes utilisés sont décrits en détails dans la section 3.3.

3.2.4 Module de stockage

Ce module est chargé de gérer les modèles construits pour les applications contrôlées. Rappelons que selon l'algorithme adopté, un autre modèle est construit pour une application spécifique. Par conséquent, des modèles différents peuvent être associés à l'application. Pour optimiser le budget mémoire nécessaire pour stocker les modèles normaux générés, nous avons adopté l'outil de compression open source « Zopfli » libéré par Google en 2013

(Alakuijala et Vandevenne). *Zopfli* utilise une technique de compression sans perte permettant de reconstruire parfaitement les données d'origine à partir des données compressées. Il compresse environ 3,7 % mieux que n'importe quel compresseur et jusqu'à plus de 65% des données.

Pour adapter le moteur de détection à la variation de la disponibilité de la mémoire, le composant de stockage peut (1) supprimer un modèle et le remplacer par un autre plus léger, (2) insérer et/ou supprimer un deuxième ou un troisième modèle de la même application, etc. De plus, les traces d'exécution sont stockées sur le dispositif et envoyées périodiquement au serveur distant. Ceci permet la flexibilité de la suppression et la contraction des modèles sur le périphérique puisque nous ne pouvons les réinsérer que lorsque les ressources sur l'appareil et le serveur distant sont de nouveau disponibles.

3.2.5 Module de profilage

Ce module est responsable de vérifier à chaque fois l'état du système afin d'adapter le processus d'analyse aux ressources disponibles sur l'appareil. Il collecte des informations sur l'état de la batterie (usage/niveau), l'espace de stockage (total et disponible), l'utilisation du processeur et de la mémoire vive RAM (*Random Access Memory*), la connectivité internet (disponibilité et bande passante) et les interfaces réseau (wifi, mobile).

Pour chaque paramètre monitoré, nous avons fixé un seuil pour être en mesure de prendre une décision à propos de la gestion de scan, des modèles et des traces.

3.2.6 Module de sérialisation et de connexion avec le serveur

Ce module permet de recueillir et de convertir les traces générées et stockées au format JSON (Crockford, 2006) afin de réduire la surcharge de l'appareil et simplifier leur exportation via le réseau à un serveur distant.

Les données sont enregistrées dans une base de données située dans le serveur, pour être utilisées pour créer des profils de comportement normaux plus précis avec les mêmes algorithmes ou autres algorithmes plus puissants, de numériser et de visualiser les données de trace. Une fois les modèles comportementaux distants créés, ils peuvent remplacer si nécessaire les modèles locaux dans le dispositif.

3.3 Algorithmes de détection d'anomalies

Comme mentionnés précédemment, nous utilisons quatre algorithmes pour la détection d'anomalie, basés sur les n-grams. Nous avons choisi ces modèles pour leur faible coût. En fait, il s'agit de la modélisation la plus simple, que nous ayons trouvée, au niveau de l'analyse et au niveau de l'espace de stockage nécessaire.

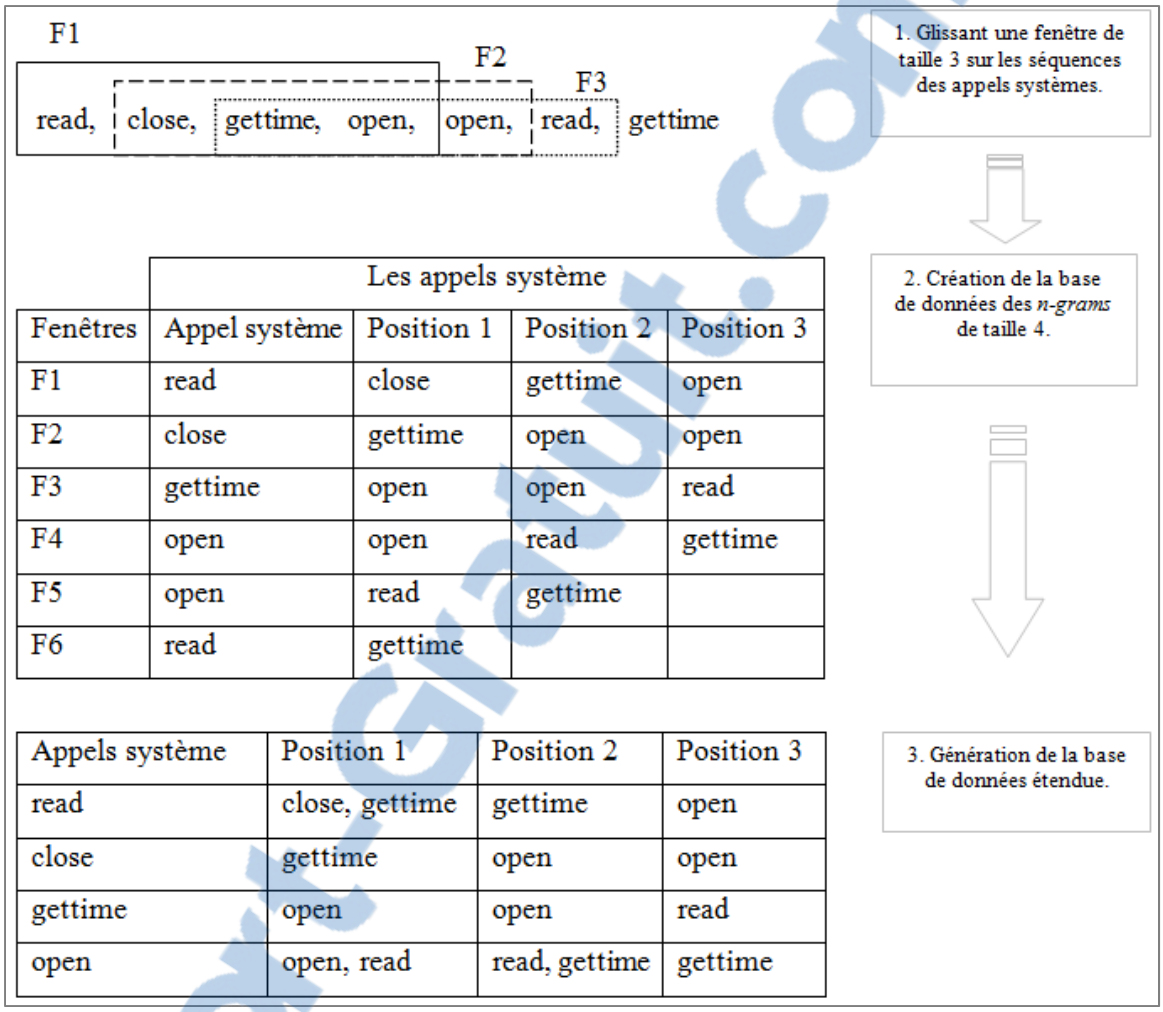
Dans cette partie, nous introduisons les algorithmes de détection d'anomalie utilisés, tout en décrivant et analysant leurs processus d'apprentissage et de détection.

3.3.1 Lookahead

La première approche, « *Lookahead* », est basée sur le travail de Forrest et al. (1996). Leur approche a établi une analogie entre le système immunitaire humain et le système de détection d'intrusion. Ils ont fait cela en proposant une méthodologie qui consiste à analyser les séquences d'appel système d'un processeur privilégié pour construire un profil normal. Dans leur article, ils ont analysé plusieurs programmes basés sur UNIX comme « sendmail », « LPR », etc. Puis ils ont montré que la corrélation dans les séquences de longueur fixe d'appels système pourrait être utilisée pour construire un profil normal d'un processeur. Ensuite, dans une phase de détection, les programmes qui montrent des séquences qui s'écartent du profil normal vont être considérées comme des logiciels malveillants.

Notre approche reprend ce travail pour l'appliquer sur les applications Android. Nous avons recueilli une base de données d'un comportement normal pour chaque application. Cette base de données est créée en faisant glisser une fenêtre de taille fixe à travers les séquences normales des appels systèmes. Puis, on enregistre le premier appel système dans la fenêtre, suivi de tous les appels qui s'apparaissent dans les positions qui le suivent jusqu'à la dernière position dans la fenêtre, en gardant le même ordre d'apparition. Chaque sous-séquence enregistrée correspond à un n-gram de taille k+1 avec k est la taille de la fenêtre glissante utilisée.
Par exemple, avec une fenêtre de taille k = 3, l'algorithme enregistre pour chaque appel, tous ses successeurs dans la même fenêtre, à la position 1, position 2 et position 3, comme le montre la figure 3.2. On obtient donc une base de données des n-grams de taille 4.

Puisqu'un appel système peut apparaitre plus qu'une fois dans la trace, il peut être donc suivi de différents appels système. Pour cela, on crée une base de données élargie, en regroupant pour chaque appel système unique, tous ses successeurs qui s'apparaissent dans la base de données des n-grams déjà construite. La base de données résultante constitue notre modèle de comportement normal qui va être utilisé, ensuite, pour surveiller le comportement de l'application.



	Les appels système			
Fenêtres	Appel système	Position 1	Position 2	Position 3
F1	read	close	gettime	open
F2	close	gettime	open	open
F3	gettime	open	open	read
F4	open	open	read	gettime
F5	open	read	gettime	
F6	read	gettime		

Appels système	Position 1	Position 2	Position 3
read	close, gettime	gettime	open
close	gettime	open	open
gettime	open	open	read
open	open, read	read, gettime	gettime

Figure 3.2 Modèle Lookahead

Puisque le modèle normal est construit en parcourant la trace une seule fois, le coût de sa mise en place est proportionnel à la taille de la trace. Par la suite, la complexité en temps de cet algorithme est linéaire et elle est seulement de l'ordre de $O(N)$ où N est le nombre des appels système dans la trace analysée. Donc avec une trace de 1000 appels système, le temps de création du modèle normale sera de 10 µs par trace (il est rapide)(Carton, 2008).

En plus de son faible coût, la taille du modèle au pire cas, est de l'ordre de $(N.k)$ par trace, où k est la largeur de la fenêtre. Par ailleurs, le tableau des paires regroupées (la base de données étendue) permet de réduire la taille du modèle en ne gardant que les appels uniques

et leurs successeurs regroupés par position. Dans l'exemple de la figure 3.2, la taille du modèle est réduite de 21 à 15 appels système avec la méthode de regroupement des pairs (la taille de la trace =7).

Dans la phase de détection, on glisse une fenêtre de la même façon que pour la construction et on compare les n-grams extraites avec celles trouvées dans le modèle normal. Un n-gram qui ne se trouve pas dans le modèle normal est considéré comme anormal. Une trace est considérée comme anormale lorsque son taux de paires anormales dépasse un certain seuil.

La structure des paires regroupées permet d'adapter l'analyse aux ressources limitées et ceci en se limitant à traiter des n-grams de longueurs inférieures à celles utilisées lors de la création du modèle. Par exemple, un modèle est construit avec une fenêtre de taille 3 génère des n-grams de taille 4. Le tableau de regroupement des pairs correspondant, définit pour chaque appel, les appels qui le suivent à la position 1, la position 2 et la position 3. De ce fait, lors du scan, on peut se limiter à vérifier pour chaque appel, seulement la présence des appels qui le suivent à la position 1 et 2 dans le tableau de regroupement des pairs déjà définit. En outre, la méthode de regroupement permet aussi de réduire le nombre de faux positifs en donnant un modèle plus général.

Cependant, cette structure peut générer des faux négatifs. En fait, en conservant en mémoire seulement les appels regroupés et non les sous-séquences complètes des n-grams, la modélisation du comportement normal sera plus générale que celle en réalité. Ceci est du au fait qu'on peut générer de la base des paires regroupées, des sous séquences des appels systèmes qui n'apparaissent pas dans les traces normales utilisées pour construire cette base.

Par exemple, au niveau du tableau de regroupement des pairs dans la figure 3.2, la séquence *open, open, read, read* est vue par le modèle comme une séquence normale, alors qu'elle n'apparaît pas dans la séquence initiale.

3.3.2 Arbre de n-grams

Cette approche est basée sur le travail de Hubballi, Biswas et Nandi (2011). Les auteurs ont décrit une nouvelle technique de détection d'anomalie sur le système Linux. Cette technique est basée sur la modélisation des courtes séquences d'appels système (n-grams) avec leurs fréquences d'occurrence dans les traces d'apprentissage.

Leur approche consiste à extraire des n-grams de différentes longueurs en faisant glisser une fenêtre de taille variable (de 1 à k) à travers les séquences normales des appels systèmes. Les n-grams de même taille sont, par la suite, enregistrées sous forme de groupes, où chaque groupe g_i contient tous les n-grams de taille i ainsi que leurs fréquences d'apparition dans les traces normales. Le profil de comportement normal est construit en utilisant une structure de stockage d'arbre désignée comme l'arbre des n-grams, où chaque niveau i de l'arbre contient tous les n-grams qui appartiennent au groupe g_i .

Ensuite, les auteurs ont appliqué le classificateur « k-means » pour classifier les n-grams selon leurs fréquences. Chaque classe t contient les n-grams dont les fréquences appartiennent à la même plage fréquentielle de t. L'approche attribue à chaque classe t, un score dit « score d'anomalie » $s(t)$ qui représente la probabilité que ses n-grams soient anormaux. Les auteurs ont supposé que les n-grams les moins fréquents ont plus de chances d'être anormaux, pour cela, ils vont avoir des scores d'anomalies supérieures à ceux attribués aux séquences les plus fréquentes. Sous cette hypothèse, le score de chaque classe est défini comme le logarithme du rapport de la somme de toutes les fréquences des n-grams (Ng) de taille k dans toutes les traces, sur la somme des fréquences des n-grams (Ngt) qui appartiennent à la classe t, suivant l'équation 3.1.

$$s(t) = \log\big(f(t)\big)\, avec\, f(t) = \frac{\sum_{i=1}^{i=Ng} frequence_i}{\sum_{i=1}^{i=Ngt} frequence_i^t} \tag{3.1}$$

La détection est faite en générant tous les n-grams d'ordre k. Si un n-gram p_j d'ordre k existe dans l'arbre, on lui attribue un score d'anomalie égale au score d'anomalie de sa classe $t(p_j)$, sinon en lui attribue un score s suivant la formule 3.2, où N_c est le nombre des classes dans le modèle normal.

$$s = \log(2.\max f(t_c)) \; avec \; 1 < c < \; N_c \tag{3.2}$$

Après avoir parcouru toute la trace de test, on calcule un score d'anomalie pour toute la trace, définit comme le rapport de la somme de tous les scores des n-grams de la trace de test sur le nombre total des n-grams de test. La trace est considérée anormale si son score dépasse un certain seuil.

Les auteurs ont évalué leur approche avec $k = 3$, en utilisant 8 programmes basés sur UNIX, dont 7 possèdent des traces normales et anormales. Les résultats de leurs expérimentations ont révélé que des séquences courtes (même d'ordre 3) modélisées avec leur fréquences d'occurrence, sont des bons discriminateurs entre les traces normales et anormales avec un taux de détection allant jusqu'à 100% et un taux maximal de faux positif de l'ordre de 1,70%.

La complexité en temps de la phase de test est linéaire et elle est de l'ordre de $O(A)$ par trace, où A est le nombre des alphabets (les appels système unique). Cependant, la complexité en temps et en espace de la création du modèle est polynomiale, elle est de l'ordre de $O(A^k)$ par trace. Dans notre travail, nous avons appliqué cette approche sur le système Android, en se limitant à la création de l'arbre de n-grams sans utiliser la classification fréquentielle des n-grams, ce qui réduit la complexité en temps de la phase d'apprentissage à $O(A^2)$, cette complexité quadratique est plus faible que celle générée par l'approche de Hubballi, Biswas et Nandi (2011).

Dans le présent travail, nous avons collecté un ensemble de courtes séquences d'appels système, en glissant une fenêtre de taille fixe sur les traces. Ces n-grams ont été enregistrés sous la forme d'un arbre similaire à l'arbre de l'approche (Hubballi, Biswas et Nandi, 2011). Chaque nœud i au niveau i de cet arbre, contient l'i^{ème} appel système a_i du n-gram p_i de taille i, construit en parcourant l'arbre de la racine jusqu'au nœud i. De plus, chaque nœud i, est un 2-uplet de $< a_i, f_i >$, où f_i est la fréquence d'occurrence du n-gram p_i dans la trace d'apprentissage. Pour une fenêtre de taille k, cette structure permet de stocker non seulement les n-grams de taille k, mais aussi les n-grams de tailles inférieures, comme le montre la figure 3.2. Cette structure permet d'adapter l'analyse aux ressources limitées et ceci en se limitant à traiter les n-grams de longueurs inférieures à celles utilisées lors de la création du modèle, tout comme le modèle *Lookahead*.

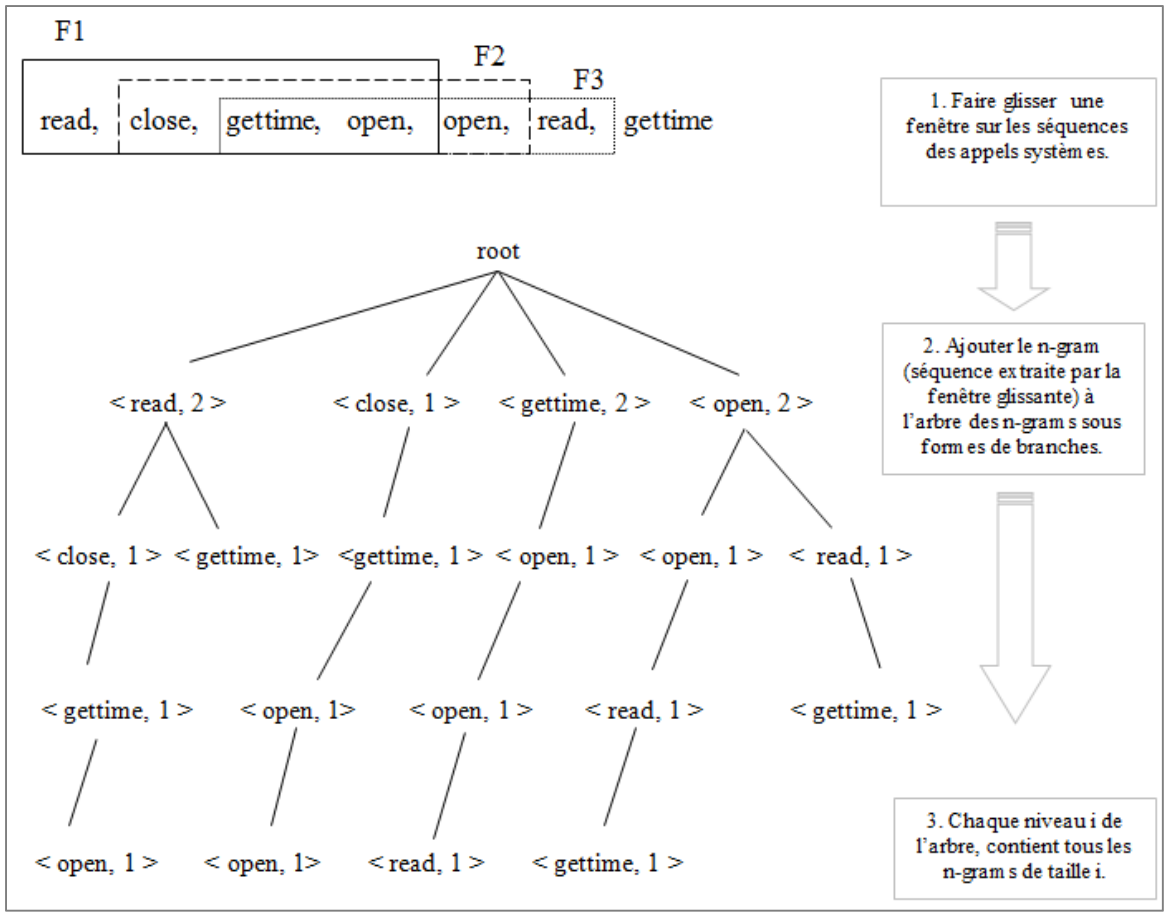


Figure 3.3 Modèle d'arbre n Gram

Chaque n-gram p_i , admet un score d'anomalie s_i qui représente la probabilité qu'il soit anormal. Ce score est calculé en fonction de sa fréquence d'occurrence f_i dans la trace et le nombre total des n-grams dans le modèle normal N_g suivant la même logique de Hubballi, Biswas et Nandi (2011) et sans utiliser l'approche de classification, selon la formule 3.3.

$$s_i = \log(N_g / f_i) \tag{3.3}$$

La phase de détection est semblable à celle définie dans le travail de Hubballi, Biswas et Nandi (2011), mais au lieu d'utiliser les scores d'anomalies des classes de n-grams, on utilise les scores d'anomalies de chaque n-grams. Si un n-gram existe dans l'arbre du modèle

normal, on lui attribue le même score d'anomalie déjà calculé dans la phase d'apprentissage, sinon en lui accorde un score selon l'équation 3.2, mais en utilisant les fréquences des n-grams au lieu des scores des classes des n-grams. Puisque tout n-gram de l'arbre d'apprentissage possède une fréquence supérieure ou égale à 1, donc le score d'anomalie d'un n-gram qui n'existe pas dans l'arbre d'apprentissage est défini par la formule 3.4.

$$s = \log(2.N_g) \tag{3.4}$$

Une trace est considérée comme anormale, si la moyenne de tous les scores d'anomalies des n-grams qui la constituent dépasse un certain seuil (voir section 4.1.2.2).

3.3.3 Machine à états finis de bi-grams

Cette approche consiste à surveiller les courtes séquences d'appels systèmes de longueur 2. Elle est basée sur le travail de Michael et Ghosh (2002).

Dans la phase d'apprentissage, on construit une table de transition TT de taille m x m où m est le nombre des appels systèmes uniques (taille de l'alphabet). Chaque transition $TT[i][j]$ est la fréquence d'occurrence de 2-gram composés par la concaténation des appels système s_i et s_j de la ligne i et la colonne j dans la table de transition, par rapport au nombre total des appels systèmes dans toutes les traces de tests. La transition $TT[i][j]$ est définie selon la formule 3.5, où N est la longueur des traces et N_T le nombre de traces.

$$TT[i][j] = f(s_i, s_j)/\ N.N_T \tag{3.5}$$

Donc chaque entrée de la table de transition reflète le taux d'apparition consécutive de deux appels systèmes dans la trace d'apprentissage globale. De cette façon, cette table enregistre les probabilités de transition d'un appel système à un autre, que l'appel s_i soit suivi par

l'appel s_j. Par conséquent, la somme des probabilités sur une ligne i correspond à la probabilité d'apparition de l'appel s_i dans toutes les traces d'apprentissage.

Le comportement normal de chaque application surveillée à l'aide de cet algorithme, est caractérisé à travers la table de transition TT qui contient les probabilités de chaque appel système et de chaque combinaison de deux appels système.

Dans la phase de détection, si une trace de test contient un appel système qui n'existe pas dans la table TT générée au cours de la phase d'apprentissage, alors cet appel est appelé un appel étranger et il va être identifié comme un appel malicieux. Pour cela, les traces de formation devraient être suffisamment génériques pour créer un ensemble complet des appels systèmes uniques $\sum$, sinon, de fausses détections peuvent découler des appels système non enregistrés.

La détection est faite en calculant l'écart global des traces de test à partir de la table du modèle normal. Il existe deux types d'écarts qui sont enregistrés dans ce modèle:

- Un symbole étranger où un appel système s_i dans les traces de tests, n'appartient pas à l'ensemble $\sum$: $s_i \notin \sum$
- Une combinaison anormale de deux appels système s_i et s_j qui appartiennent à l'ensemble $\sum$ et qu'ils ne possèdent pas une transition normale (dans le modèle normal) entre eux : $TT[i][j] = 0$

Dans ce modèle, on calcule le score de déviation pour chaque type d'écarts et on calcule la probabilité qu'une trace soit anormale en calculant les scores d'écart générés à partir d'un nombre d'écarts tout au long de la trace de test.

La méthode de machine à états finis présente l'avantage de créer et de mettre à jour la table de transition facilement pendant l'apprentissage. De plus, la complexité de la phase de

détection est faible et est linéairement dépendante de la taille des traces. En outre, la précision de la détection dépend fortement des traces de formation et du domaine d'applications. De plus amples détails sont fournis dans le chapitre 4, où nous appliquons le test à des études de cas empiriques.

Cependant, la machine à états finis ne permet pas de surveiller des n-grams de taille supérieure à 2. En plus, sa complexité en espace est forte et est quadratique en fonction du nombre des alphabets dans les traces d'apprentissage.

3.3.4 N-grams de longueurs variées (VLN)

Contrairement aux modèles précédents qui caractérisent les n-grams de tailles fixes, ce modèle permet de détecter des séquences significatives de n Gram de longueurs différentes.

Cet algorithme est inspiré des travaux de Guofei et al. (2007). Dans leur approche, les auteurs construisent des séquences de différentes longueurs et un automate à état fini pour caractériser le comportement normal du système. Une nouvelle trace est comparée à l'automate déjà formé dans une phase d'apprentissage, pour savoir si elle est anormale afin de détecter et diagnostiquer les défaillances dans les systèmes distribués à grande échelle.

Cette approche est basée sur l'hypothèse que le fait d'ignorer les n-grams les moins fréquents n'affecte pas la validité du modèle. Par conséquent, elle consiste à surveiller seulement les n-grams les plus fréquents. Elle utilise un seuil α (de 0 à 1) pour contrôler la capacité de généralisation du modèle en filtrant les n-grams les moins fréquents.

Dans le travail de Guofei et al. (2007), le modèle normal est établi suivant deux étapes. La première étape consiste à construire des ensembles C_k des n-grams valides de longueur k, en commençant par $k = 1$. Chaque n-gram c_{k+1}^{i} de longueur $(k + 1)$ est construit en fusionnant deux n-grams c_k^n et c_k^m, qui appartiennent à l'ensemble C_k et dont le suffixe de

l'un est égal au préfixe de l'autre. Le n-gram résultant, c_{k+1}^{i}, est dit n-gram valide si sa fréquence d'occurrence dans la trace est supérieure au produit du seuil α avec le minimum des fréquences des n-grams qui le constituent (équation 3.6).

$$f\left(c_{k+1}^{i}\right) > \alpha.\min(f(c_k^n), f(c_k^m)) \;\; avec \; 0 < \alpha < \; 1 \qquad (3.6)$$

La complexité de cette étape est linéairement dépendante de la longueur des traces, mais elle est exponentielle en fonction du nombre de n-grams. Cependant, dans la pratique, cet algorithme pourrait converger rapidement, surtout avec un seuil α élevé.

La deuxième étape consiste à construire un automate à états finis. Cet automate est construit, en parcourant les traces normales et en remplaçant chaque n-gram unique, généré dans la première étape, par un état, en commençant par celui le plus long et le plus fréquent. Cependant, avec des petites valeurs de α, l'automate ne garde qu'un petit nombre de n-grams (que les n-grams les plus longs), surtout si α tends vers 0, on ne garde que N_T n-grams de longueurs N, où N_T est le nombre de traces, c'est-à-dire chaque trace va représenter un n-gram. Pour éviter cet inconvénient, les auteurs ont créé une autre automate qui généralise l'automate précédent, mais avec une complexité de calcul exponentielle en fonction du nombre des n-grams et qui sera difficile de l'exécuter sur un appareil de ressources limitées. Pour réduire le coût de calcul et la taille du modèle, dans notre travail, nous reprenons seulement l'algorithme d'extraction de patterns de l'approche de Guofei et al. (2007) et l'appliquons pour détecter les malwares dans la plateforme Android.

Notre approche forme un modèle normal conformément à l'algorithme d'extraction des patterns pour générer un ensemble de courtes séquences d'appels système, en respectant la formule 3.6. La figure 3.4 illustre un exemple de création du modèle normal.

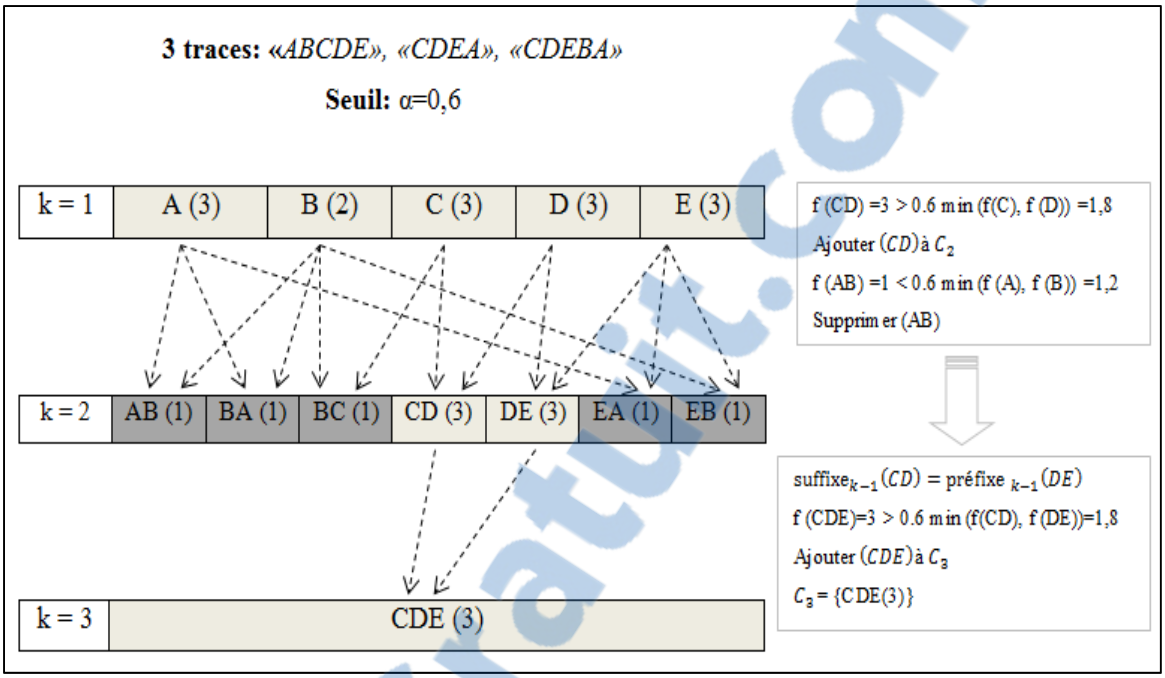


Figure 3.4 Modèle de N-grams de longueur varié

Au cours de la phase de détection, un modèle est construit de manière similaire, en générant différents ensembles de n Gram de longueurs différentes. Ce modèle est comparé au modèle normal en vérifiant certaines conditions :

- Si un n-gram c_k^i ne se trouve pas dans le modèle de comportement normal et dans son ensemble c_k, il est considéré anormal.
- Si le nombre des n-grams anormaux dans un même ensemble c_k dépasse un certain seuil s_k, on considère c_k comme anormal.
- S'il existe un ensemble c_k anormal alors la trace est considérée comme malicieuse.

Puisque chaque ensemble c_k présente les plus fréquents n-grammes de taille k, il est donc rare de trouver un n-gram étranger dans un ensemble c_j avec $j \gg k$ .s'il existe alors il est fort possible que l'ensemble c_j soit anormal. Cependant, nous pouvons trouver un n-gram étranger de plus petites tailles. Ainsi, lorsque la taille des n-grams augmente, le seuil diminue. En nous basant sur cette hypothèse, nous avons proposé un seuil s_k dynamique pour chaque ensemble c_k des n-grams de longueur k, défini selon la formule 3.7.

$$s_k = (k \times seuil)\ avec\ 0 < seuil < 1 \qquad (3.7)$$

Ce modèle présente l'avantage de pouvoir facilement surveiller de courtes séquences d'appels système de différentes longueurs (n-grams, $n > 2$). Il peut également réduire la taille d'un ensemble de données tout en maintenant une efficacité de détection. De plus, si les limitations de ressources ne sont pas respectées, la génération et la mise à jour du modèle normal peuvent être effectués sur l'appareil.

Cependant, la mise à jour de sa base d'apprentissage est difficile à déployer. De plus, si le seuil α augmente, le modèle ne sera pas suffisamment général pour représenter le comportement normal, ce qui augmente la probabilité de fausses détections. Alors, pour diminuer le taux de fausses alertes, nous avons conçu une nouvelle version de VLN. La nouvelle version permet de créer un sous-modèle normal pour chaque trace d'appels système au lieu d'un modèle normal pour la totalité des traces. Le modèle normal global résultant est créé en fusionnant tous les sous-modèles (chaque ensemble c_k du modèle résultant contient tous les k-grams de tous les sous-modèles).

Nous fournissons des détails supplémentaires concernant l'utilisation des ressources et la performance dans le chapitre 4, où nous appliquons le test à des études de cas empiriques.

3.4 Décision adaptative

3.4.1 Gestion des traces

La compression et/ou l'envoi des traces vers le serveur dépendent de la disponibilité de la connexion réseau et de l'espace de stockage libre dans l'appareil.

Nous avons surveillé la disponibilité des interfaces réseaux et nous avons choisi celle qui a la plus large bande passante (B_{max}). Puis, nous avons comparé cette bande passante avec la bande passante minimale requise (α_B). Nous avons comparé aussi le pourcentage de l'espace de stockage libre (S_{free}) avec la valeur minimale requise (α_S). Le tableau 3.1 résume les différentes décisions que nous avons prises pour chaque état.

Tableau 3-1 Gestion des traces

Interface réseau active	**Espace mémoire libre**	**Décisions**
$\boldsymbol{B_{max} > \alpha_B}$	----------	- Envoyer les traces en cours et les traces enregistrées (compressées) vers le serveur. - Mise à jour du modèle comportemental
$\boldsymbol{B_{max} < \alpha_B}$	$\boldsymbol{S_{free} > \alpha_S}$	- Enregistrer les traces dans l'appareil - Compresser les traces lorsqu'ils atteignent un certain nombre (la compression est lente, mais elle permet d'obtenir plus d'espace et de diminuer les frais de transfert de données et l'utilisation de la batterie)
$\boldsymbol{B_{max} < \alpha_B}$	$\boldsymbol{S_{free} < \alpha_S}$	- Augmenter le seuil α du modèle « Varied-length N-grams » afin de diminuer la taille du modèle à enregistrer. - Diminuer la taille des n-grams (taille de la fenêtre) pour les modèles lookahead et arbre.

De plus, pour gérer l'accès concurrentiel à la file d'attente des traces prétraitées, nous avons utilisé deux threads. Le premier thread est utilisé pour traiter les traces prétraitées, pour créer

des modèles et pour analyser les applications, alors que le second est utilisé pour envoyer les traces directement vers le serveur.

3.4.2 Gestion des modèles et de numérisation

La création des modèles et/ou la numérisation avec un ou plusieurs algorithmes dépendent de la puissance du processeur, la capacité de la RAM et de l'autonomie restante de la batterie. Nous avons surveillé ces paramètres et nous les avons comparés avec leurs seuils.
Le tableau 3.2 résume les différentes décisions que nous avons prises pour chaque état.

Tableau 3-2 Gestion des modèles et de numérisation

Batterie	**RAM**	**CPU**	**Décisions**
$B > \alpha_{Batt}$	$R < \alpha_{RAM}$	$C < \alpha_{CPU}$	- Scanner - Utiliser plus qu'un seul modèle. - ***Maximiser la précision*** ✓ Augmenter la taille de n-grams (taille de la fenêtre) pour le modèle arbre ✓ diminuer le seuil α du modèle « VLN ».
$B > \alpha_{Batt}$	$R > \alpha_{RAM}$	$C < \alpha_{CPU}$	- Scanner - Utiliser un seul modèle. - ***Minimiser la quantité des données en cours de traitement*** ✓ Diminuer la taille de n-grams (taille de la fenêtre) pour les modèles lookahead et arbre. ✓ Augmenter le seuil α du modèle « VLN».
$B > \alpha_{Batt}$	----------	$C > \alpha_{CPU}$	- Scanner - Utiliser un seul modèle. - ***Minimiser la quantité des données en cours de***

			traitement ✓ Diminuer la taille de n-grams (taille de la fenêtre) pour les modèles lookahead et arbre. ✓ Augmenter le seuil α du modèle « VLN».
Batterie	**RAM**	**CPU**	**Décisions**
			- ***Minimiser le nombre de traitements*** ✓ Ne pas envoyer les traces vers le serveur ✓ Ne pas compresser les traces
$B < \alpha_{Batt}$	----------	----------	- Scanner seulement

CHAPITRE 4

TEST ET EXPÉRIMENTATIONS

4.1 Méthodologie

L'objectif des expériences menées est de fournir des informations détaillées concernant la performance des algorithmes de détection d'anomalies déployés sur les appareils mobiles à ressources limitées. Nous avons mesuré l'utilisation des ressources de ces algorithmes et nous avons analysé leurs taux de détection et de précision. L'évaluation des performances a été effectuée sur des ensembles de données réelles d'appels système. Ces données ont été extraites à partir des applications bénignes qui s'exécutent sur un appareil mobile et en utilisant le protocole expérimental décrit dans la suite.

Les expériences ont été faites sur un Samsung Galaxy S3 exécutant le système d'exploitation Android 4.1.1, avec 2G de RAM, un processeur dual-core cadencé à 1,5 GHz et une mémoire interne de 12G. La surveillance dans un environnement réel garantit que les applications surveillées se comportent comme s'elles se comporteraient sur n'importe quel téléphone et elles ne sont pas susceptibles de détecter un environnement de type sandbox où elles peuvent cacher leur comportement malveillant. L'expérience a occupé 3.41MB d'espace mémoire.

4.1.1 Data Set

Pour réaliser les expériences précitées, nous avons utilisé un ensemble de données des appels système qui sont utilisés pour l'apprentissage du modèle de comportement normal. Ces traces d'appels système sont collectées pendant le fonctionnement normal des applications bénignes sur l'appareil mobile. Par conséquent, ces traces sont supposées être bénignes et pourraient être utilisées pour représenter l'attitude normale des applications.

Les traces de test sont des séquences d'appels système collectées à partir des processus infectés qui contiennent des comportements malicieux. Par conséquent, pendant les tests, les traces comprennent à la fois le comportement normal et anormal. Nous décrivons ci-dessous les trois ensembles de données des appels système du monde réel utilisés dans nos expériences. Les expériences sont réalisées sur des données générées à partir de versions malicieuses correspondant à trois applications décrites dans le tableau 4.1.

Tableau 4-1
Description des différentes versions des applications de test

Apps	Versions normales (VN)	Versions malicieuses	
		Réel (Intelligence, 2013)	Code malicieux injecté dans la VN
Angry birds	1.1.0	1.1.2 (Purandarear, 2011) - Charger du code supplémentaire. - Localiser l'appareil - Voler les contacts - Envoyer des messages texte.	Prend des captures d'écran et les stocke dans la carte SD.
Candy Star	1.0.3	1.0.2 (diamant2320, 2014) - Charger une bibliothèque partagée et un fichier DEX - Lire / modifier / supprimer le contenu de la carte SD.	Supprime toutes les photos prises par l'appareil
Ninja Chicken	1.4.8	1.4.5 (diamant2320, 2014) - Charger une bibliothèque partagée et un fichier DEX - Lire / modifier / supprimer le contenu de la carte SD. Lire l'état du téléphone - Identifier les applications en cours	Envoi des SMS comportant l'ID de l'appareil.

		d'exécution.	

Puisque les versions malicieuses des applications contiennent le comportement normal et malicieux, il est difficile d'identifier exactement les actions malicieuses de celles bégnines lors de la phase de test. Pour augmenter la précision nous avons injecté du code malicieux dans un endroit bien déterminé de chaque application bégnine. La modification d'une application Android peut se faire soit au niveau du code source Java (en général la recompilation génère des erreurs), ou au niveau du code *Smali*.

Nous avons créé trois applications malicieuses en injectant du code malicieux au niveau du code smali des trois applications bégnines du Tableau 4-1. Le déclenchement du code malicieux se fait à travers une autre application externe qui communique avec l'application de test en utilisant la méthode RPC (*Remote Procedure Call*) (Srinivasan, 1995) de la technologie IPC(developer.android, 2015).

Le tableau 4-2 donne plus de détails sur chaque ensemble de données abordé dans cette section. Il comporte le nombre de processus et d'appels système uniques (Alphabets) présentés dans les traces globales, les traces d'apprentissages et les traces de tests de chaque application.

Tableau 4-2
Les caractéristiques des data Set (P: Processus, A: Alphabet)

	Traces				Apprentissage		Test					
	Normales		Anormales				Normales		Anormales			
									Réels		Modifiés	
	P	A	P	A	P	A	P	A	P	A	P	A
Angry-Birds	49	57	37	58	29	44	45	65	35	47	28	42
Ninja-Chicken	67	52	41	53	38	44	42	51	41	53	61	63
Candy-Star	1240	45	784	43	170	37	939	39	784	43	147	51

4.1.1.1 Angry Birds

Chaque trace de l'application *AngryBirds*, est générée lors de l'exécution de cette application sur un appareil mobile. Les appels système de chaque trace peuvent appartenir à différents processus. Nous avons extrait les traces d'appels système générées par chaque processus, ces traces contiennent différents alphabets comme le montre le tableau 4-2. Après le prétraitement des données, nous avons obtenu des traces d'apprentissage et de test.

L'ensemble des données de l'application AngryBirds comprend plusieurs traces normales avec une longueur totale de 899 020 appels système. Ces traces contiennent 49 processus où les plus courts processus contiennent 1 appels système (PID = 2315, 2264, 2240) alors que le plus long contient 37 appels système différents (PID = 2482).

Les traces de test contiennent 45 processus normaux, 35 processus anormaux de la version malicieuse réelle et 28 de la version modifiée. Les processus normaux les plus courts ont les PID 2315 et 2264 alors que le plus long processus a le PID 2482. Il y a un certain nombre de processus anormaux avec un seul appel système (PID = 14438, 14416 et autres). De plus, le plus long processus dans la version réelle contient 21 appels système (PID = 14370) et dans la version modifiée contient 28 appels (PID=15900).

4.1.1.2 NinjaChicken

Similaire à l'application *AngryBird*, chaque trace de l'application *NinjaChicken*, est générée lors de l'exécution de cette application sur un appareil mobile. Chaque trace admet un certain nombre d'appels système générés par différents processus (voir le tableau 4-2).

Les traces normales comportent 1923906 appels système. Ces traces contiennent 67 processus dont les plus courts contiennent 1 appel système (PID = 2855, 2845, 2872) et celui le plus long contient 32 appels système (PID = 2841).

Les traces de test comportent 42 processus normaux, 41 processus anormaux extraits de la version malicieuse réelle et 61 de la version modifiée. Les processus anormaux les plus courts sont de PID = 2872, 2855 et celui le plus long est PID = 2481. Il y a un certain nombre de processus anormaux avec un seul appel système (PID = 15303, 15309 et quelques autres). Aussi, le plus long processus contient 30 appels système (PID = 15310) dans la version réelle et 36 appels systèmes pour la version modifiée (PID = 18255).

4.1.1.3 Candy Star

L'application CandyStar est semblable aux deux autres applications citées précédemment, chaque trace de l'application CandyStar est également générée lors de l'exécution de cette application sur un appareil mobile. Dans chaque trace, il y a un certain nombre d'appels système générés par différents processus (voir le tableau 4-2). L'ensemble de données de l'application CandyStar comprend plusieurs traces normales avec une longueur totale de 707 818 appels système. Ces traces contiennent 1,240 processus dont les plus courts contiennent 2 appels système (PID = 8345, 8342, 8343) et le plus long contient 36 appels système (PID = 8336).

Les traces de test contiennent 939 processus normaux, 784 processus anormaux issus de la version malicieuse réelle et 147 de la version modifiée. Les processus normaux les plus courts sont de PID = 8338, 8345 et celui le plus long est de PID = 8364. Il y a un certain nombre de processus anormaux avec un seul appel système (PID = 11 150, 11 191 et quelques autres). Il y a aussi un long processus avec 28 appels système (PID = 11 143) dans la version réelle et 35 appels système (ID=2076) dans la version modifiée.

4.1.2 Le protocole expérimental

Nous avons appliqué les techniques de « Lookahead », de l'arbre de n-grams, de la Machine à états finis et de N-grams de longueurs variées pour créer les profils de comportement

normal de chaque application. Le tableau 4-2 fournit de plus amples détails sur la façon selon laquelle nous avons classé les données normales et de test.

4.1.2.1 Lookahead

Dans cette approche et pour une fenêtre de taille k, les traces d'apprentissages sont utilisées comme entrée pour extraire les informations suivantes:

- ✓ Toutes les séquences des appels systèmes de taille $k+1$, observées pour créer l'ensemble des (k+1)-grams.
- ✓ Tous les appels systèmes observés pour créer l'alphabet des appels système (ensembleΣ).
- ✓ La longueur totale des traces d'entrée.

La base de données étendue est créée en fonction des alphabets et contient la liste de tous les successeurs jusqu'à la position $k+1$ de chaque appel système de l'ensemble Σ. Cette base est utilisée par la suite comme une référence qui caractérise le modèle normal.

Dans la phase de détection, on fait glisser une fenêtre de la même taille sur les traces de tests, afin d'extraire les n-grams et les comparer avec ceux enregistrés dans la base d'apprentissage afin d'identifier les écarts de comportement par rapport au modèle normal. Selon le type de l'écart et le nombre d'écarts des n-grams de la trace de test, la trace peut être classée comme normale ou anormale.

On calcule deux types d'écarts pour chaque n-gram selon l'équation 4.1: (E1) un appel système du n-gram de test ($Ngram_i\ (j)$) qui n'appartient pas à l'alphabet d'apprentissage Σ; et (E2) un appel système du n-gram de test suivi par un appel autre que celui identifié dans la base de données d'apprentissage étendue B associée à l'alphabet qui débute ce n-gram et à la même position.

$$E1(Ngram_i) = \forall j \in [1,n], Ngram_i(j) \in \sum \quad (4.1)$$
$$E2(Ngram_i) = \forall j \in [2,n], Ngram_i(j) \in B_j^{Ngram_i(1)}$$

Ces écarts sont calculés pour chaque n-gram dans les traces de test, pour pouvoir les classer comme normaux ou anormaux selon la formule 4.2. Un n-gram est étiqueté comme anormal si sa polarité est nulle.

$$Polarité(Ngram_i) = E1(Ngram_i).E2(Ngram_i) \quad (4.2)$$

Une trace est considérée comme anormale lorsque son taux des n-grams anormales (dont la polarité est nulle) par rapport au nombre total des n-grams dans la trace de test, dépasse un certain seuil.

Durant les expérimentations, nous testons ce modèle avec différents tailles de la fenêtre (différents n-grams) et différentes valeurs du seuil de détection.

4.1.2.2 Arbre de n Gram

Comme la méthode de Lookahead, cette approche extrait un ensemble de courtes séquences d'appels systèmes en utilisant une fenêtre de taille fixe. Cependant, au lieu de les enregistrer par position, cette méthode les enregistre sous forme d'arbre où chaque nœud présente 2-uplet; le n-gram et sa fréquence d'apparition et chaque niveau i de l'arbre présente tous les n-grams d'ordre i.

Pour une fenêtre de taille k, on extrait les informations suivantes depuis les traces d'apprentissage:

- ✓ Toutes les séquences des appels systèmes de taille k, observées pour créer l'ensemble des k-grams.
- ✓ La fréquence de chaque séquence d'appels système.

- ✓ Tous les appels systèmes observés pour créer l'alphabet des appels système (ensembleΣ).
- ✓ La longueur totale des traces d'entrée.

Pour chaque n-gram, nous avons associé un score d'anomalie calculé en fonction de sa fréquence d'occurrence dans la trace et le nombre total des n-grams dans le modèle normal selon l'équation 3.3.

Dans la phase de détection, on extrait de la même façon des n-grams de taille k. Si un n-gram existe dans le modèle normal, on lui attribue le même score d'anomalie déjà calculé dans la phase d'apprentissage, sinon en lui attribue un score selon la formule 3.4 (voir 3.3.2).

Une trace est considérée malicieuse, si la moyenne de tous les scores d'anomalies des n-grams qui la constituent dépasse un certain seuil.

Durant les expérimentations, nous testons ce modèle avec différentes tailles de la fenêtre (différents n-grams) et différentes valeurs du seuil de détection.

4.1.2.3 Machine à états finis de bi-grams

Dans cette approche, lors de la phase d'apprentissage, on extrait les informations suivantes:

- ✓ Tous les appels systèmes observés pour créer l'alphabet des appels système (ensemble $\sum$).
- ✓ La fréquence de chaque séquence d'appels système qui commence par un appel système qui appartient à l'alphabet.
- ✓ La longueur totale des traces d'entrée.

La table de transition est créé avec une taille de $|\sum| \times |\sum|$ contenant la probabilité de transition entre chaque paire d'appels système $(S_1, S) \in \sum \times \sum$. Cette probabilité est calculée

en divisant la fréquence d'apparition de chaque pair dans toutes les traces d'apprentissage, par la longueur totale des traces accumulées. Par conséquent, la somme de toutes les probabilités dans la table de transition est égale à 1.

Ci-après, nous allons nous référer à la table de transition calculée à partir de l'entrée d'apprentissage qui caractérise le « modèle de comportement normal». Ce modèle peut être utilisé pour identifier les écarts par rapport au comportement normal dans les traces de test. Selon le type de l'écart et le nombre d'écarts de la trace, la trace peut être étiquetée comme normale ou anormale.

Ils existent deux types d'écarts: (1) un symbole étranger, qui désigne un appel système de la trace de test qui n'appartient pas à l'alphabet d'apprentissage $\sum$; et (2) une paire étrangère, qui désigne une paire d'appels système dont la probabilité de transition dans le modèle de comportement normal est nulle (une nouvelle combinaison des pairs ou un ordre d'apparition non défini dans la table de transition). Pour chaque type, une probabilité est calculée selon l'équation 4.3.

$$P(s_1, s_2) = \frac{(1 + STD_N(s_1))(Z_N(s_1) + 3)}{(Br_N(s_1))(Z_T(s_2) + 3)(Z_N(s_2) + 3)} \quad (4.3)$$

$P(s_1, s_2)$ est la probabilité d'une activité anormale initiée à partir de l'appel système s_1 suivi par l'appel système s_2. Cette probabilité tient en compte les paramètres suivantes :

- l'écart-type $STD_N(s_1)$ de l'apparition de l'appel système s_1 dans l'ensemble de données d'apprentissage.
- la variable centrée réduite $Z_N(s_1)$ de l'appel système s_1 dans l'ensemble de données d'apprentissage.
- Le nombre des paires différentes $Br_N(s_1)$ dans l'ensemble de données d'apprentissage initiée à partir de l'appel système s_1.

- La variable centrée réduite $Z_T(s_2)$ de l'appel système s_2 dans l'ensemble de données de test.
- La variable centrée réduite $Z_N(s_2)$ de l'appel système s_2 dans l'ensemble de données d'apprentissage.

Toute trace de test qui présente des déviations (pour les deux types), est étiquetée comme une activité anormale si la valeur maximale (voir équation 4.4) des écarts identifiés dépasse un certain seuil. Dans cette équation, $d(t)$ désigne la distance d'une trace t du modèle de comportement normal. Cette distance est calculée dans l'intervalle [0, 1] pour chaque trace de test et à partir du modèle de comportement normal où 0 représente la trace la plus proche du modèle normal et 1 représente celle la plus éloignée (la plus différente du comportement bénin). Le même processus est suivi pour les traces d'apprentissage et de test afin de pouvoir les classer en deux classes; normal et anomalie.

$$d(t) = max_{\forall(s_1,s_2)\in t} P(s_1, s_2) \tag{4.4}$$

Durant les expérimentations, nous testons ce modèle avec différentes valeurs du seuil de détection.

4.1.2.4 N-grams de longueurs variées

La quatrième approche de détection d'anomalies est basée sur les n-grams de longueurs variables. Dans la phase d'apprentissage, on extrait les n-grams les plus fréquents depuis les traces normaux et selon la formule 3.6. L'idée est de réduire la taille des données d'apprentissage en ne gardant que les courtes séquences d'appels système les plus fréquentes.

Similaire aux méthodes précédentes, lors de la phase d'apprentissage, on extrait les informations suivantes:

- ✓ Tous les appels systèmes observés pour créer l'alphabet des appels système (ensemble Σ).
- ✓ La fréquence de chaque séquence d'appels système qui commence par un appel système qui appartient à l'alphabet.
- ✓ La longueur totale des traces d'entrée.

Le profil de comportement normal est créé en fonction des n-grams de longueurs différentes et selon le procédé présenté dans la section 3.3.4. En d'autres termes, toute trace normale peut être construite en utilisant ces n-grams.

Une nouvelle trace est comparée au modèle appris pour déterminer si elle est anormale (ou normal). Cette étape consiste à créer un modèle de test de la même façon qu'en phase d'apprentissage. Une fois créé, ce modèle sera comparé avec celui normal en parcourant tous les groupes de n Gram de chaque modèle.

Un ensemble de n-grams est considéré comme anormal si le taux des n-grams étrangers qui le constituent dépasse un certain seuil dynamique défini selon la formule 3.7. Une trace est considérée comme anormale si elle possède un ensemble anormal.

Durant les expérimentations, nous testons ce modèle avec différents valeurs d'alpha (seuil du modèle) et différentes valeurs du seuil dynamique (seuil de détection par ensemble).

4.1.3 Paramètres d'évaluation

Pour évaluer la performance des différentes approches dans la détection des traces anormales, nous avons utilisé le taux de vrai positifs (TP), qui représente la proportion de traces anormales correctement détectés sur le nombre total d'attaques dans l'ensemble de test.

Nous avons aussi mesuré le taux de faux positifs (FP). Ce taux représente la proportion de traces bénignes incorrectement classées comme anormales par rapport au nombre total de traces normales dans l'ensemble de test.

Pour une représentation plus claire et plus efficace du taux de TP en fonction de FP, nous avons utilisé la fonction d'efficacité du récepteur (Hanley et McNeil, 1982), connue sous le nom de courbe ROC (*Receiver Operating Characteristic*). Cette courbe fournit une estimation sur la valeur optimale du seuil de détection permettant de réaliser un compromis entre le TP et le FP. Une courbe ROC est, donc, un tracé de taux de vrais positifs contre le taux de faux positifs pour différents seuils de détection.

On a aussi mesuré l'aire sous la courbe ROC (AUC: *Area Under Curve*). La valeur AUC est généralement utilisée pour comparer les performances de détecteurs indépendamment des seuils de décision. Une AUC=1 indique un détecteur parfait qui détecte toutes les anomalies sans fausses alarmes (TPR = 1, FPR = 0), tandis qu'un détecteur aléatoire aura une AUC=0,5. Plus la valeur AUC est grande, plus la courbe s'éloigne de la ligne du classificateur aléatoire (droite linéaire) et se rapproche de l'angle du classificateur idéal.

C'est bien d'avoir un TP de 100% et un FP de 0%, mais ceux-ci seuls ne nous permettent pas de bien estimer la performance du détecteur. Pour cela, nous avons aussi mesuré le nombre des attaques non détectées, en calculant le taux d'exactitude (ACC) selon la formule (4.5), où TN désigne le taux de vrai négatifs (la proportion de traces normales correctement classées comme normales sur le nombre total de traces normales dans l'ensemble de test) et FN désigne le taux de faux négatifs (la proportion des attaques incorrectement classées comme normales par rapport au nombre total de traces anormales dans l'ensemble de test) .

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.5}$$

Pour les tests de surconsommations, nous avons évalué le pourcentage d'utilisation de la RAM, de la CPU et de la Batterie. Nous avons mesuré également, la taille des traces et des différents modèles en kilo-octets (Ko).

4.2 Tests de surconsommations

Cette section comprend les tests de surconsommations faites avec les quatre algorithmes, plusieurs configurations et selon différents scénarios.

4.2.1 Scénarios de tests

Nous avons réalisé nos expérimentations de surconsommations selon différents scénarios dans lesquels nous avons mesuré la surcharge du processeur, de la RAM et de la mémoire de stockage. Ces mesures ont été faites avec différents algorithmes et différentes configurations (la taille de n-grams, le seuil alpha de génération, le nombre de traces).

Dans le premier scénario, nous avons mesuré les ressources utilisées en termes de CPU et RAM lors de la création d'un modèle de comportement normal de chaque application de test. Le modèle normal est créé sur la base des traces d'exécution normale générées par l'outil d'instrumentation « strace ». Chaque modèle est généré à partir d'environ 480,000 appels système.

Dans le deuxième scénario, nous avons mesuré les ressources utilisées en termes de CPU et RAM lors du scan en parallèle de trois applications avec chaque algorithme. Ce test a été fait avec 100,000 appels système recueillis à partir des versions malicieuses réelles de chaque application.

Le troisième scenario consiste à mesurer l'espace mémoire de stockage utilisée lors de la création d'un modèle normal composé de 7000 appels système avec différentes configurations de chaque algorithme.

Dans le quatrième scénario, nous avons mesuré la taille des modèles et des traces avant et après la compression avec l'outil « Zopfli ». Dans ce test, les modèles sont formés avec 500,000 appels système et une configuration bien déterminée. Ce scénario permet d'évaluer la puissance de l'outil de compression utilisé, en comparant la taille des modèles et des traces aussi avant et après la compression.

Ces mesures ont été effectuées grâce au module de profilage. Ce module récupère, toutes les 30 secondes, l'utilisation des ressources en se basant sur la lecture des fichiers du noyau Linux (le fichier « /proc/[PID]/stat » pour l'utilisation de la CPU et de la RAM par un processus de PID=[PID] et le fichier « /proc/uptime » pour la durée d'exécution du système).

Dans ce qui suit, on note que le pourcentage de CPU mesuré correspond à la valeur utilisée par un seul cœur du processeur et non la valeur de CPU totale de l'appareil. Dans notre cas, les expérimentations sont faites sur un Samsung Galaxy S3 avec un processeur dual-core, donc le pourcentage total de CPU est de 200%.

4.2.2 Surcharge engendrée par la création des modèles

Les Figures 4.1 (4.2), 4.3 et 4.4 (4.5) présentent, respectivement, le pourcentage de RAM (CPU) utilisé lors de la création des modèles Lookahead et arbre de n-grams, du modèle de machines à états finis et du modèle de n-grams de longueurs variées (VLN). L'axe des abscisses des courbes de Lookahead et d'arbre de n-grams, correspond à la profondeur d'analyse (taille des n-grams). Pour le modèle VLN, l'axe des abscisses présente la valeur du seuil de génération des n-grams (alpha).

Les résultats montrent que la consommation de la RAM et de CPU dépend fortement de la longueur de n-gram et de la valeur du seuil alpha, sauf pour le modèle FSM qui ne dépend d'aucun paramètre.

En fait, pour les modèles Lookahead et arbre de n-grams, chaque n-gram comprend un nombre d'appels système récupérés à travers une fenêtre de taille fixe. Alors plus la taille de la fenêtre est large, plus le nombre d'appels système à investiguer est grand, entraînant ainsi une forte consommation de RAM et de CPU.

La même logique s'applique au modèle VLN. En effet, si on diminue la valeur d'alpha, le modèle générera plus de classes de n-grams, ce qui augmente le nombre des données à investiguer et provoquant ainsi une forte consommation de RAM et de CPU.

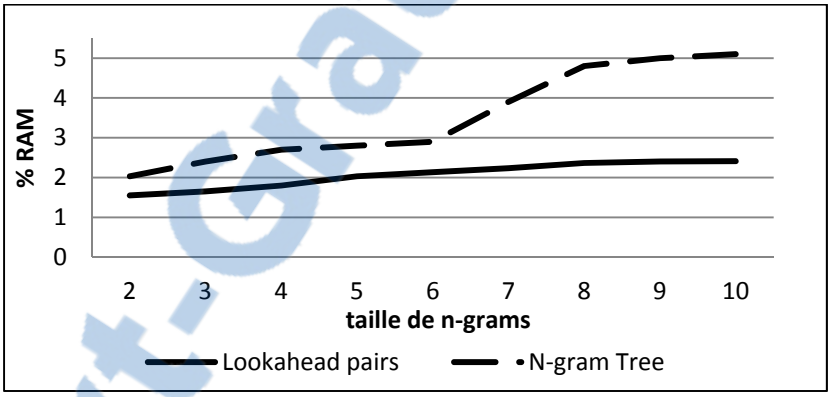


Figure 4.1 Consommation RAM de création des modèles Lookahead et Arbre de n-grams

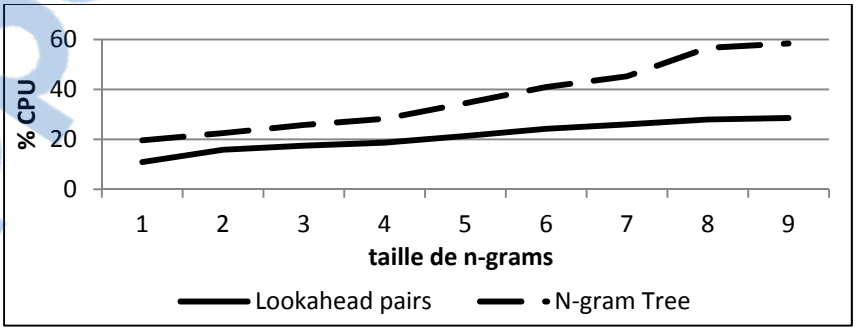


Figure 4.2 Consommation CPU de création des modèles Lookahead et Arbre de n-grams

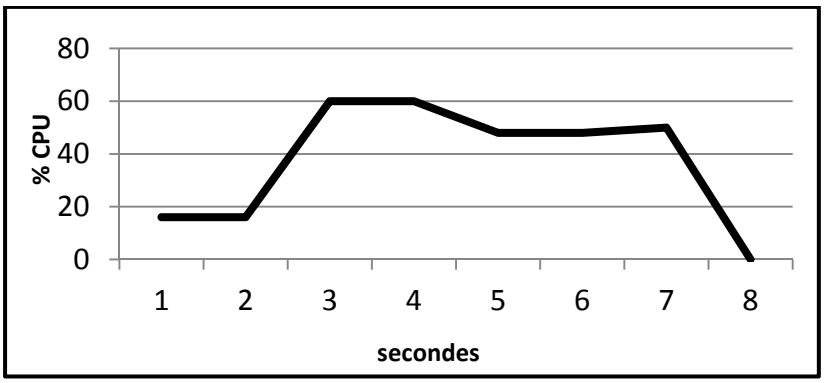


Figure 4.3 Consommation CPU de création du modèle FSM

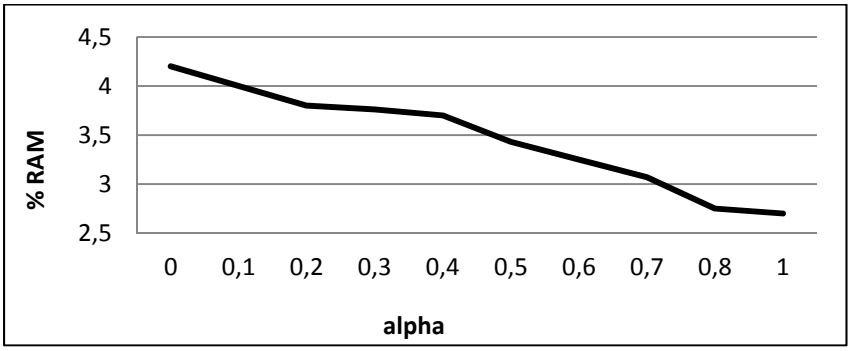


Figure 4.4 Consommation RAM de la création du modèle VLN

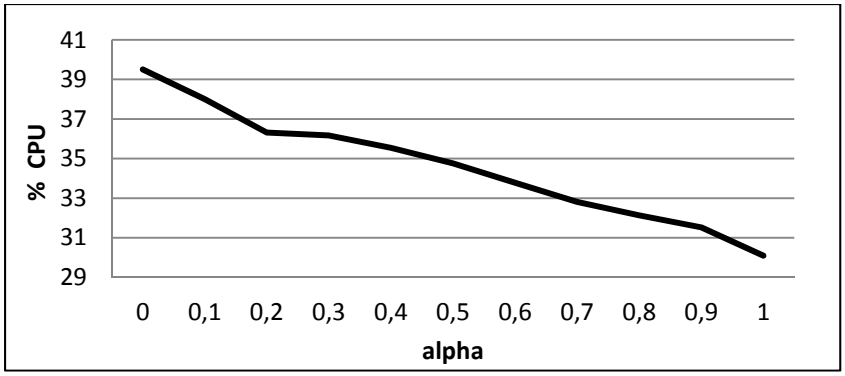


Figure 4.5 Consommation CPU de création de modèle VLN

Les figures 4.1 et 4.2 montrent que la consommation de la RAM et de CPU avec l'algorithme VLN augmente en diminuant la valeur du seuil alpha. Cet algorithme consomme dans le pire cas (avec alpha égale à 0) environ 4,5% de RAM et de 40% de CPU (ce qui correspond à

20% de la capacité maximale de système). Cependant, ces valeurs peuvent diminuer tout en augmentant alpha, pour atteindre 2,7% de RAM et 30 % de CPU.

On constate aussi que pour les algorithmes Lookahead et arbre de n-grams, cette consommation augmente en augmentant la profondeur d'analyse et donc la longueur de n-gram. Nous remarquons ainsi que le modèle de type arbre consommera jusqu'à 5% de la RAM et 58% de CPU en utilisant 10-grams, alors qu'avec seulement 2-gram, il consommera juste 2% de RAM et 19,6% de CPU.

La création du modèle Lookahead semble être la plus légère avec une consommation de RAM et de CPU qui ne dépasse pas, dans le pire cas, les valeurs de 2,5 % et 29%, respectivement. Par contre, la création du modèle FSM fournit la plus haute consommation de mémoire vive et de CPU (6% de RAM et 60% de CPU).

4.2.3 Surcharge de la numérisation en parallèle

Dans cette section nous avons mesuré la surcharge de l'analyse avec chaque algorithme de détection d'anomalies utilisé.

Les Figures 4.6 (4.7), 4.8 (4.9), 4.10 (4.11) et 4.12 (4.13) présentent, respectivement, le pourcentage de RAM (CPU) utilisé lors de l'analyse avec le modèle Lookahead, arbre de n-grams, machines à états finis et de n-grams de longueurs variées (VLN). L'axe des abscisses des figures correspondent au nombre d'applications scannées en parallèle.

Les résultats montrent que le modèle Lookahead (figures 4.6 et 4.7) est le plus performant en termes de consommation de RAM et de CPU, avec des valeurs qui ne dépassent pas 5 % et 32 % respectivement, dans le pire cas (3 applications avec 10-grams).

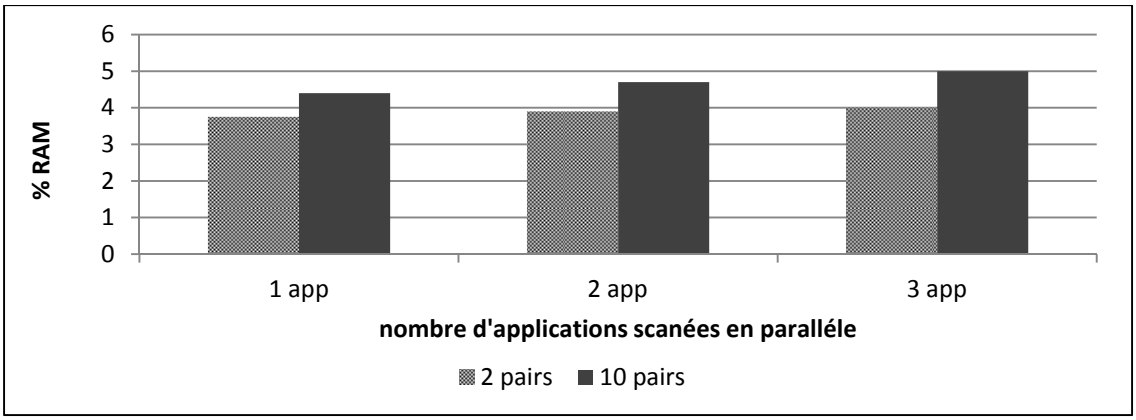


Figure 4.6 Consommation RAM de la numérisation avec le modèle Lookahead

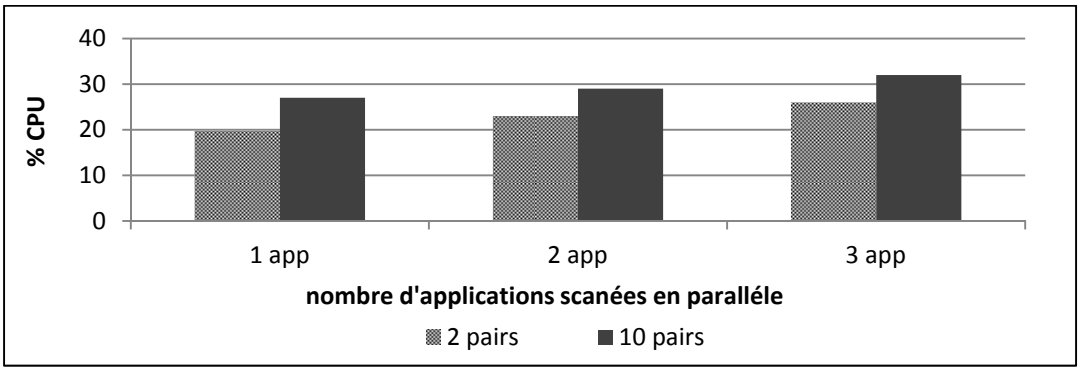


Figure 4.7 Consommation CPU de la numérisation avec le modèle Lookahead

Nous notons aussi que dans le pire des cas, avec 10-grams, nous sommes capables de scanner trois applications en même temps en utilisant le modèle d'arbre (figures 4.8 et 4.9) tout en restant au-dessous de 6 % de mémoire vive et de 58 % du processeur.

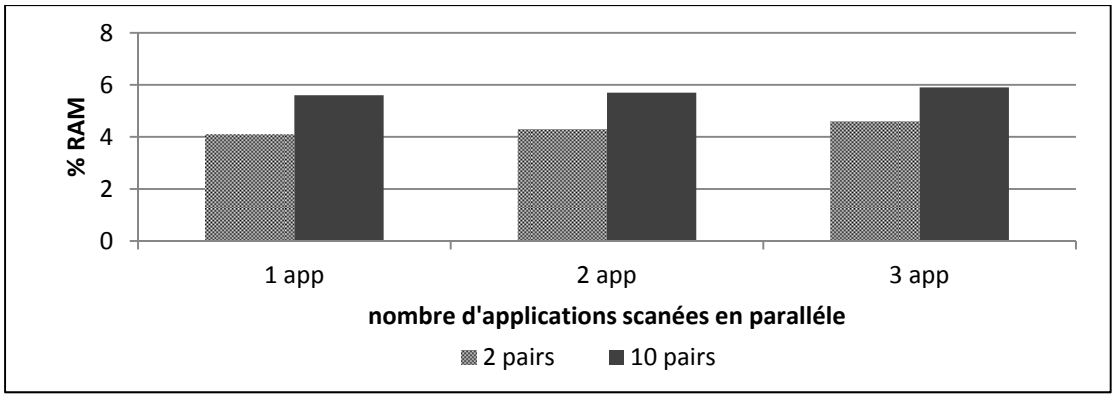


Figure 4.8 Consommation RAM de la numérisation avec le modèle Arbre n-grams

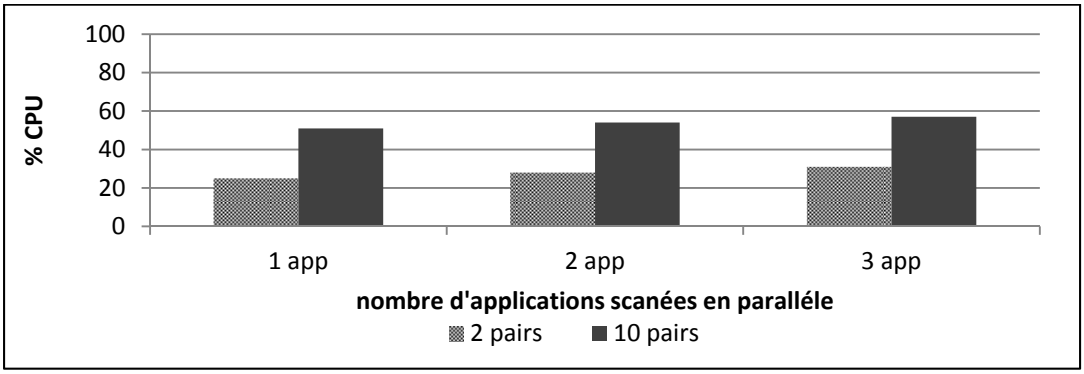


Figure 4.9 Consommation CPU de la numérisation avec le modèle Arbre n-grams

À l'opposé, la numérisation avec le modèle FSM (figures 4.10 et 4.11) fournit la plus grande consommation de RAM et de CPU qui atteint, respectivement, 9 % et 69 %. Cependant, on peut se limiter à scanner 2 applications en parallèle.

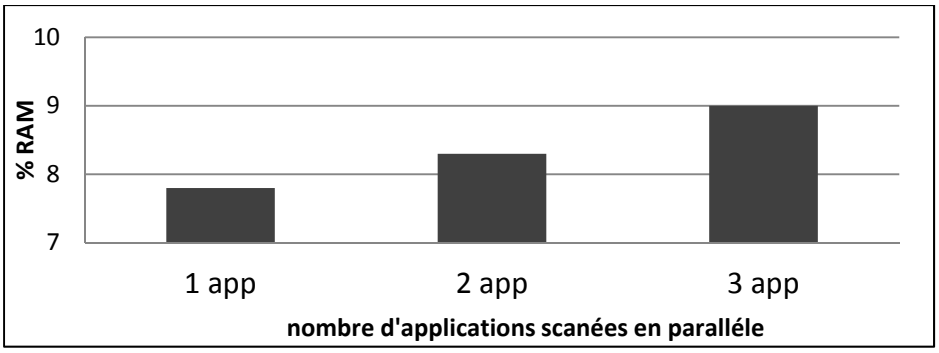


Figure 4.10 Consommation RAM de la numérisation
avec le modèle FSM

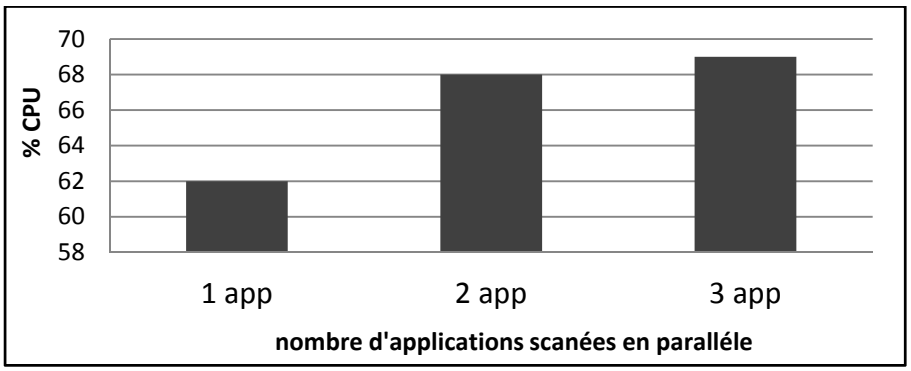


Figure 4.11 Consommation CPU de la numérisation
avec le modèle FSM

Nous remarquons aussi qu'avec l'algorithme VLN (figures 4.12 et 4.13), nous pouvons effectuer jusqu'à trois exécutions parallèles avec un seuil de 0,1 (dans le pire des cas). Cependant, ceci génère une forte consommation de RAM et de CPU (de 8% et 64% respectivement).

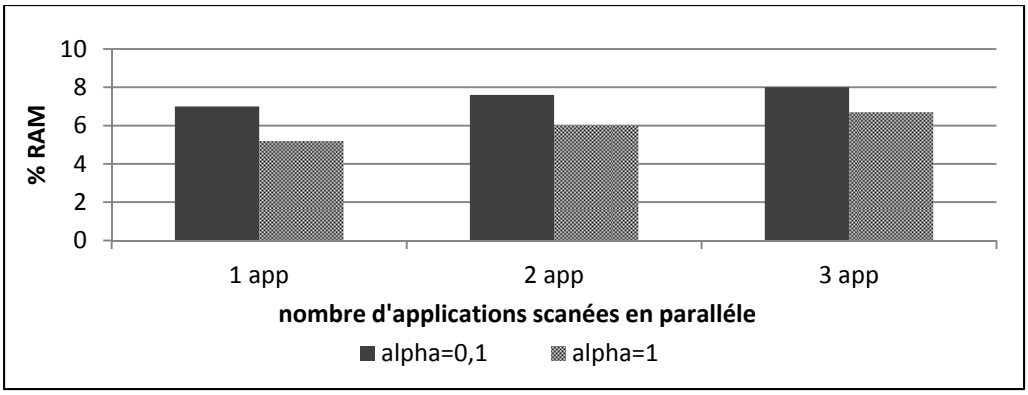


Figure 4.12 Consommation RAM de la numérisation avec le modèle VLN

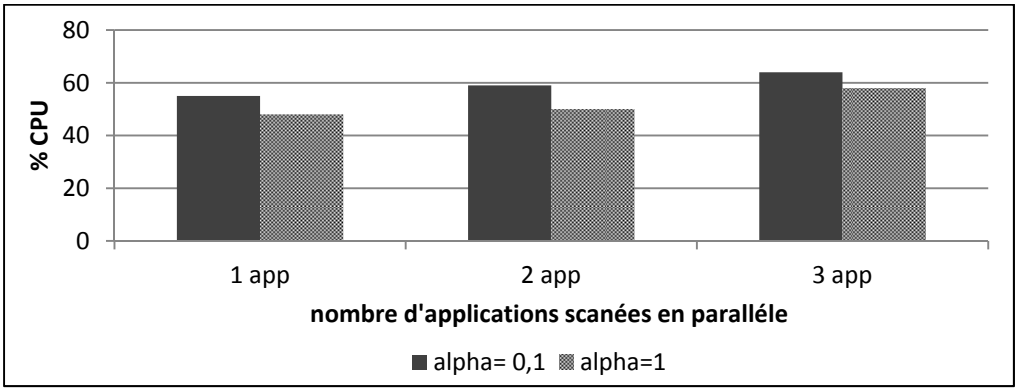


Figure 4.13 Consommation CPU de la numérisation avec le modèle VLN

Ces résultats montrent que notre cadre est capable de lancer plus d'un algorithme de détection sans affecter les ressources système.

4.2.4 Tailles des modèles

Dans cette partie, nous mesurons l'espace mémoire occupée par chaque algorithme de détection d'anomalie utilisé. Il est important d'évaluer la taille des modèles et ceci afin de savoir à quel point pouvons-nous stocker plus qu'un seul modèle sur l'appareil.

Les Figures 4.14 et 4.15 présentent les tailles en kilobits (kb) du modèle de n-grams de longueurs variées (VLN) et de Lookahead avec arbre de n-grams, respectivement. L'axe des

abscisses de la courbe de VLN, correspond à la valeur du seuil de génération des n-grams (alpha). Pour les modèles Lookahead et arbre de n-grams, l'axe des abscisses présente la profondeur d'analyse (taille des n-grams).

Les résultats montrent que la taille des modèles varie selon la longueur des n-grams utilisés pour la création des modèles Lookahead et arbre et selon la valeur du seuil de génération de l'algorithme VLN. En fait, un modèle de type VLN créé avec une faible valeur d'alpha, comportera un grand nombre de n-grams, alors qu'avec une grande valeur d'alpha l'algorithme VLN ne génèrera qu'un petit nombre de n-grams, résultant ainsi un modèle normal de taille plus petite. Les résultats de la figure 4.14 confirment cette analyse et montrent que la taille du modèle VLN augmente en diminuant la valeur d'alpha.

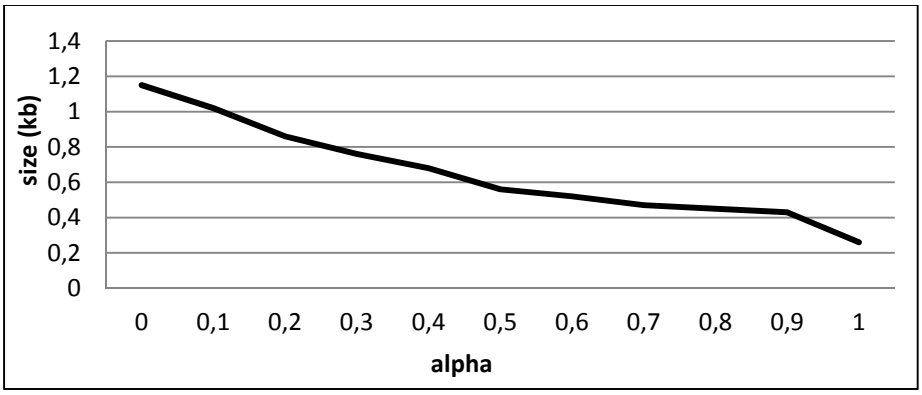


Figure 4.14 Taille du modèle VLN (7 traces)

Suivant la même logique, plus la taille de la fenêtre est grande, plus le nombre d'appels systèmes qu'elle comporte est grand et donc plus le modèle généré est volumineux. Ceci est décrit dans la figure 4.15 qui montre que la taille des modèles Lookahead et Arbre augmente si on augmente la taille des n-grams.

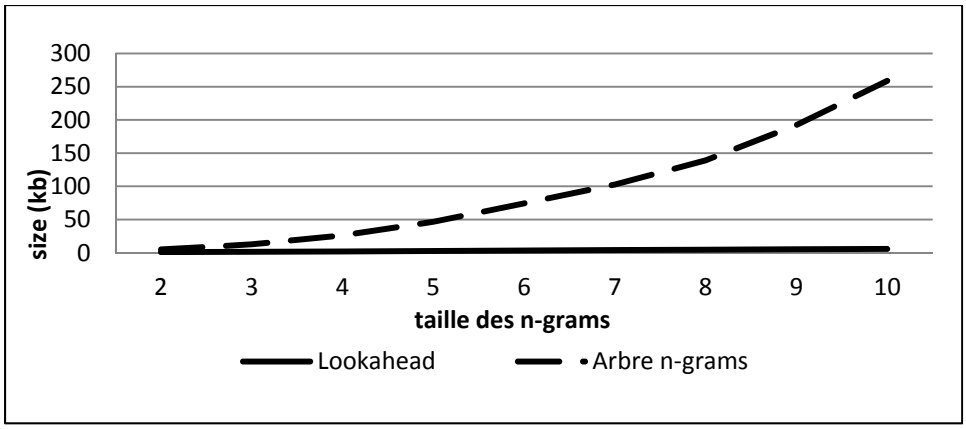


Figure 4.15 Taille des modèles Lookahead et Arbre n-gram (7 traces)

En outre, nous remarquons que les deux modèles VLN et Lookahead occupent moins d'espace mémoire, avec une valeur inférieure à 1.2 kb et 5.6 kb respectivement. Cela semble assez faible pour stocker plus qu'un seul modèle sur l'appareil.

En revanche, la taille du modèle d'arbre augmente d'une façon polynomiale pour atteindre plus de 250 Ko en utilisant 10-grams, ce qui est difficile à tolérer. En fait, si nous enregistrons des dizaines de modèles d'arbres avec 6-grams, nous atteignons rapidement plusieurs Mégabits (Mb) de données. Cependant, si nous limitons à des modèles avec 2-grams ou 3-grams, leur taille reste acceptable. La taille du modèle FSM est aussi acceptable avec une valeur de 24 kb.

4.2.5 Surcharge avant et après la compression

Pour pouvoir stocker plusieurs modèles aisément sur l'appareil mobile qui se limite en général à quelques Mégabits d'espace mémoire, nous avons utilisé l'outil de compression Zopfli (voir section 3.2.4). Nous avons mesuré la taille des traces (Figure 4.16) et la taille des modèles (Figure 4.17) avant et après la compression. L'axe des abscisses des figures 4.16 et 4.17 correspond respectivement au nombre de traces et au type du modèle.

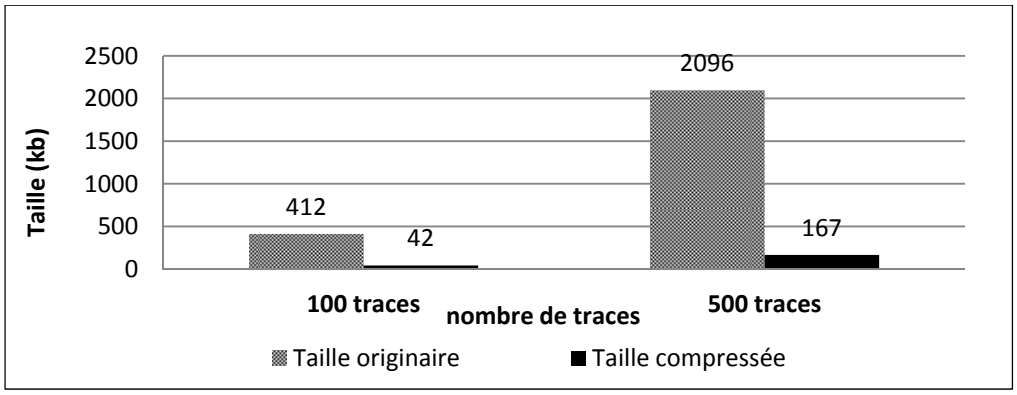


Figure 4.16 Tailles des Traces avant et après la compression avec Zopfli
(1 trace =1000 appels système)

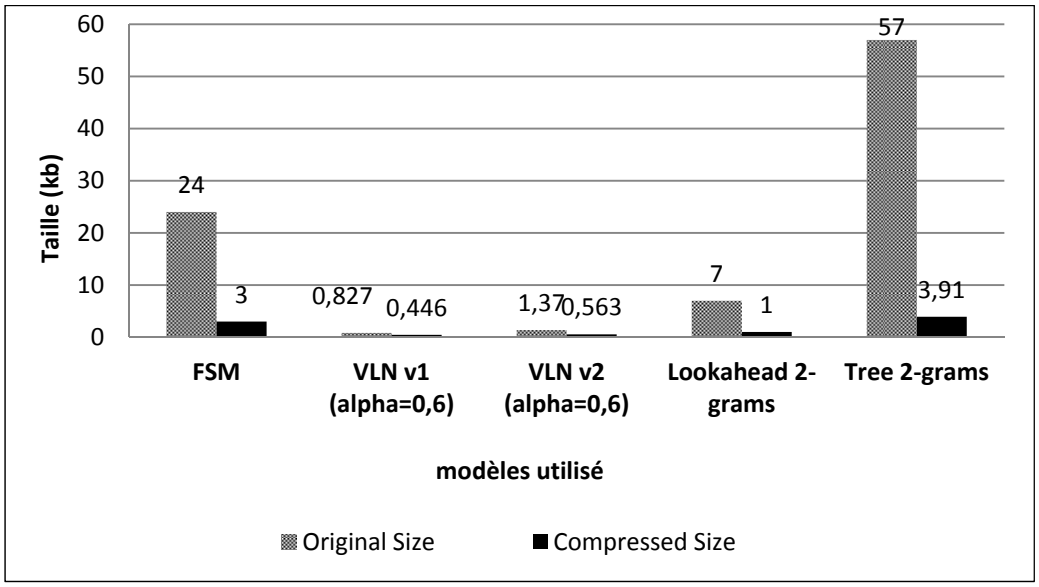


Figure 4.17 Tailles des modèles avant et après la compression avec Zopfli

D'après les résultats, on constate que l'outil de compression permet de réduire la taille des traces jusqu'à 92 % (de 2096 kb à 167kb). Il peut aussi fournir une densité de données très importante en réduisant les tailles des modèles jusqu'un facteur de 14 (modèle arbre). Cela semble assez faible pour stocker sans problème de nombreux modèles sur le téléphone.

4.3 Tests de détection

Cette section comprend les tests de détections faites avec les quatre algorithmes, plusieurs configurations et trois scénarios.

4.3.1 Scénarios de tests

Nous avons réalisé nos expérimentations selon différents scénarios et en utilisant un modèle normal formé avec 500 traces d'appels système, où une trace est composée de 1000 appels système.

Dans le premier scénario, nous avons testé nos approches avec 100 traces de l'ensemble d'apprentissages et 100 traces recueillies des versions malicieuses modifiées. Ce scénario présente le meilleur cas de test, puisqu'une grande partie de l'ensemble de tests est bien définie durant la phase d'apprentissage. Ici, on teste si les algorithmes de détection sont en mesure de distinguer correctement les comportements malicieux de ceux normaux. Un tel test peut fournir des résultats optimaux au niveau des taux de TP et TN pour les différentes approches de détections d'anomalies.

Le deuxième scénario consiste à évaluer la performance de détection en utilisant des données de test de 100 traces normales d'une nouvelle exécution de l'application avec 100 traces anormales recueillies des versions malicieuses modifiées. Ici, on teste à quel point le modèle normal est complet. Les résultats de ce test doivent présenter un faible taux de FP et un taux élevé de TP, ce qui montre que le modèle d'apprentissage englobe le maximum des séquences normales qui peuvent se présenter dans de différentes nouvelles exécutions de l'application.

Le troisième scénario consiste à appliquer le seuil optimal qui va être identifié à partir des deux scénarios précédents, sur des versions malicieuses du monde réel, afin de pouvoir

correctement détecter les attaques dans des applications malicieuses réelles et non modifiées. Ici on mesure le nombre des traces détectées comme anormales parmi 100 traces recueillies à partir des versions malicieuses réelles et en utilisant un certain seuil de détection prédéfini.

4.3.2 Scénario 1

Les figures 4.20 et 4.21 illustrent les courbes de fonction d'efficacité du récepteur (ROC) et de précisions (ACC), pour différentes valeurs d'alpha du modèle VLN, effectuées à partir de 200 traces de tests dont 100 appartiennent au modèle normal et 100 traces de l'application malicieuse modifiée. Les mêmes traces sont appliquées à l'algorithme Lookahead (figure 4.22 pour ROC et 4.23 pour ACC) et Arbre de n-grams (figure 4.24 pour ROC et 4.25 pour ACC) pour différentes longueurs de n-grams, ainsi que pour l'algorithme FSM (figure 4.18 pour ROC et 4.19 pour ACC).

Les courbes de ROC sont créées en reliant 101 points de données (*Voir* ANNEXE I) représentant chacun un seuil de 0 à 1,00 (par une unité de 0,01). Chaque point de données est une paire de taux de faux positifs (sur l'axe des abscisses) et taux de vrais positifs (sur l'axe des ordonnées).

Alors que chaque point de données dans les courbes de précisions (ACC) est une paire de taux de précision (sur l'axe des abscisses) et de la valeur du seuil de détection correspondante. (Sur l'axe des ordonnées).

Nous remarquons que dans le premier scénario, la courbe ROC du modèle FSM est superposée à l'axe vertical des taux de TP avec un taux de FP nulle et ceci pour tous les seuils de détections. Nous remarquons aussi que les modèles VLN et FSM possèdent des courbes de ROC idéales avec une très large aire sous la courbe (respectivement de 0,99 et 1) (figure 4.26) ceci montre que le TP est largement supérieur à celui de FP. Ces algorithmes

admettent aussi un taux de précision allant jusqu'à son maximum 1 et ceci en utilisant de faibles seuils de détection.

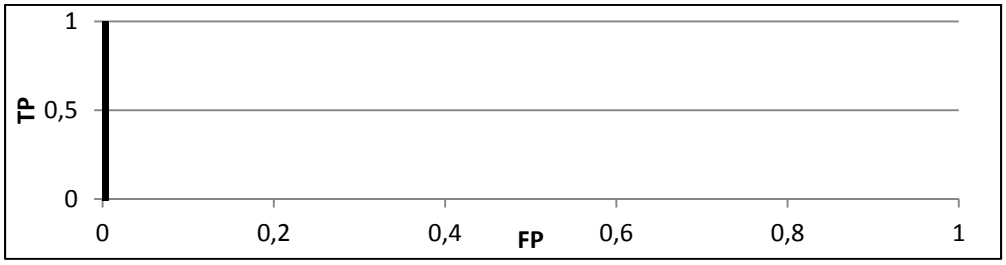


Figure 4.18 Courbe ROC du modèle FSM (scénario 1)

De plus, nous remarquons que le modèle FSM offre une précision maximale avec des seuils de détection allant de 0.0 à 0.57. Cependant avec des valeurs de seuils supérieures à 0.57, le modèle FSM devient beaucoup moins précis

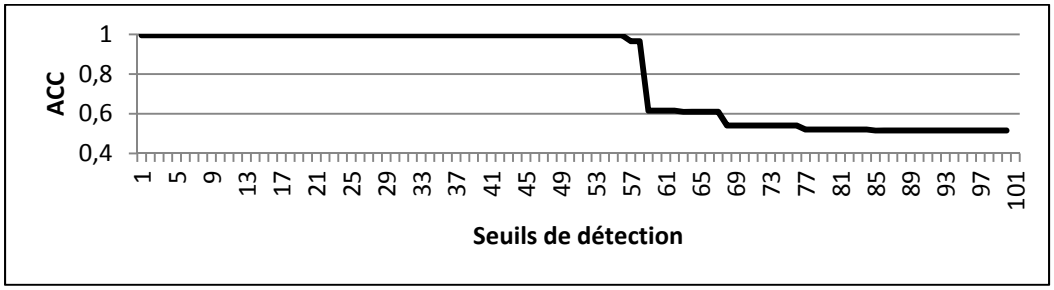


Figure 4.19 ACC du modèle FSM (scénario 1)

Nous remarquons aussi que les courbes de VLN tracées avec différentes valeurs d'alpha sont presque superposées et elles présentent donc des résultats semblables au niveau de la courbe ROC et ACC. Par contre, nous notons une légère différence dans l'intervalle de détection [0,15 et 0,27] de la courbe ACC de VLN (figure 4.21), où les résultats avec la valeur 0,4 d'alpha, surperforme les autres courbes de précisions.

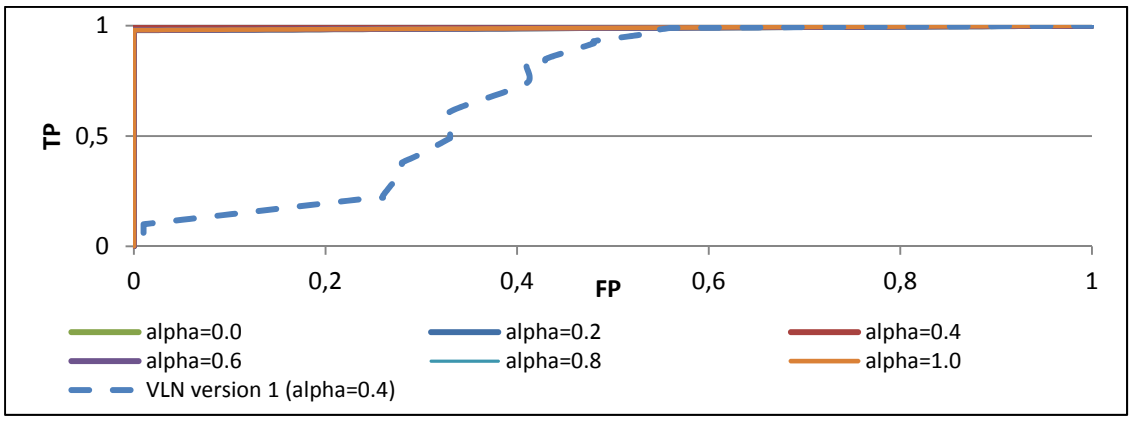


Figure 4.20 Courbe ROC du modèle VLN (scénario 1)

Nous avons également effectué une comparaison entre la nouvelle et l'ancienne version du modèle VLN. Il est clair que la nouvelle version surperforme l'ancienne en améliorant la précision d'un facteur de 1,5 (courbe 4.21) et le TP d'un facteur de 10 (courbe 4.20).

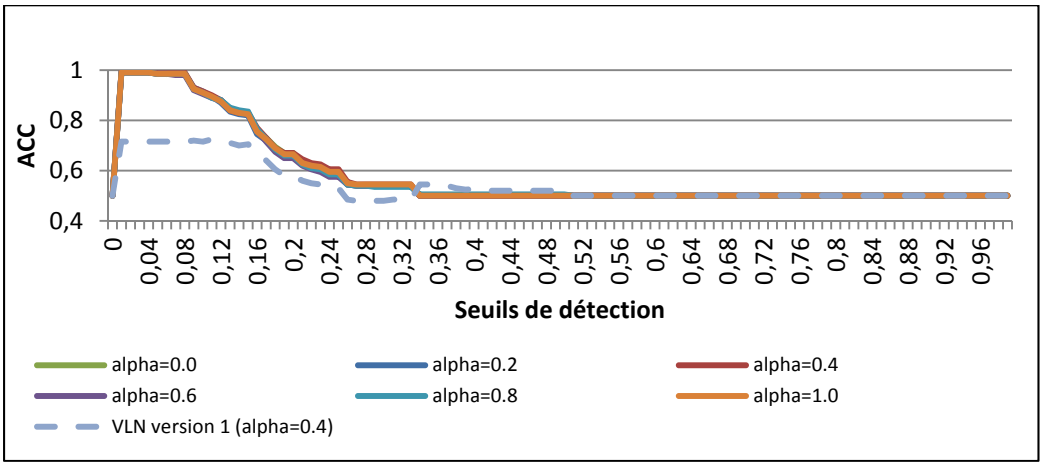


Figure 4.21 ACC du modèle N-grams de longueurs variées (scénario 1)

Concernant le modèle Lookahead, nous remarquons que son taux de TP et de précision augmente en diminuant la longueur de n Gram. Ce modèle nous permet de détecter jusqu'à 70 % des vraies attaques et une précision allant jusqu'à 85 % avec 2-grams et de faibles valeurs de seuils de détection (figure 4.23).

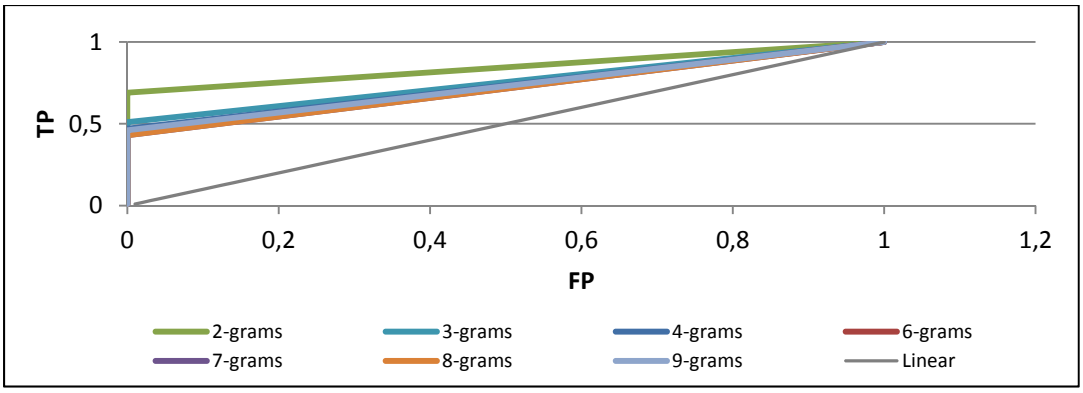


Figure 4.22 Courbe ROC du modèle Lookahead (scénario 1)

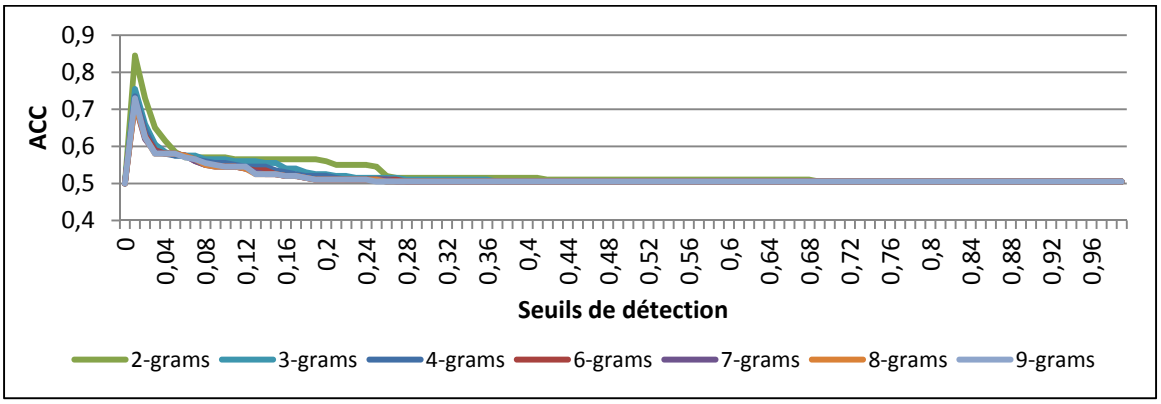


Figure 4.23 ACC du modèle Lookahead (scénario 1)

Par contre, les taux de TP et de précision pour le modèle de type Arbre, augmentent en augmentant la profondeur d'analyse et donc la taille de n Gram analysés.

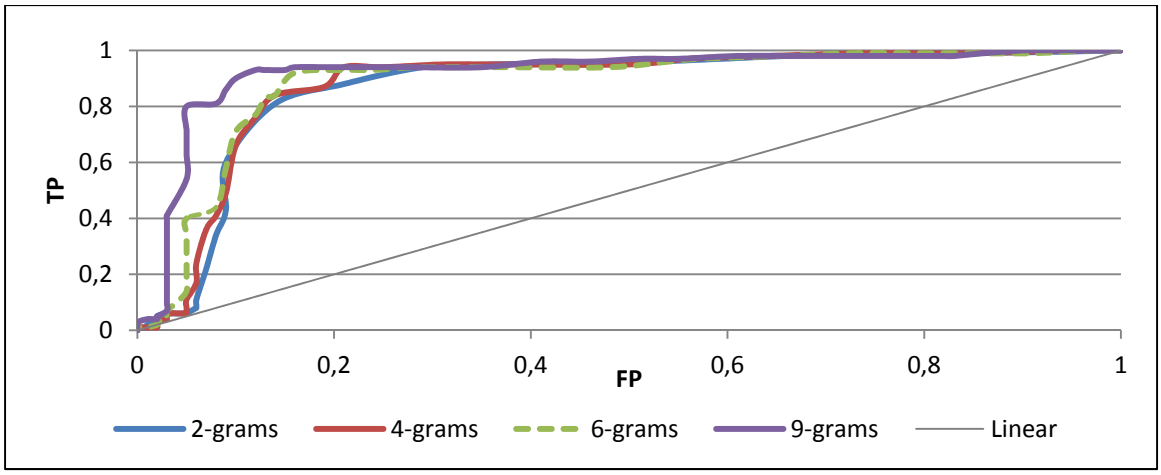


Figure 4.24 Courbe ROC du modèle Arbre n-grams (scénario 1)

Nous remarquons qu'avec 9-grams, le modèle est capable de détecter jusqu'à 93 % des attaques en générant moins de 14 % de FP. En plus, ce modèle possède un taux de précision allant jusqu'à 90 % e utilisant 9-grams. Cependant, si on se limite à analyser des 2-grams, le taux de précision diminue à 83 %, ce qui reste largement suffisant pour détecter assez de vraies attaques.

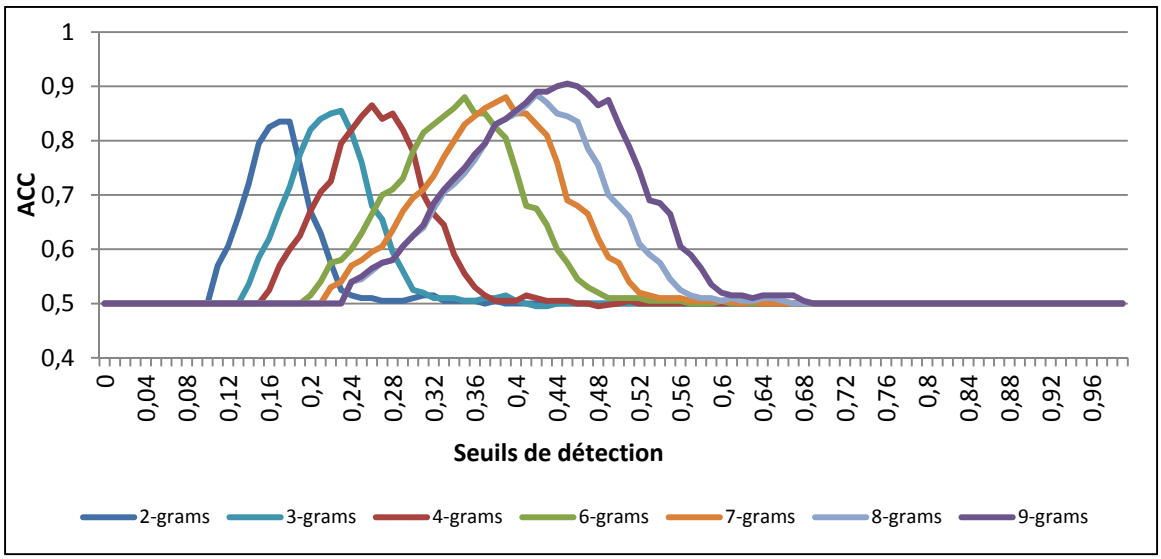


Figure 4.25 ACC du modèle Arbre n-grams (scénario 1)

Afin de comparer les différents modèles, nous avons mesuré l'aire sous leurs courbes ROC (AUC), comme le montre la Figure 4.26. L'axe des abscisses correspond à la valeur d'alpha pour le modèle VLN et à la taille de n-grams pour les modèles Lookahead et Arbre. Pour pouvoir dessiner les courbes de différents algorithmes dans le même graphe afin de bien comparer les modèles, nous avons multiplié la valeur d'alpha pour le modèle VLN, par un facteur de 10, pour garder la même échelle de grandeurs.

On note que le modèle FSM possède un AUC maximal égal à 1. Par conséquent et d'après la figure 4.26, il est clair que les modèles VLN et FSM surperforment les autres modèles pour n'importe quelle valeur d'alpha pour VLN. Nous remarquons également, que la valeur d'AUC pour le modèle de type arbre augmente en augmentant la taille de n-grams, pour atteindre une valeur de 0.92 en utilisant 9-grams. Au contraire, l'AUC du modèle Lookahead augmente en diminuant la profondeur d'analyse, pour atteindre une valeur de 0.85 avec 2-grams. Cependant, cette valeur reste inférieure à la valeur minimale obtenue avec le modèle de type arbre.

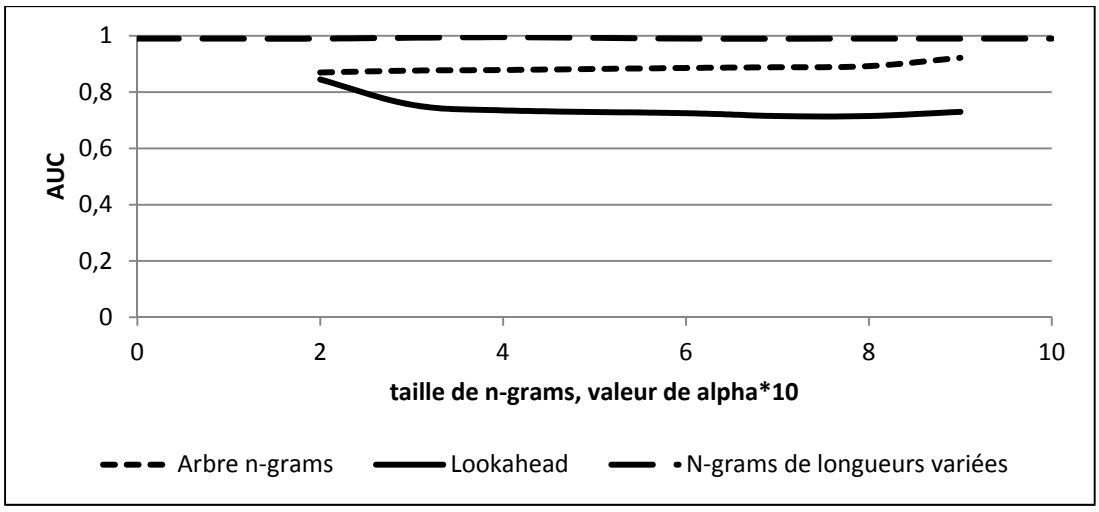


Figure 4.26 AUC de Lookahead, Arbre de n-grams et VLN (scénario 1)

4.3.3 Scénario 2

Les figures 4.28 et 4.29 illustrent les courbes de fonction d'efficacité du récepteur (ROC) et de précisions (ACC), pour différentes valeurs d'alpha du modèle VLN, effectuées à partir de 200 traces de tests dont 100 appartiennent à une nouvelle exécution de l'application et 100 traces de l'application malicieuse modifiée (*Voir* ANNEXE II). Les mêmes traces sont appliquées à l'algorithme Lookahead (Figure 4.30 pour ROC et 4.31 pour ACC) et Arbre de n-grams (Figure 4.32 pour ROC et 4.33 pour ACC) pour différentes longueurs de n-grams, ainsi que pour l'algorithme FSM (Figure 4.28 pour ROC et 4.27 pour ACC).

Nous remarquons que le deuxième scénario présente une faible dégradation des performances au niveau de TP, FP et de précisions, et ceci pour tous les algorithmes. Cette dégradation est causée par de nouveaux patterns non vus dans le modèle normal.

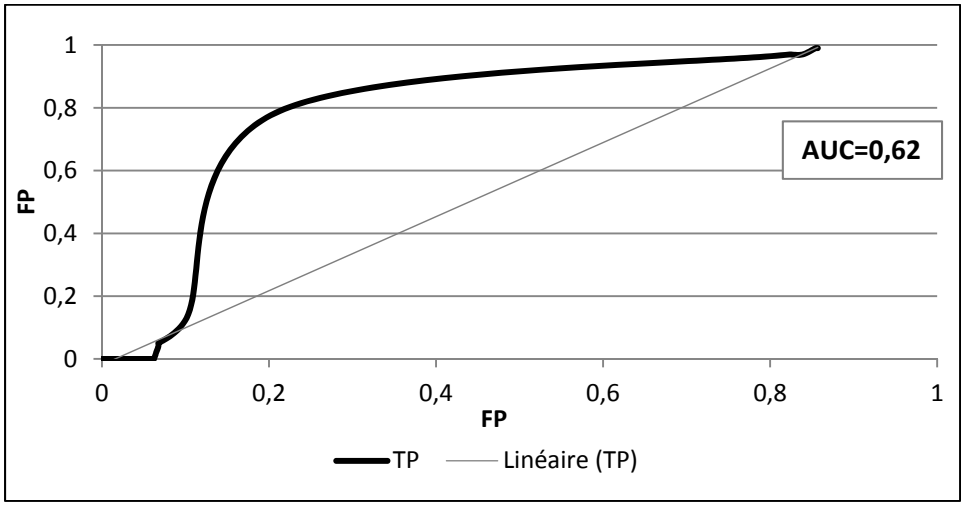


Figure 4.27 Courbe ROC du modèle FSM (scénario 2)

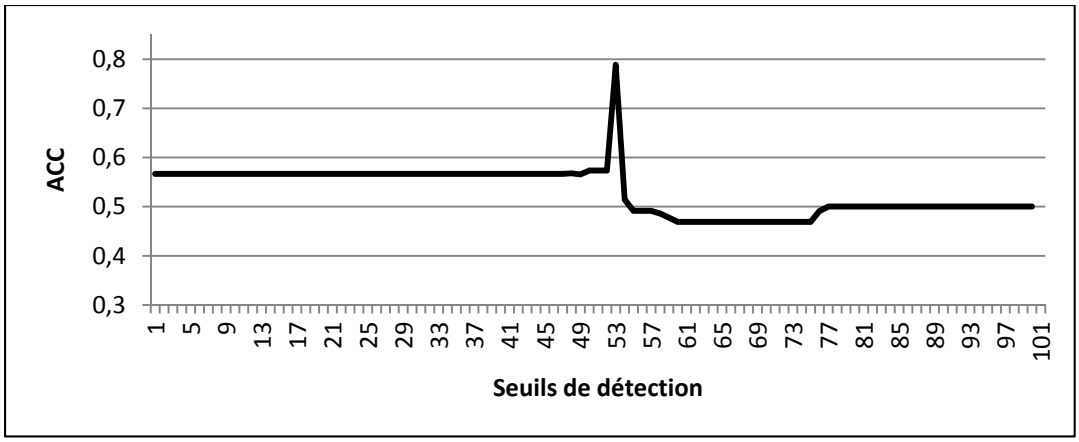


Figure 4.28 ACC du modèle FSM (scénario 2)

Nous remarquons aussi que malgré cette dégradation, les modèles VLN et FSM gardent le meilleur taux de TP et de précision par rapport aux autres modèles, avec des valeurs qui dépassent respectivement 90 % et 75 %. De plus, les résultats de VLN avec un alpha de 0.4 présentent de meilleures performances par rapport aux autres courbes de précisions et de TP.

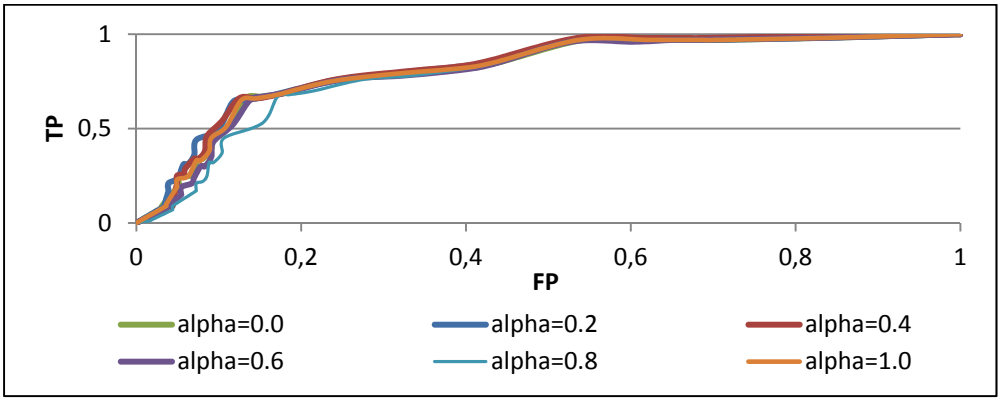


Figure 4.29 Courbe ROC du modèle VLN(scénario 2)

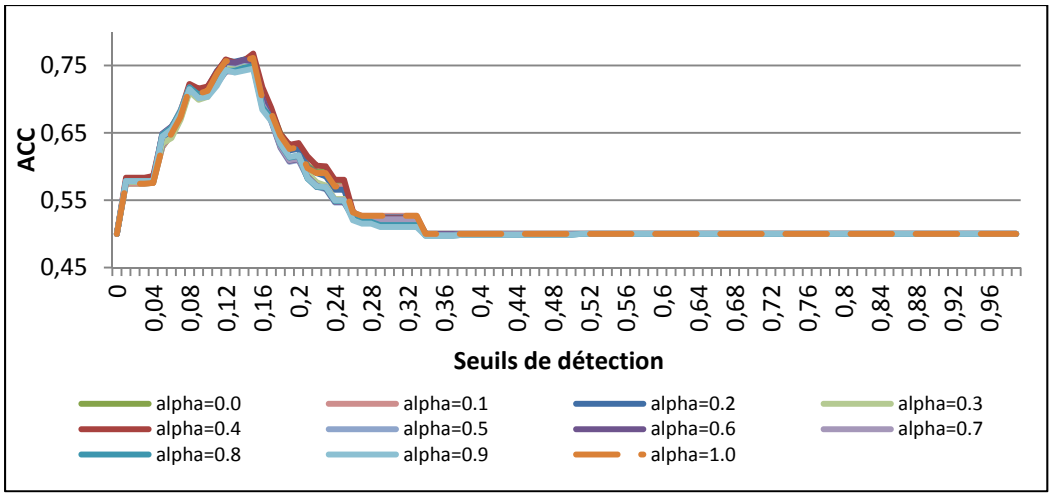


Figure 4.30 ACC du modèle VLN (scénario 2)

Concernant les modèles Lookahead et arbre de n-grams, nous remarquons que les deux taux de TP et FP augmentent avec la même vitesse. Cela signifie que les algorithmes génèrent un FP relativement élevé pour identifier les attaques, avec une précision maximale de 60 %. Cependant, les courbes ROCs restent sur la courbe linéaire qui correspond à un détecteur aléatoire. Ceci montre que les résultats de nos algorithmes restent fiables.

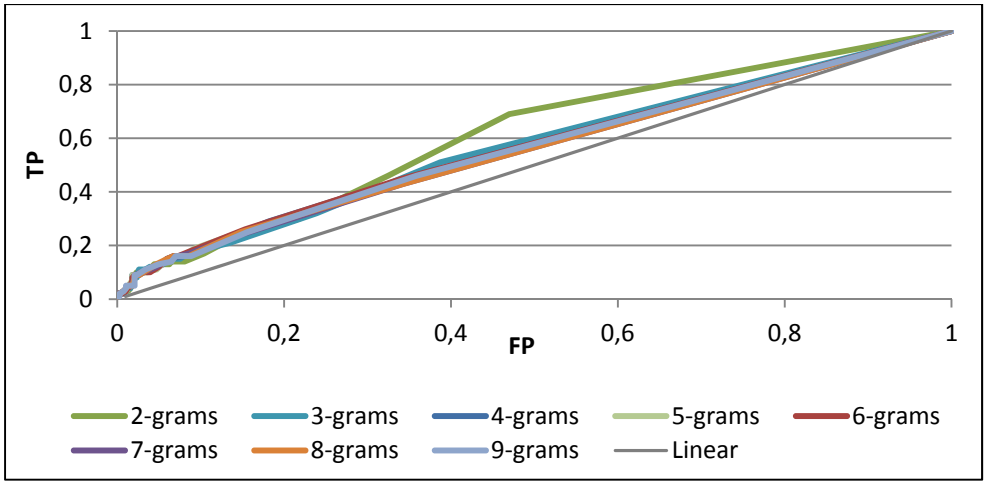


Figure 4.31 Courbe ROC du modèle Lookahead (scénario 2)

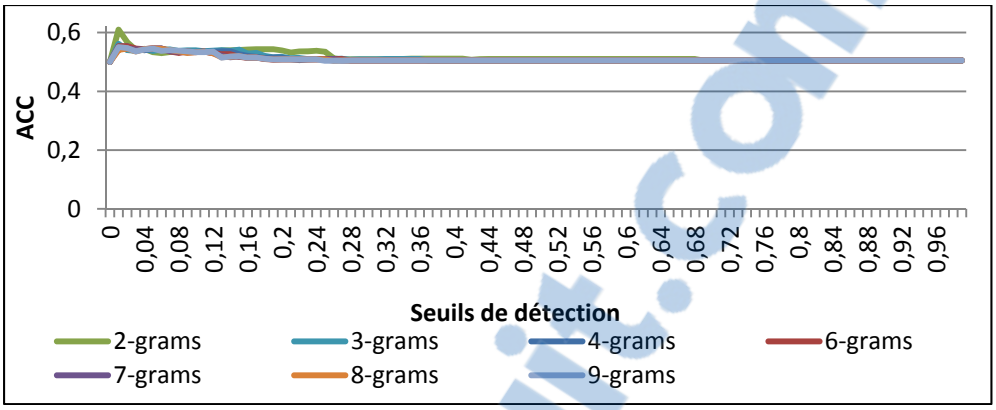


Figure 4.32 ACC du modèle Lookahead (scénario 2)

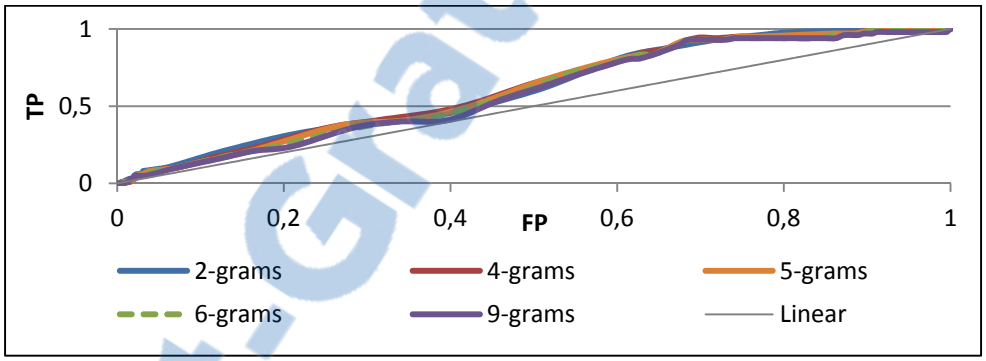


Figure 4.33 Courbe ROC du modèle Arbre n-grams (scénario 2)

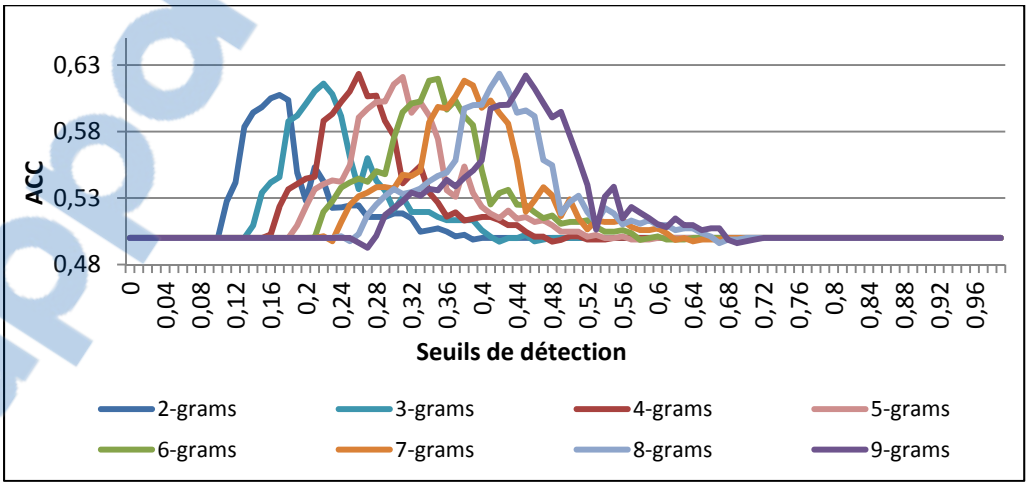


Figure 4.34 ACC du modèle Arbre n-grams (scénario 2)

Comme dans le scénario 1, nous avons comparé les différents algorithmes en nous basant sur l'aire sous leurs courbes ROC (AUC), comme le montre la figure 4.35. On note aussi que le modèle FSM possède un AUC maximal égal à 0.62. Par conséquent et d'après cette figure, il est clair que les modèles VLN et FSM présentent de meilleures performances par rapport aux autres modèles et ceci pour n'importe quelle valeur d'alpha pour VLN.

Nous remarquons également, que les valeurs d'AUC pour les différents modèles sont supérieures à la valeur 0.5 qui correspond à un classificateur aléatoire.

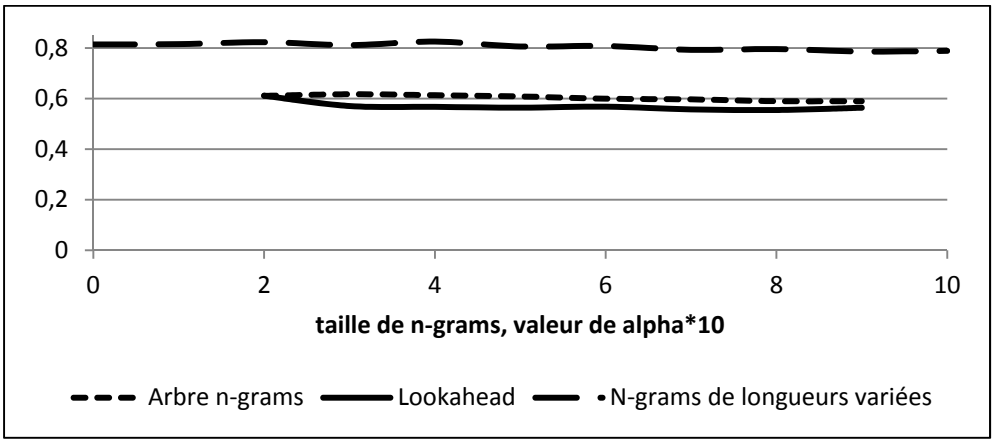


Figure 4.35 AUC de Lookahead, Arbre de n-grams et VLN (scénario 2)

4.3.4 Scénario 3

D'après les deux scénarios précédents, nous avons pu identifier les meilleurs paramètres et seuils de détection, pour chaque algorithme, qui nous permettent d'avoir des résultats optimaux au niveau de la précision de détection et de la consommation des ressources, selon le tableau 4.3.

Tableau 4-3
Configuration optimale des algorithmes de détection

Approche	Lookahead	Arbre	VLN (v2)	FSM

Paramètres		2-grams		3-grams		Alpha=0.4				-------	
Seuil optimal		0,01		0,22		**0,15**		0,04		0,52	
Scénario		S1	S2	S1	S2	S1	S2	S1	S2	S1	S2
ACC		0,84	0,6	0,85	0,61	0,83	0,76	0,99	0,58	0,99	0,78
TP		0,7	0,7	0,91	0,91	0,66	0,66	0,99	0,99	0,99	0,79
FP		0	0,47	0,21	0,67	0	0,12	0	0,81	0	0,12
% RAM	numérisation	3.75		4,3		6.3				7.8	
	Modèle	1.55		2.4		3.7				6	
% CPU	numérisation	19.7		28		52				62	
	Modèle	19.6		25.81		35				60	
Stockage (Kb)		0.78		12.77		0.68				24	

Ce test nous permet d'appliquer le seuil optimal identifié à partir des deux scénarios précédents afin de correctement détecter les attaques dans des applications malicieuses réelles et non modifiées

Les figures 4.36, 4.37 et 4.38 illustrent les courbes de détection pour les applications malicieuses du monde réel, « AngryBirds », « CandyStar » et « NinjaChicken ». Ces courbes ont été faites à partir des configurations optimales identifiées à partir des deux scénarios précédents et prédéfinies selon le tableau 4-3.

D'après ces figures, nous remarquons que pour chaque application, il y a un certain nombre de traces qui ont été classées comme malicieuses avec tous les algorithmes. Nous remarquons aussi que le modèle arbre détecte toujours plus de traces, ceci est causé par son taux de FP élevé. Cependant, la plupart des traces détectées par FSM et VLN, sont aussi détectées par les autres algorithmes, ce qui confirme leurs taux élevés de TP et de précision.

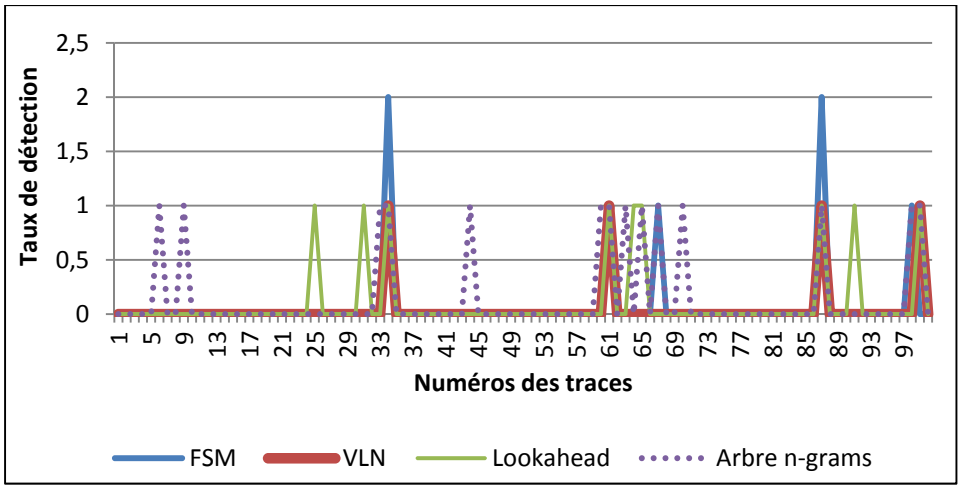


Figure 4.36 Taux de détection avec l'application Angrybirds (scénario 3)

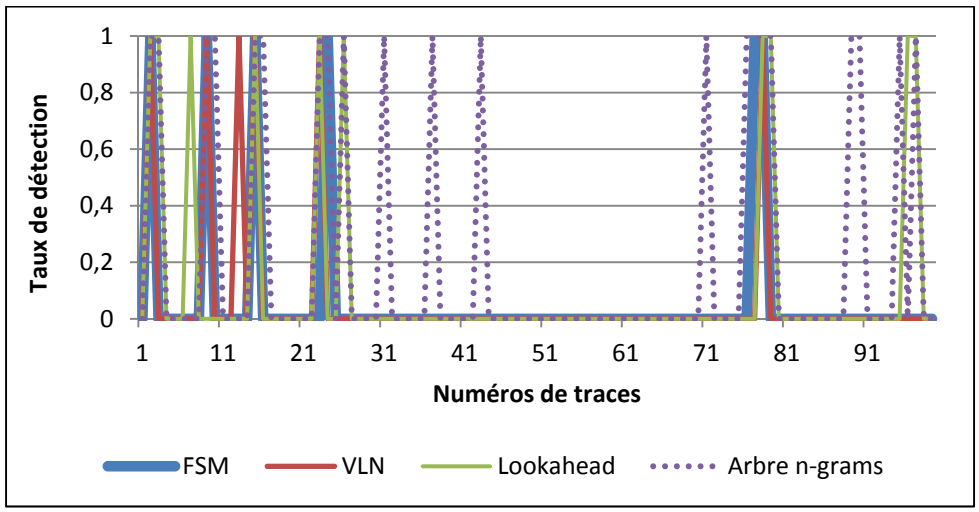


Figure 4.37 Taux de détection avec l'application CandyStars (scénario 3)

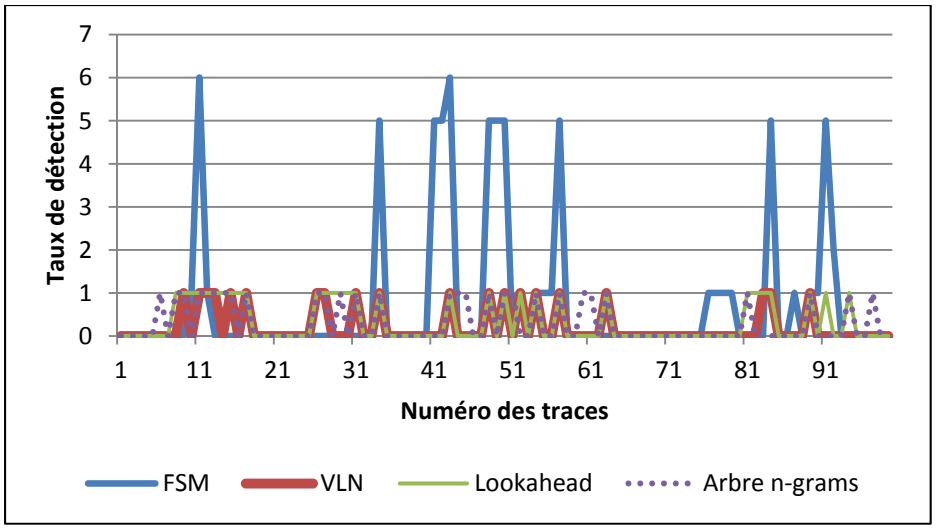


Figure 4.38 Taux de détection avec l'application NinjaChicken (scénario 3)

4.4 Discussion

D'autant que l'on peut dire à partir de la littérature, notre cadre de détection est le premier analyseur comportemental et dynamique des logiciels malveillants sur le système Android, permettant d'établir un compromis entre la détection et la consommation des ressources, sans compter entièrement sur un serveur distant.

Les résultats de TP et FP sont meilleurs que celles des systèmes de détection comportementale basés sur le système Android (Sanz et al., 2013b; Shabtai et al., 2014), avec une précision qui peut atteindre 100 %.

De plus, notre approche nous permet de surveiller plus qu'une application à la fois sans affecter l'exécution normale du système et en consommant le moins de ressources possible qui peuvent aller jusqu'à 4 % de mémoire vive et 26 % de CPU (ce qui correspond à 13% de CPU total), en surveillant trois applications en parallèle. La consommation optimale lors de

la numérisation d'une seule application est de 3.75 % de RAM, ce qui est plus faible de celle des autres travaux (Salman et al., 2014; Wang et al., 2013)
En outre, notre cadre de détection est capable d'exécuter différents algorithmes de détection avec différentes configurations. Chaque algorithme a ses avantages et ses inconvénients. Lookahead a l'avantage d'être le modèle le plus léger en termes de consommation de mémoire vive et de processeur, alors qu'il ne permet de détecter que 69 % des vrais malwares.

Le modèle de type arbre de n Gram, permet de fournir un taux de vrais positifs plus élevé que celui de Lookahead et qui peut atteindre jusqu'à 90 %. Cependant son taux de détection augmente en argumentant la profondeur d'analyse, ce qui va donc augmenter sa consommation de ressources et la taille du modèle d'une façon polynomiale pour atteindre plus de 250 Ko en utilisant 10-grams.

Les modèles FSM et VLN permettent d'offrir une précision maximale de 100 % et ils peuvent aussi détecter jusqu'à 100 % des vrais malwares (sans FP). Cependant le modèle FSM est relativement gourmand en ressources par rapport aux autres modèles, alors que la consommation en CPU, RAM et mémoire de stockage pour VLN, augmente en diminuant la valeur du seuil de génération « alpha ». Malgré ça, le modèle VLN est le moins gourmand en espace mémoire et ceci pour toutes les valeurs d'alpha.

Il faut, néanmoins, noter que la taille des modèles reste acceptable avec la compression de données fournies par le module de stockage. Ce module est basé sur l'outil « Zopfli » qui permet de réduire la taille des données jusqu'à 14 fois.

Sur la base des résultats expérimentaux, un compromis entre la précision et la consommation de ressources peut être établi en variant le nombre de paires utilisées, le seuil d'extraction des n-grams et le seuil de détection. Par conséquent, nous ajustons ces paramètres selon le

tableau 4.1. Cela améliore les performances des algorithmes et fournit une plus grande précision (nous avons testé la précision de détection avec les trois applications Android).

CONCLUSION

La sécurité des systèmes embarqués, qui représentent une large gamme de produits tels que les téléphones portables, les routeurs de réseau, les cartes à puce..., est une préoccupation importante pour les entreprises et les exploitants de ces produits.

Beaucoup de solutions de sécurité ont été développées et mises en place sur la plus part des systèmes d'exploitation, tels que les ADSs. La détection d'anomalies sur les systèmes traditionnels est l'un des domaines les plus définis et largement étudiés. Cependant, la détection d'anomalies dans les appareils mobiles suit généralement les mêmes principes et exigent donc des opérations de calculs lourds, ce qui rend difficile son déploiement sur les appareils à ressources limités. En effet, l'environnement des téléphones intelligents à ressources limitées impose des limites strictes sur l'utilisation du processeur, de la batterie et des ressources mémoires disponibles. De plus, les ADSs mobiles possèdent généralement des ressources résidant sur l'infonuagique. Ainsi puisque le dispositif mobile passe par une variété de réseaux avec des configurations souvent inconnues, les ADS mobiles deviennent de plus en plus vulnérables. Toutefois l'absence d'une solution mobile, légère et efficace de détection d'anomalies constitue un grand défi pour son déploiement sur les systèmes à ressources limitées d'où l'importance de concevoir un ADS capable de protéger ce type de dispositifs.

Durant notre travail de recherche, nous nous sommes intéressés à proposer un cadre adaptatif de détection d'anomalies, basé sur le système Android, qui malgré sa forte croissance reste toujours l'un des systèmes les plus vulnérables. Ce cadre peut utiliser un serveur distant pour améliorer sa performance de détection, tout en étant capable de tourner parfaitement sans en dépendre.

Tout d'abord, nous avons commencé par une étude approfondie sur l'architecture et la sécurité du système Android afin d'identifier ses problèmes et ses lacunes. Nous avons ciblé notre étude aux attaques par injection du code malveillant dans les applications mobiles.
Puis, nous avons analysé et comparé les plus importants travaux connexes, ce qui nous a permis d'identifier les principaux défis de la détection d'anomalies sur les systèmes à ressources limitées. À cette phase, nous n'avons pas pu trouver des travaux sur le compromis entre la détection d'anomalies et la consommation de ressources sur l'hôte Android, ce qui a rendu notre phase de recherche plus difficile. Cette difficulté a été surmontée grâce à une recherche encore plus approfondie sur les différentes solutions existantes sur les autres plateformes mobiles comme Symbian (Bickford et al., 2011).

En nous basant sur cette revue, nous avons ensuite conçu notre cadre de détection avec ses différents modules; module de collecte de données, de traitement de données, de modélisation et de numérisation, de stockage, de profilage et de connexion avec le serveur.

Au niveau du module de collecte de données, nous avons rencontré un problème de déploiement de l'outil d'instrumentation LTTng sur le système Android. Suite à ce problème nous avons choisi de travailler avec l'outil « strace », en tant que meilleurs analyseur des appels systèmes compatibles avec Android au moment de choix.

Le module de profilage est capable d'extraire des informations sur l'état du système (CPU, RAM, Batterie, Réseaux), alors que le module de stockage est basé sur l'outil de compression des données « Zopfli » fourni par Google.

Au niveau du module de modélisation et de numérisation, nous avons implémenté et étudié quatre algorithmes de détection d'anomalies basés sur les courtes séquences d'appels systèmes (n-grams) ; Lookahead, Arbre n-grams, VLN (avec ses deux versions) et FSM. Au niveau du test, nous n'avons pas trouvé des versions normales et malicieuses de la même application Android, puisque le marché d'Android (Google Play) ne garde que la dernière

version bégnine de chaque application. Pour cela, nous avons essayé de trouver les versions malicieuses les plus près possibles de celles normales. Cependant, nous n'avons trouvé que 3 versions malicieuses de 3 applications. Pour élargir la base de données des tests, nous avons modifié les trois applications normales, en les injectant des fonctionnalités anormales. En outre, nous avons testé notre cadre sur les applications Android normales, leurs versions modifiées et leurs versions réelles malveillantes.

Les résultats expérimentaux montrent qu'en fonction de la taille de n-gram, la valeur de seuil de détection et la valeur du seuil d'extraction des n-grams pour VLN, un taux de détection de 100 % peut être atteint, tout en maintenant un faible taux de faux positifs et en consommant qu'une quantité raisonnable de ressources. Ces résultats nous ont permis de concevoir un décideur dynamique qui surveille l'état des ressources du système et sélectionne la meilleure configuration des algorithmes de détection qui permet le meilleur compromis entre la précision de la détection et la consommation de ressources.

Dans un contexte de recherche scientifique, toute approche proposée est en perpétuelle évolution. Il y a toujours des points à améliorer et des détails à discuter. Dans l'état actuel de notre architecture, plusieurs points à améliorer ressortent. Au niveau de la collecte des données, nous pouvons utiliser l'outil Lttng au lieu de Strace, puisque Lttng est plus léger et permettant aussi une analyse en temps réel. Au niveau du module de modélisation et de numérisation, nous pouvons améliorer la performance de détection de l'algorithme Lookahead qui présente des faibles taux de détection par rapport aux modèles. En effet, pour des besoins d'adaptation aux ressources limitées et pour pouvoir comparer différents algorithmes, nous avons utilisé des versions simplifiées des algorithmes de détection de référence tout en améliorant la capacité de détection. Cependant, vu que le modèle Lookahead est léger, nous l'avons pris tel qu'il est, sans l'améliorer. Pour cela nous prévoyons de concevoir une version de ce modèle, plus performante en termes de taux de détections. Nous pouvons ainsi, définir un seuil de détection dynamique pour chaque entrée

de la table des pairs regroupés, afin d'éviter les faux négatifs des sous séquences d'appels systèmes qui n'apparaissent pas dans les traces normales.

Nous prévoyons, également, d'étudier d'autres classes d'algorithmes de détection d'anomalies et de recueillir localement ou à partir de serveur un ensemble de données de formation plus complet et plus représentatif. En outre, nous allons élargir notre base de données de test en utilisant des applications Android avec des techniques de reconditionnement plus avancées. Enfin, nous avons l'intention de migrer vers le multiplateforme, afin de tester notre solution sur les différentes plateformes mobiles et embarquées.

ANNEXE I

RÉSULTATS DE TEST DE DÉTECTION (SCÉNARIO 1)

Tableau 4-4 Test de détection pour le modèle Lookahead avec 2-grams (scénario 1)

threshold	FP	TN	TP	FN	Accuracy= (TP+TN)/(TP+TN+FP+FN)
0	1	0	1	0	0,5
0,01	**0**	**1**	**0,69**	**0,31**	**0,845**
0,02	0	1	0,46	0,54	0,73
0,03	0	1	0,3	0,7	0,65
0,04	0	1	0,23	0,77	0,615
0,05	0	1	0,17	0,83	0,585
0,06	0	1	0,14	0,86	0,57
0,07	0	1	0,14	0,86	0,57
0,08	0	1	0,14	0,86	0,57
0,09	0	1	0,14	0,86	0,57
0,1	0	1	0,14	0,86	0,57
0,11	0	1	0,13	0,87	0,565
0,12	0	1	0,13	0,87	0,565
0,13	0	1	0,13	0,87	0,565
0,14	0	1	0,13	0,87	0,565
0,15	0	1	0,13	0,87	0,565
0,16	0	1	0,13	0,87	0,565
0,17	0	1	0,13	0,87	0,565
0,18	0	1	0,13	0,87	0,565
0,19	0	1	0,13	0,87	0,565
0,2	0	1	0,12	0,88	0,56
0,21	0	1	0,1	0,9	0,55
0,22	0	1	0,1	0,9	0,55
0,23	0	1	0,1	0,9	0,55
0,24	0	1	0,1	0,9	0,55
0,25	0	1	0,09	0,91	0,545
0,26	0	1	0,04	0,96	0,52
0,27	0	1	0,03	0,97	0,515

0,28	0	1	0,03	0,97	0,515
0,29	0	1	0,03	0,97	0,515
0,3	0	1	0,03	0,97	0,515
0,31	0	1	0,03	0,97	0,515
0,32	0	1	0,03	0,97	0,515
0,33	0	1	0,03	0,97	0,515
0,34	0	1	0,03	0,97	0,515
0,35	0	1	0,03	0,97	0,515
0,36	0	1	0,03	0,97	0,515
0,37	0	1	0,03	0,97	0,515
0,38	0	1	0,03	0,97	0,515
0,39	0	1	0,03	0,97	0,515
0,4	0	1	0,03	0,97	0,515
0,41	0	1	0,03	0,97	0,515
0,42	0	1	0,02	0,98	0,51
0,43	0	1	0,02	0,98	0,51
0,44	0	1	0,02	0,98	0,51
0,45	0	1	0,02	0,98	0,51
0,46	0	1	0,02	0,98	0,51
0,47	0	1	0,02	0,98	0,51
0,48	0	1	0,02	0,98	0,51
0,49	0	1	0,02	0,98	0,51
0,5	0	1	0,02	0,98	0,51
0,51	0	1	0,02	0,98	0,51
0,52	0	1	0,02	0,98	0,51
0,53	0	1	0,02	0,98	0,51
0,54	0	1	0,02	0,98	0,51
0,55	0	1	0,02	0,98	0,51
0,56	0	1	0,02	0,98	0,51
0,57	0	1	0,02	0,98	0,51
0,58	0	1	0,02	0,98	0,51
0,59	0	1	0,02	0,98	0,51
0,6	0	1	0,02	0,98	0,51
0,61	0	1	0,02	0,98	0,51
0,62	0	1	0,02	0,98	0,51
0,63	0	1	0,02	0,98	0,51
0,64	0	1	0,02	0,98	0,51
0,65	0	1	0,02	0,98	0,51

0,66	0	1	0,02	0,98	0,51
0,67	0	1	0,02	0,98	0,51
0,68	0	1	0,02	0,98	0,51
0,69	0	1	0,01	0,99	0,505
0,7	0	1	0,01	0,99	0,505
0,71	0	1	0,01	0,99	0,505
0,72	0	1	0,01	0,99	0,505
0,73	0	1	0,01	0,99	0,505
0,74	0	1	0,01	0,99	0,505
0,75	0	1	0,01	0,99	0,505
0,76	0	1	0,01	0,99	0,505
0,77	0	1	0,01	0,99	0,505
0,78	0	1	0,01	0,99	0,505
0,79	0	1	0,01	0,99	0,505
0,8	0	1	0,01	0,99	0,505
0,81	0	1	0,01	0,99	0,505
0,82	0	1	0,01	0,99	0,505
0,83	0	1	0,01	0,99	0,505
0,84	0	1	0,01	0,99	0,505
0,85	0	1	0,01	0,99	0,505
0,86	0	1	0,01	0,99	0,505
0,87	0	1	0,01	0,99	0,505
0,88	0	1	0,01	0,99	0,505
0,89	0	1	0,01	0,99	0,505
0,9	0	1	0,01	0,99	0,505
0,91	0	1	0,01	0,99	0,505
0,92	0	1	0,01	0,99	0,505
0,93	0	1	0,01	0,99	0,505
0,94	0	1	0,01	0,99	0,505
0,95	0	1	0,01	0,99	0,505
0,96	0	1	0,01	0,99	0,505
0,97	0	1	0,01	0,99	0,505
0,98	0	1	0,01	0,99	0,505
0,99	0	1	0,01	0,99	0,505

Tableau 4-5 Test de détection pour le modèle Arbre avec 3-grams (scénario 1)

threshold	FP	TN	TP	FN	Accuracy= (TP+TN)/(TP+TN+FP+FN)
0	1	0	1	0	0,5
1	1	0	1	0	0,5
2	1	0	1	0	0,5
3	1	0	1	0	0,5
4	1	0	1	0	0,5
5	1	0	1	0	0,5
6	1	0	1	0	0,5
7	1	0	1	0	0,5
8	1	0	1	0	0,5
9	1	0	1	0	0,5
10	1	0	1	0	0,5
11	1	0	1	0	0,5
12	1	0	1	0	0,5
13	1	0	1	0	0,5
14	0,92	0,08	0,99	0,01	0,535
15	0,82	0,18	0,99	0,01	0,585
16	0,75	0,25	0,99	0,01	0,62
17	0,64	0,36	0,98	0,02	0,67
18	0,54	0,46	0,97	0,03	0,715
19	0,4	0,6	0,95	0,05	0,775
20	0,31	0,69	0,95	0,05	0,82
21	0,26	0,74	0,94	0,06	0,84
22	**0,21**	**0,79**	**0,91**	**0,09**	**0,85**
23	0,14	0,86	0,85	0,15	0,855
24	0,13	0,87	0,76	0,24	0,815
25	0,09	0,91	0,61	0,39	0,76
26	0,09	0,91	0,45	0,55	0,68
27	0,08	0,92	0,39	0,61	0,655
28	0,07	0,93	0,26	0,74	0,595
29	0,06	0,94	0,18	0,82	0,56
30	0,06	0,94	0,11	0,89	0,525
31	0,06	0,94	0,1	0,9	0,52
32	0,05	0,95	0,07	0,93	0,51
33	0,05	0,95	0,07	0,93	0,51
34	0,05	0,95	0,07	0,93	0,51
35	0,05	0,95	0,06	0,94	0,505

36	0,04	0,96	0,05	0,95	0,505
37	0,03	0,97	0,05	0,95	0,51
38	0,03	0,97	0,05	0,95	0,51
39	0,02	0,98	0,05	0,95	0,515
40	0,02	0,98	0,03	0,97	0,505
41	0,02	0,98	0,02	0,98	0,5
42	0,02	0,98	0,01	0,99	0,495
43	0,02	0,98	0,01	0,99	0,495
44	0,01	0,99	0,01	0,99	0,5
45	0,01	0,99	0,01	0,99	0,5
46	0	1	0	1	0,5
47	0	1	0	1	0,5
48	0	1	0	1	0,5
49	0	1	0	1	0,5
50	0	1	0	1	0,5
51	0	1	0	1	0,5
52	0	1	0	1	0,5
53	0	1	0	1	0,5
54	0	1	0	1	0,5
55	0	1	0	1	0,5
56	0	1	0	1	0,5
57	0	1	0	1	0,5
58	0	1	0	1	0,5
59	0	1	0	1	0,5
60	0	1	0	1	0,5
61	0	1	0	1	0,5
62	0	1	0	1	0,5
63	0	1	0	1	0,5
64	0	1	0	1	0,5
65	0	1	0	1	0,5
66	0	1	0	1	0,5
67	0	1	0	1	0,5
68	0	1	0	1	0,5
69	0	1	0	1	0,5
70	0	1	0	1	0,5
71	0	1	0	1	0,5
72	0	1	0	1	0,5
73	0	1	0	1	0,5

74	0	1	0	1	0,5
75	0	1	0	1	0,5
76	0	1	0	1	0,5
77	0	1	0	1	0,5
78	0	1	0	1	0,5
79	0	1	0	1	0,5
80	0	1	0	1	0,5
81	0	1	0	1	0,5
82	0	1	0	1	0,5
83	0	1	0	1	0,5
84	0	1	0	1	0,5
85	0	1	0	1	0,5
86	0	1	0	1	0,5
87	0	1	0	1	0,5
88	0	1	0	1	0,5
89	0	1	0	1	0,5
90	0	1	0	1	0,5
91	0	1	0	1	0,5
92	0	1	0	1	0,5
93	0	1	0	1	0,5
94	0	1	0	1	0,5
95	0	1	0	1	0,5
96	0	1	0	1	0,5
97	0	1	0	1	0,5
98	0	1	0	1	0,5
99	0	1	0	1	0,5

Tableau 4-6 Test de détection pour le modèle VLN avec alpha=0,4 (scénario 1)

threshold	FP	TN	TP	FN	Accuracy= (TP+TN)/(TP+TN+FP+FN)
0	1	0	1	0	0,5
0,01	0	1	0,99	0,01	0,995
0,02	0	1	0,99	0,01	0,995
0,03	0	1	0,99	0,01	0,995
0,04	0	1	0,99	0,01	0,995
0,05	0	1	0,98	0,02	0,99

0,06	0	1	0,98	0,02	0,99
0,07	0	1	0,98	0,02	0,99
0,08	0	1	0,98	0,02	0,99
0,09	0	1	0,86	0,14	0,93
0,1	0	1	0,83	0,17	0,915
0,11	0	1	0,8	0,2	0,9
0,12	0	1	0,76	0,24	0,88
0,13	0	1	0,68	0,32	0,84
0,14	0	1	0,66	0,34	0,83
0,15	**0**	**1**	**0,66**	**0,34**	**0,83**
0,16	0	1	0,54	0,46	0,77
0,17	0	1	0,46	0,54	0,73
0,18	0	1	0,38	0,62	0,69
0,19	0	1	0,34	0,66	0,67
0,2	0	1	0,34	0,66	0,67
0,21	0	1	0,29	0,71	0,645
0,22	0	1	0,26	0,74	0,63
0,23	0	1	0,25	0,75	0,625
0,24	0	1	0,21	0,79	0,605
0,25	0	1	0,21	0,79	0,605
0,26	0	1	0,11	0,89	0,555
0,27	0	1	0,09	0,91	0,545
0,28	0	1	0,09	0,91	0,545
0,29	0	1	0,09	0,91	0,545
0,3	0	1	0,09	0,91	0,545
0,31	0	1	0,09	0,91	0,545
0,32	0	1	0,09	0,91	0,545
0,33	0	1	0,09	0,91	0,545
0,34	0	1	0	1	0,5
0,35	0	1	0	1	0,5
0,36	0	1	0	1	0,5
0,37	0	1	0	1	0,5
0,38	0	1	0	1	0,5
0,39	0	1	0	1	0,5
0,4	0	1	0	1	0,5
0,41	0	1	0	1	0,5
0,42	0	1	0	1	0,5
0,43	0	1	0	1	0,5

0,44	0	1	0	1	0,5
0,45	0	1	0	1	0,5
0,46	0	1	0	1	0,5
0,47	0	1	0	1	0,5
0,48	0	1	0	1	0,5
0,49	0	1	0	1	0,5
0,5	0	1	0	1	0,5
0,51	0	1	0	1	0,5
0,52	0	1	0	1	0,5
0,53	0	1	0	1	0,5
0,54	0	1	0	1	0,5
0,55	0	1	0	1	0,5
0,56	0	1	0	1	0,5
0,57	0	1	0	1	0,5
0,58	0	1	0	1	0,5
0,59	0	1	0	1	0,5
0,6	0	1	0	1	0,5
0,61	0	1	0	1	0,5
0,62	0	1	0	1	0,5
0,63	0	1	0	1	0,5
0,64	0	1	0	1	0,5
0,65	0	1	0	1	0,5
0,66	0	1	0	1	0,5
0,67	0	1	0	1	0,5
0,68	0	1	0	1	0,5
0,69	0	1	0	1	0,5
0,7	0	1	0	1	0,5
0,71	0	1	0	1	0,5
0,72	0	1	0	1	0,5
0,73	0	1	0	1	0,5
0,74	0	1	0	1	0,5
0,75	0	1	0	1	0,5
0,76	0	1	0	1	0,5
0,77	0	1	0	1	0,5
0,78	0	1	0	1	0,5
0,79	0	1	0	1	0,5
0,8	0	1	0	1	0,5
0,81	0	1	0	1	0,5

0,82	0	1	0	1	0,5
0,83	0	1	0	1	0,5
0,84	0	1	0	1	0,5
0,85	0	1	0	1	0,5
0,86	0	1	0	1	0,5
0,87	0	1	0	1	0,5
0,88	0	1	0	1	0,5
0,89	0	1	0	1	0,5
0,9	0	1	0	1	0,5
0,91	0	1	0	1	0,5
0,92	0	1	0	1	0,5
0,93	0	1	0	1	0,5
0,94	0	1	0	1	0,5
0,95	0	1	0	1	0,5
0,96	0	1	0	1	0,5
0,97	0	1	0	1	0,5
0,98	0	1	0	1	0,5
0,99	0	1	0	1	0,5

Tableau 4-7 Test de détection pour le modèle FSM (scénario 1)

threshold	FP	TN	TP	FN	Accuracy= (TP+TN)/(TP+TN+FP+FN)
0	0,85714286	0,14285714	0,99	0,01	0,56642857
1	0,85714286	0,14285714	0,99	0,01	0,56642857
2	0,85714286	0,14285714	0,99	0,01	0,56642857
3	0,85714286	0,14285714	0,99	0,01	0,56642857
4	0,85714286	0,14285714	0,99	0,01	0,56642857
5	0,85714286	0,14285714	0,99	0,01	0,56642857
6	0,85714286	0,14285714	0,99	0,01	0,56642857
7	0,85714286	0,14285714	0,99	0,01	0,56642857
8	0,85714286	0,14285714	0,99	0,01	0,56642857
9	0,85714286	0,14285714	0,99	0,01	0,56642857
10	0,85714286	0,14285714	0,99	0,01	0,56642857
11	0,85714286	0,14285714	0,99	0,01	0,56642857
12	0,85714286	0,14285714	0,99	0,01	0,56642857
13	0,85714286	0,14285714	0,99	0,01	0,56642857
14	0,85714286	0,14285714	0,99	0,01	0,56642857

15	0,85714286	0,14285714	0,99	0,01	0,56642857
16	0,85714286	0,14285714	0,99	0,01	0,56642857
17	0,85714286	0,14285714	0,99	0,01	0,56642857
18	0,85714286	0,14285714	0,99	0,01	0,56642857
19	0,85714286	0,14285714	0,99	0,01	0,56642857
20	0,85714286	0,14285714	0,99	0,01	0,56642857
21	0,85714286	0,14285714	0,99	0,01	0,56642857
22	0,85714286	0,14285714	0,99	0,01	0,56642857
23	0,85714286	0,14285714	0,99	0,01	0,56642857
24	0,85714286	0,14285714	0,99	0,01	0,56642857
25	0,85714286	0,14285714	0,99	0,01	0,56642857
26	0,85714286	0,14285714	0,99	0,01	0,56642857
27	0,85714286	0,14285714	0,99	0,01	0,56642857
28	0,85714286	0,14285714	0,99	0,01	0,56642857
29	0,85714286	0,14285714	0,99	0,01	0,56642857
30	0,85714286	0,14285714	0,99	0,01	0,56642857
31	0,85714286	0,14285714	0,99	0,01	0,56642857
32	0,85714286	0,14285714	0,99	0,01	0,56642857
33	0,85714286	0,14285714	0,99	0,01	0,56642857
34	0,85714286	0,14285714	0,99	0,01	0,56642857
35	0,85714286	0,14285714	0,99	0,01	0,56642857
36	0,85714286	0,14285714	0,99	0,01	0,56642857
37	0,85714286	0,14285714	0,99	0,01	0,56642857
38	0,85714286	0,14285714	0,99	0,01	0,56642857
39	0,85714286	0,14285714	0,99	0,01	0,56642857
40	0,85714286	0,14285714	0,99	0,01	0,56642857
41	0,85714286	0,14285714	0,99	0,01	0,56642857
42	0,85714286	0,14285714	0,99	0,01	0,56642857
43	0,85714286	0,14285714	0,99	0,01	0,56642857
44	0,85714286	0,14285714	0,99	0,01	0,56642857
45	0,85714286	0,14285714	0,99	0,01	0,56642857
46	0,85714286	0,14285714	0,99	0,01	0,56642857
47	0,85454545	0,14545455	0,99	0,01	0,56772727
48	0,83896104	0,16103896	0,97	0,03	0,56551948
49	0,82337662	0,17662338	0,97	0,03	0,57331169
50	0,82337662	0,17662338	0,97	0,03	0,57331169
51	0,82337662	0,17662338	0,97	0,03	0,57331169
52	**0,21298701**	**0,78701299**	**0,79**	**0,21**	**0,78850649**

53	0,1012987	0,8987013	0,13	0,87	0,51435065
54	0,06753247	0,93246753	0,05	0,95	0,49123377
55	0,06753247	0,93246753	0,05	0,95	0,49123377
56	0,06753247	0,93246753	0,05	0,95	0,49123377
57	0,06753247	0,93246753	0,04	0,96	0,48623377
58	0,06493506	0,93506494	0,02	0,98	0,47753247
59	0,06233766	0,93766234	0	1	0,46883117
60	0,06233766	0,93766234	0	1	0,46883117
61	0,06233766	0,93766234	0	1	0,46883117
62	0,06233766	0,93766234	0	1	0,46883117
63	0,06233766	0,93766234	0	1	0,46883117
64	0,06233766	0,93766234	0	1	0,46883117
65	0,06233766	0,93766234	0	1	0,46883117
66	0,06233766	0,93766234	0	1	0,46883117
67	0,06233766	0,93766234	0	1	0,46883117
68	0,06233766	0,93766234	0	1	0,46883117
69	0,06233766	0,93766234	0	1	0,46883117
70	0,06233766	0,93766234	0	1	0,46883117
71	0,06233766	0,93766234	0	1	0,46883117
72	0,06233766	0,93766234	0	1	0,46883117
73	0,06233766	0,93766234	0	1	0,46883117
74	0,06233766	0,93766234	0	1	0,46883117
75	0,01818182	0,98181818	0	1	0,49090909
76	0	1	0	1	0,5
77	0	1	0	1	0,5
78	0	1	0	1	0,5
79	0	1	0	1	0,5
80	0	1	0	1	0,5
81	0	1	0	1	0,5
82	0	1	0	1	0,5
83	0	1	0	1	0,5
84	0	1	0	1	0,5
85	0	1	0	1	0,5
86	0	1	0	1	0,5
87	0	1	0	1	0,5
88	0	1	0	1	0,5
89	0	1	0	1	0,5
90	0	1	0	1	0,5

91	0	1	0	1	0,5
92	0	1	0	1	0,5
93	0	1	0	1	0,5
94	0	1	0	1	0,5
95	0	1	0	1	0,5
96	0	1	0	1	0,5
97	0	1	0	1	0,5
98	0	1	0	1	0,5
99	0	1	0	1	0,5

ANNEXE II

RÉSULTATS DE TEST DE DÉTECTION (SCÉNARIO 2)

Tableau 4-8 Test de détection pour le modèle Lookahead avec 2-grams (scénario 2)

threshold	FP	TN	TP	FN	Accuracy= (TP+TN)/(TP+TN+FP+FN)
0	1	0	1	0	0,5
0,01	**0,47012987**	**0,52987013**	**0,69**	**0,31**	**0,609935065**
0,02	0,32467532	0,67532468	0,46	0,54	0,567662338
0,03	0,21818182	0,78181818	0,3	0,7	0,540909091
0,04	0,14025974	0,85974026	0,23	0,77	0,54487013
0,05	0,1038961	0,8961039	0,17	0,83	0,533051948
0,06	0,08051948	0,91948052	0,14	0,86	0,52974026
0,07	0,07272727	0,92727273	0,14	0,86	0,533636364
0,08	0,07012987	0,92987013	0,14	0,86	0,534935065
0,09	0,06493506	0,93506494	0,14	0,86	0,537532468
0,1	0,06493506	0,93506494	0,14	0,86	0,537532468
0,11	0,06233766	0,93766234	0,13	0,87	0,533831169
0,12	0,05974026	0,94025974	0,13	0,87	0,53512987
0,13	0,05194805	0,94805195	0,13	0,87	0,539025974
0,14	0,05194805	0,94805195	0,13	0,87	0,539025974
0,15	0,04935065	0,95064935	0,13	0,87	0,540324675
0,16	0,04675325	0,95324675	0,13	0,87	0,541623377
0,17	0,04415584	0,95584416	0,13	0,87	0,542922078
0,18	0,04415584	0,95584416	0,13	0,87	0,542922078
0,19	0,04415584	0,95584416	0,13	0,87	0,542922078
0,2	0,04155844	0,95844156	0,12	0,88	0,539220779
0,21	0,03636364	0,96363636	0,1	0,9	0,531818182
0,22	0,02857143	0,97142857	0,1	0,9	0,535714286
0,23	0,02857143	0,97142857	0,1	0,9	0,535714286
0,24	0,02337662	0,97662338	0,1	0,9	0,538311688
0,25	0,02077922	0,97922078	0,09	0,91	0,53461039
0,26	0,01558442	0,98441558	0,04	0,96	0,512207792

0,27	0,01298701	0,98701299	0,03	0,97	0,508506494
0,28	0,01298701	0,98701299	0,03	0,97	0,508506494
0,29	0,01298701	0,98701299	0,03	0,97	0,508506494
0,3	0,01298701	0,98701299	0,03	0,97	0,508506494
0,31	0,01298701	0,98701299	0,03	0,97	0,508506494
0,32	0,01298701	0,98701299	0,03	0,97	0,508506494
0,33	0,01038961	0,98961039	0,03	0,97	0,509805195
0,34	0,01038961	0,98961039	0,03	0,97	0,509805195
0,35	0,00779221	0,99220779	0,03	0,97	0,511103896
0,36	0,00779221	0,99220779	0,03	0,97	0,511103896
0,37	0,00779221	0,99220779	0,03	0,97	0,511103896
0,38	0,00779221	0,99220779	0,03	0,97	0,511103896
0,39	0,00779221	0,99220779	0,03	0,97	0,511103896
0,4	0,00779221	0,99220779	0,03	0,97	0,511103896
0,41	0,00779221	0,99220779	0,03	0,97	0,511103896
0,42	0,00519481	0,99480519	0,02	0,98	0,507402597
0,43	0	1	0,02	0,98	0,51
0,44	0	1	0,02	0,98	0,51
0,45	0	1	0,02	0,98	0,51
0,46	0	1	0,02	0,98	0,51
0,47	0	1	0,02	0,98	0,51
0,48	0	1	0,02	0,98	0,51
0,49	0	1	0,02	0,98	0,51
0,5	0	1	0,02	0,98	0,51
0,51	0	1	0,02	0,98	0,51
0,52	0	1	0,02	0,98	0,51
0,53	0	1	0,02	0,98	0,51
0,54	0	1	0,02	0,98	0,51
0,55	0	1	0,02	0,98	0,51
0,56	0	1	0,02	0,98	0,51
0,57	0	1	0,02	0,98	0,51
0,58	0	1	0,02	0,98	0,51
0,59	0	1	0,02	0,98	0,51
0,6	0	1	0,02	0,98	0,51
0,61	0	1	0,02	0,98	0,51
0,62	0	1	0,02	0,98	0,51
0,63	0	1	0,02	0,98	0,51
0,64	0	1	0,02	0,98	0,51

0,65	0	1	0,02	0,98	0,51
0,66	0	1	0,02	0,98	0,51
0,67	0	1	0,02	0,98	0,51
0,68	0	1	0,02	0,98	0,51
0,69	0	1	0,01	0,99	0,505
0,7	0	1	0,01	0,99	0,505
0,71	0	1	0,01	0,99	0,505
0,72	0	1	0,01	0,99	0,505
0,73	0	1	0,01	0,99	0,505
0,74	0	1	0,01	0,99	0,505
0,75	0	1	0,01	0,99	0,505
0,76	0	1	0,01	0,99	0,505
0,77	0	1	0,01	0,99	0,505
0,78	0	1	0,01	0,99	0,505
0,79	0	1	0,01	0,99	0,505
0,8	0	1	0,01	0,99	0,505
0,81	0	1	0,01	0,99	0,505
0,82	0	1	0,01	0,99	0,505
0,83	0	1	0,01	0,99	0,505
0,84	0	1	0,01	0,99	0,505
0,85	0	1	0,01	0,99	0,505
0,86	0	1	0,01	0,99	0,505
0,87	0	1	0,01	0,99	0,505
0,88	0	1	0,01	0,99	0,505
0,89	0	1	0,01	0,99	0,505
0,9	0	1	0,01	0,99	0,505
0,91	0	1	0,01	0,99	0,505
0,92	0	1	0,01	0,99	0,505
0,93	0	1	0,01	0,99	0,505
0,94	0	1	0,01	0,99	0,505
0,95	0	1	0,01	0,99	0,505
0,96	0	1	0,01	0,99	0,505
0,97	0	1	0,01	0,99	0,505
0,98	0	1	0,01	0,99	0,505
0,99	0	1	0,01	0,99	0,505

Tableau 4-9 Test de détection pour le modèle Arbre avec 3-grams (scénario 2)

threshold	FP	TN	TP	FN	Accuracy= (TP+TN)/(TP+TN+FP+FN)
0	1	0	1	0	0,5
1	1	0	1	0	0,5
2	1	0	1	0	0,5
3	1	0	1	0	0,5
4	1	0	1	0	0,5
5	1	0	1	0	0,5
6	1	0	1	0	0,5
7	1	0	1	0	0,5
8	1	0	1	0	0,5
9	1	0	1	0	0,5
10	1	0	1	0	0,5
11	1	0	1	0	0,5
12	1	0	1	0	0,5
13	1	0	1	0	0,5
14	0,97142857	0,02857143	0,99	0,01	0,50928571
15	0,92207792	0,07792208	0,99	0,01	0,53396104
16	0,90649351	0,09350649	0,99	0,01	0,54175325
17	0,88831169	0,11168831	0,98	0,02	0,54584416
18	0,79480519	0,20519481	0,97	0,03	0,5875974
19	0,76623377	0,23376623	0,95	0,05	0,59188312
20	0,74805195	0,25194805	0,95	0,05	0,60097403
21	0,71948052	0,28051948	0,94	0,06	0,61025974
22	**0,67792208**	**0,32207792**	**0,91**	**0,09**	**0,61603896**
23	0,63376623	0,36623377	0,85	0,15	0,60811688
24	0,57662338	0,42337662	0,76	0,24	0,59168831
25	0,48831169	0,51168831	0,61	0,39	0,56084416
26	0,37662338	0,62337662	0,45	0,55	0,53668831
27	0,27012987	0,72987013	0,39	0,61	0,55993506
28	0,17402597	0,82597403	0,26	0,74	0,54298701
29	0,10909091	0,89090909	0,18	0,82	0,53545455
30	0,06753247	0,93246753	0,11	0,89	0,52123377
31	0,03896104	0,96103896	0,1	0,9	0,53051948
32	0,03116883	0,96883117	0,07	0,93	0,51941558
33	0,03116883	0,96883117	0,07	0,93	0,51941558
34	0,03116883	0,96883117	0,07	0,93	0,51941558
35	0,02857143	0,97142857	0,06	0,94	0,51571429

36	0,02337662	0,97662338	0,05	0,95	0,51331169
37	0,02337662	0,97662338	0,05	0,95	0,51331169
38	0,02337662	0,97662338	0,05	0,95	0,51331169
39	0,02337662	0,97662338	0,05	0,95	0,51331169
40	0,01818182	0,98181818	0,03	0,97	0,50590909
41	0,01818182	0,98181818	0,02	0,98	0,50090909
42	0,01558442	0,98441558	0,01	0,99	0,49720779
43	0,01038961	0,98961039	0,01	0,99	0,49980519
44	0,01038961	0,98961039	0,01	0,99	0,49980519
45	0,00519481	0,99480519	0,01	0,99	0,5024026
46	0,00519481	0,99480519	0	1	0,4974026
47	0,0025974	0,9974026	0	1	0,4987013
48	0	1	0	1	0,5
49	0	1	0	1	0,5
50	0	1	0	1	0,5
51	0	1	0	1	0,5
52	0	1	0	1	0,5
53	0	1	0	1	0,5
54	0	1	0	1	0,5
55	0	1	0	1	0,5
56	0	1	0	1	0,5
57	0	1	0	1	0,5
58	0	1	0	1	0,5
59	0	1	0	1	0,5
60	0	1	0	1	0,5
61	0	1	0	1	0,5
62	0	1	0	1	0,5
63	0	1	0	1	0,5
64	0	1	0	1	0,5
65	0	1	0	1	0,5
66	0	1	0	1	0,5
67	0	1	0	1	0,5
68	0	1	0	1	0,5
69	0	1	0	1	0,5
70	0	1	0	1	0,5
71	0	1	0	1	0,5
72	0	1	0	1	0,5
73	0	1	0	1	0,5

74	0	1	0	1	0,5
75	0	1	0	1	0,5
76	0	1	0	1	0,5
77	0	1	0	1	0,5
78	0	1	0	1	0,5
79	0	1	0	1	0,5
80	0	1	0	1	0,5
81	0	1	0	1	0,5
82	0	1	0	1	0,5
83	0	1	0	1	0,5
84	0	1	0	1	0,5
85	0	1	0	1	0,5
86	0	1	0	1	0,5
87	0	1	0	1	0,5
88	0	1	0	1	0,5
89	0	1	0	1	0,5
90	0	1	0	1	0,5
91	0	1	0	1	0,5
92	0	1	0	1	0,5
93	0	1	0	1	0,5
94	0	1	0	1	0,5
95	0	1	0	1	0,5
96	0	1	0	1	0,5
97	0	1	0	1	0,5
98	0	1	0	1	0,5
99	0	1	0	1	0,5

Tableau 4-10 Test de détection pour le modèle VLN avec alpha=0,4 (scénario 2)

threshold	FP	TN	TP	FN	Accuracy= (TP+TN)/(TP+TN+FP+FN)
0	1	0	1	0	0,5
0,01	0,82337662	0,17662338	0,99	0,01	0,58331169
0,02	0,82337662	0,17662338	0,99	0,01	0,58331169
0,03	0,82337662	0,17662338	0,99	0,01	0,58331169
0,04	0,81818182	0,18181818	0,99	0,01	0,58590909
0,05	0,68831169	0,31168831	0,98	0,02	0,64584416

0,06	0,66493506	0,33506494	0,98	0,02	0,65753247
0,07	0,61818182	0,38181818	0,98	0,02	0,68090909
0,08	0,53506494	0,46493506	0,98	0,02	0,72246753
0,09	0,42857143	0,57142857	0,86	0,14	0,71571429
0,1	0,39220779	0,60779221	0,83	0,17	0,7188961
0,11	0,31948052	0,68051948	0,8	0,2	0,74025974
0,12	0,24155844	0,75844156	0,76	0,24	0,75922078
0,13	0,17142857	0,82857143	0,68	0,32	0,75428571
0,14	0,14545455	0,85454545	0,66	0,34	0,75727273
0,15	0,12467532	0,87532468	0,66	0,34	0,76766234
0,16	0,1038961	0,8961039	0,54	0,46	0,71805195
0,17	0,08571429	0,91428571	0,46	0,54	0,68714286
0,18	0,08311688	0,91688312	0,38	0,62	0,64844156
0,19	0,07532468	0,92467532	0,34	0,66	0,63233766
0,2	0,07012987	0,92987013	0,34	0,66	0,63493506
0,21	0,05974026	0,94025974	0,29	0,71	0,61512987
0,22	0,05714286	0,94285714	0,26	0,74	0,60142857
0,23	0,04935065	0,95064935	0,25	0,75	0,60032468
0,24	0,04935065	0,95064935	0,21	0,79	0,58032468
0,25	0,04935065	0,95064935	0,21	0,79	0,58032468
0,26	0,04415584	0,95584416	0,11	0,89	0,53292208
0,27	0,04415584	0,95584416	0,09	0,91	0,52292208
0,28	0,04415584	0,95584416	0,09	0,91	0,52292208
0,29	0,04415584	0,95584416	0,09	0,91	0,52292208
0,3	0,04415584	0,95584416	0,09	0,91	0,52292208
0,31	0,04415584	0,95584416	0,09	0,91	0,52292208
0,32	0,04415584	0,95584416	0,09	0,91	0,52292208
0,33	0,04415584	0,95584416	0,09	0,91	0,52292208
0,34	0	1	0	1	0,5
0,35	0	1	0	1	0,5
0,36	0	1	0	1	0,5
0,37	0	1	0	1	0,5
0,38	0	1	0	1	0,5
0,39	0	1	0	1	0,5
0,4	0	1	0	1	0,5
0,41	0	1	0	1	0,5
0,42	0	1	0	1	0,5
0,43	0	1	0	1	0,5

0,44	0	1	0	1	0,5
0,45	0	1	0	1	0,5
0,46	0	1	0	1	0,5
0,47	0	1	0	1	0,5
0,48	0	1	0	1	0,5
0,49	0	1	0	1	0,5
0,5	0	1	0	1	0,5
0,51	0	1	0	1	0,5
0,52	0	1	0	1	0,5
0,53	0	1	0	1	0,5
0,54	0	1	0	1	0,5
0,55	0	1	0	1	0,5
0,56	0	1	0	1	0,5
0,57	0	1	0	1	0,5
0,58	0	1	0	1	0,5
0,59	0	1	0	1	0,5
0,6	0	1	0	1	0,5
0,61	0	1	0	1	0,5
0,62	0	1	0	1	0,5
0,63	0	1	0	1	0,5
0,64	0	1	0	1	0,5
0,65	0	1	0	1	0,5
0,66	0	1	0	1	0,5
0,67	0	1	0	1	0,5
0,68	0	1	0	1	0,5
0,69	0	1	0	1	0,5
0,7	0	1	0	1	0,5
0,71	0	1	0	1	0,5
0,72	0	1	0	1	0,5
0,73	0	1	0	1	0,5
0,74	0	1	0	1	0,5
0,75	0	1	0	1	0,5
0,76	0	1	0	1	0,5
0,77	0	1	0	1	0,5
0,78	0	1	0	1	0,5
0,79	0	1	0	1	0,5
0,8	0	1	0	1	0,5
0,81	0	1	0	1	0,5

0,82	0	1	0	1	0,5
0,83	0	1	0	1	0,5
0,84	0	1	0	1	0,5
0,85	0	1	0	1	0,5
0,86	0	1	0	1	0,5
0,87	0	1	0	1	0,5
0,88	0	1	0	1	0,5
0,89	0	1	0	1	0,5
0,9	0	1	0	1	0,5
0,91	0	1	0	1	0,5
0,92	0	1	0	1	0,5
0,93	0	1	0	1	0,5
0,94	0	1	0	1	0,5
0,95	0	1	0	1	0,5
0,96	0	1	0	1	0,5
0,97	0	1	0	1	0,5
0,98	0	1	0	1	0,5
0,99	0	1	0	1	0,5

Tableau 4-11 Test de détection pour le modèle FSM (scénario 2)

threshold	FP	TN	FP	TN	Accuracy= (TP+TN)/(TP+TN+FP+FN)
0	0,85714286	0,14285714	0,99	0,01	0,56642857
1	0,85714286	0,14285714	0,99	0,01	0,56642857
2	0,85714286	0,14285714	0,99	0,01	0,56642857
3	0,85714286	0,14285714	0,99	0,01	0,56642857
4	0,85714286	0,14285714	0,99	0,01	0,56642857
5	0,85714286	0,14285714	0,99	0,01	0,56642857
6	0,85714286	0,14285714	0,99	0,01	0,56642857
7	0,85714286	0,14285714	0,99	0,01	0,56642857
8	0,85714286	0,14285714	0,99	0,01	0,56642857
9	0,85714286	0,14285714	0,99	0,01	0,56642857
10	0,85714286	0,14285714	0,99	0,01	0,56642857
11	0,85714286	0,14285714	0,99	0,01	0,56642857
12	0,85714286	0,14285714	0,99	0,01	0,56642857
13	0,85714286	0,14285714	0,99	0,01	0,56642857
14	0,85714286	0,14285714	0,99	0,01	0,56642857

15	0,85714286	0,14285714	0,99	0,01	0,56642857
16	0,85714286	0,14285714	0,99	0,01	0,56642857
17	0,85714286	0,14285714	0,99	0,01	0,56642857
18	0,85714286	0,14285714	0,99	0,01	0,56642857
19	0,85714286	0,14285714	0,99	0,01	0,56642857
20	0,85714286	0,14285714	0,99	0,01	0,56642857
21	0,85714286	0,14285714	0,99	0,01	0,56642857
22	0,85714286	0,14285714	0,99	0,01	0,56642857
23	0,85714286	0,14285714	0,99	0,01	0,56642857
24	0,85714286	0,14285714	0,99	0,01	0,56642857
25	0,85714286	0,14285714	0,99	0,01	0,56642857
26	0,85714286	0,14285714	0,99	0,01	0,56642857
27	0,85714286	0,14285714	0,99	0,01	0,56642857
28	0,85714286	0,14285714	0,99	0,01	0,56642857
29	0,85714286	0,14285714	0,99	0,01	0,56642857
30	0,85714286	0,14285714	0,99	0,01	0,56642857
31	0,85714286	0,14285714	0,99	0,01	0,56642857
32	0,85714286	0,14285714	0,99	0,01	0,56642857
33	0,85714286	0,14285714	0,99	0,01	0,56642857
34	0,85714286	0,14285714	0,99	0,01	0,56642857
35	0,85714286	0,14285714	0,99	0,01	0,56642857
36	0,85714286	0,14285714	0,99	0,01	0,56642857
37	0,85714286	0,14285714	0,99	0,01	0,56642857
38	0,85714286	0,14285714	0,99	0,01	0,56642857
39	0,85714286	0,14285714	0,99	0,01	0,56642857
40	0,85714286	0,14285714	0,99	0,01	0,56642857
41	0,85714286	0,14285714	0,99	0,01	0,56642857
42	0,85714286	0,14285714	0,99	0,01	0,56642857
43	0,85714286	0,14285714	0,99	0,01	0,56642857
44	0,85714286	0,14285714	0,99	0,01	0,56642857
45	0,85714286	0,14285714	0,99	0,01	0,56642857
46	0,85714286	0,14285714	0,99	0,01	0,56642857
47	0,85454545	0,14545455	0,99	0,01	0,56772727
48	0,83896104	0,16103896	0,97	0,03	0,56551948
49	0,82337662	0,17662338	0,97	0,03	0,57331169
50	0,82337662	0,17662338	0,97	0,03	0,57331169
51	0,82337662	0,17662338	0,97	0,03	0,57331169
52	**0,21298701**	**0,78701299**	**0,79**	**0,21**	**0,78850649**

53	0,1012987	0,8987013	0,13	0,87	0,51435065
54	0,06753247	0,93246753	0,05	0,95	0,49123377
55	0,06753247	0,93246753	0,05	0,95	0,49123377
56	0,06753247	0,93246753	0,05	0,95	0,49123377
57	0,06753247	0,93246753	0,04	0,96	0,48623377
58	0,06493506	0,93506494	0,02	0,98	0,47753247
59	0,06233766	0,93766234	0	1	0,46883117
60	0,06233766	0,93766234	0	1	0,46883117
61	0,06233766	0,93766234	0	1	0,46883117
62	0,06233766	0,93766234	0	1	0,46883117
63	0,06233766	0,93766234	0	1	0,46883117
64	0,06233766	0,93766234	0	1	0,46883117
65	0,06233766	0,93766234	0	1	0,46883117
66	0,06233766	0,93766234	0	1	0,46883117
67	0,06233766	0,93766234	0	1	0,46883117
68	0,06233766	0,93766234	0	1	0,46883117
69	0,06233766	0,93766234	0	1	0,46883117
70	0,06233766	0,93766234	0	1	0,46883117
71	0,06233766	0,93766234	0	1	0,46883117
72	0,06233766	0,93766234	0	1	0,46883117
73	0,06233766	0,93766234	0	1	0,46883117
74	0,06233766	0,93766234	0	1	0,46883117
75	0,01818182	0,98181818	0	1	0,49090909
76	0	1	0	1	0,5
77	0	1	0	1	0,5
78	0	1	0	1	0,5
79	0	1	0	1	0,5
80	0	1	0	1	0,5
81	0	1	0	1	0,5
82	0	1	0	1	0,5
83	0	1	0	1	0,5
84	0	1	0	1	0,5
85	0	1	0	1	0,5
86	0	1	0	1	0,5
87	0	1	0	1	0,5
88	0	1	0	1	0,5
89	0	1	0	1	0,5
90	0	1	0	1	0,5

91	0	1	0	1	0,5
92	0	1	0	1	0,5
93	0	1	0	1	0,5
94	0	1	0	1	0,5
95	0	1	0	1	0,5
96	0	1	0	1	0,5
97	0	1	0	1	0,5
98	0	1	0	1	0,5
99	0	1	0	1	0,5

ANNEXE III

ARTICLE

Maroua Ben Attia, Chamseddine Talhi, Abdelwahab Hamou-Lhadj, Babak Khosravifar, Vincent Turpaud, and Mario Couture. 2015. On-device anomaly detection for resource-limited systems. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (SAC '15). ACM, New York, NY, USA, 548-554. DOI=10.1145/2695664.2695813 http://doi.acm.org/10.1145/2695664.2695813

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Aafer, Yousra, Wenliang Du et Heng Yin. 2013. « DroidAPIMiner: Mining API-level features for robust malware detection in android ». In *Security and Privacy in Communication Networks*. p. 86-103. Springer.

ABHISHEK, J. 2014. « Android SMS malware hosted on Google Play infects 1.2 Million users ». < http://www.hackleaks.in/2014/02/android-sms-malwarehosted-on-google.htm >.

Alakuijala, Jyrki, et Lode Vandevenne. *Data compression using Zopfli*. Tech. rep. Google Inc., Feb. 2013.□□□: https://zopfli. googlecode. com/files/Data _ compression_using_Zopfli. pdf.

Amos, B., H. Turner et J. White. 2013. « Applying machine learning classifiers to dynamic Android malware detection at scale ». In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*. (1-5 July 2013), p. 1666-1671.

Anderson, James P. 1980. « Computer Security Threat Monitoring and Surveillance ».

android.googlesource.com. 2012. « syscallent.h ». < https://android.googlesource.com/platform/external/strace/+/android-4.1.1_r6.1/linux/arm/syscallent.h >.

apc.io. 2014. « APC 8750 ». < http://apc.io/products/8750a/ >.

Arp, Daniel, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck et CERT Siemens. 2014. « Drebin: Effective and explainable detection of android malware in your pocket ». In *Proc. of NDSS*.

Aung, Zarni, et Win Zaw. 2013. « Permission-based Android malware detection ». *International Journal of Scientific and Technology Research,* vol. 2, n° 3, p. 228-234.

Avenger. 2013. « Uncovering an Android botnet involved in SMS fraud ». < http://update2tech.blogspot.ca/2013/09/uncovering-android-botnet-involved-in.html >.

Ben Attia, M., Talhi,C., Hamou-Lhadj,A., Khosravifar,B., Turpaud,V., & Couture,M. (2015, April). 2015a. « On device anomaly detection for resource-limited systems ». In *30th Annual ACM Symposium on Applied Computing (SAC'2015)*. (Salamanca, Spain, April).

Ben Attia, Maroua. 2015c. « On-device Anomaly Detection for Resource-limited Systems Using Behavioral Analysis ». In *CSER: Consortium of Software Engineering Research*. (Département de Génie Logiciel et Informatique, Polytechnique Montréal, Canada, March). < https://sites.google.com/site/cser2015spring/home/ >.

Bickford, Jeffrey, H Andrés Lagar-Cavilla, Alexander Varshavsky, Vinod Ganapathy et Liviu Iftode. 2011. « Security versus energy tradeoffs in host-based mobile malware detection ». In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. p. 225-238. ACM.

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

Bose, Abhijit, Xin Hu, Kang G Shin et Taejoon Park. 2008. « Behavioral detection of malware on mobile handsets ». In *Proceedings of the 6th international conference on Mobile systems, applications, and services*. p. 225-238. ACM.

Burguera, Iker, Urko Zurutuza et Simin Nadjm-Tehrani. 2011. « Crowdroid: behavior-based malware detection system for Android ». In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. (Chicago, Illinois, USA), p. 15-26. 2046619: ACM.

Carton, Olivier. 2008. *Langages formels, calculabilité et complexit*.

Christodorescu, Mihai, et Somesh Jha. 2003. « Static analysis of executables to detect malicious patterns ». In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. (Washington, DC), p. 12-12. 1251365: USENIX Association.

Crockford, Douglas. 2006 «JSON: The Fat-Free Alternative to XML ». < http://www.json.org/fatfree.html >. Consulté le December.

Danchev, Dancho. 2013. « Fake Chrome/Firefox/Internet Explorer/Safari Updates Expose Users to Android Malware ». < http://ddanchev.blogspot.ca/2013/11/fake-chromefirefoxinternet.html >.

Datta, S. K., C. Bonnet et N. Nikaein. 2012. « Android power management: Current and future trends ». In *Enabling Technologies for Smartphone and Internet of Things (ETSIoT), 2012 First IEEE Workshop on*. (18-18 June 2012), p. 48-53.

Denning, DE. 1987. « Software Engineering for Security: a Roadmap ». p. 1–20.

Desnos, Anthony, et Geoffroy Gueguen. 2011. « Android: From reversing to decompilation ». *Proc. of Black Hat Abu Dhabi*.

Desnoyers, Mathieu, et M Dagenais. 2006. « OS tracing for hardware, driver and binary reverse engineering in Linux ». *CodeBreakers Journal,* vol. 1, n° 2.

developer.android. 2015. « Interprocess Communication ». < http://developer.android.com/guide/components/processes-and-threads.html#IPC >.

diamant2320. 2014. « Malware_set ». < http://pastebin.com/257cCpuP >.

die.net. « strace(1) - Linux man page ». < http://linux.die.net/man/1/strace >.

Dini, Gianluca, Fabio Martinelli, Andrea Saracino et Daniele Sgandurra. 2012. « Madam: a multi-level anomaly detector for android malware ». In *Computer network security*. p. 240-253. Springer.

Dong-Jie, Wu, Mao Ching-Hao, Wei Te-En, Lee Hahn-Ming et Wu Kuo-Ping. 2012. « DroidMat: Android Malware Detection through Manifest and API Calls Tracing ». In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. (9-10 Aug. 2012), p. 62-69.

Enck, William, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel et Anmol N Sheth. 2014. « TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones ». *ACM Transactions on Computer Systems (TOCS),* vol. 32, n° 2, p. 5.

Fedler, Rafael, Julian Schütte et Marcel Kulicke. 2013. « On the effectiveness of malware protection on Android ». *Fraunhofer AISEC April 2013*.

Forrest, S., S. A. Hofmeyr, A. Somayaji et T. A. Longstaff. 1996. « A sense of self for Unix processes ». In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*. (6-8 May 1996), p. 120-128.

Forrest, Stephanie, Steven Hofmeyr et Anil Somayaji. 2008. « The evolution of system-call monitoring ». In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. p. 418-430. IEEE.

FRAMINGHAM, Mass. 2015. « Android and iOS Squeeze the Competition, Swelling to 96.3% of the Smartphone Operating System Market for Both 4Q14 and CY14,

According to IDC ». < http://www.idc.com/getdoc.jsp?containerId=prUS25450615 >. Consulté le February 24, 2015.

Frazier, M. 2008. « The BeagleBoard: $149 Linux System ». < http://www.linuxjournal.com/content/beagleboard-149linux-system >.

Fuchs, Adam P, Avik Chaudhuri et Jeffrey S Foster. 2009. « Scandroid: Automated security certification of android applications ». *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/avik/projects/scandroidascaa,* vol. 2, n° 3.

Gilbert, Peter, Byung-Gon Chun, Landon P. Cox et Jaeyeon Jung. 2011. « Vision: automated security validation of mobile apps at app markets ». In *Proceedings of the second international workshop on Mobile cloud computing and services*. (Bethesda, Maryland, USA), p. 21-26. 1999740: ACM.

Griffin, Kent, Scott Schneider, Xin Hu et Tzi-Cker Chiueh. 2009. « Automatic Generation of String Signatures for Malware Detection ». In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. (Saint-Malo, France), p. 101-120. 1691146: Springer-Verlag.

Gumstix.com. « Overo FE COM ». < https://store.gumstix.com/index.php/products/256/ >.

Guofei, Jiang, Chen Haifeng, C. Ungureanu et K. Yoshihira. 2007. « Multiresolution Abnormal Trace Detection Using Varied-Length n-Grams and Automata ». *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on,* vol. 37, n° 1, p. 86-97.

Hanley, James A, et Barbara J McNeil. 1982. « The meaning and use of the area under a receiver operating characteristic (ROC) curve ». *Radiology,* vol. 143, n° 1, p. 29-36.

Hofmeyr, Steven A, Stephanie Forrest et Anil Somayaji. 1998. « Intrusion detection using sequences of system calls ». *Journal of computer security,* vol. 6, n° 3, p. 151-180.

Huang, Yi-an, Wei Fan, Wenke Lee et Philip S Yu. 2003. « Cross-feature analysis for detecting ad-hoc routing anomalies ». In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*. p. 478-487. IEEE.

Hubballi, N., S. Biswas et S. Nandi. 2011. « Sequencegram: n-gram modeling of system calls for program based anomaly detection ». In *Communication Systems and Networks (COMSNETS), 2011 Third International Conference on*. (4-8 Jan. 2011), p. 1-10.

Husted, Nathaniel, Hassen Saïdi et Ashish Gehani. 2011. « Smartphone security limitations: conflicting traditions ». In *Proceedings of the 2011 Workshop on Governance of Technology, Information, and Policies*. p. 5-12. ACM.

Hyo-Sik, Ham, et Choi Mi-Jung. 2013. « Analysis of Android malware detection performance using machine learning classifiers ». In *ICT Convergence (ICTC), 2013 International Conference on*. (14-16 Oct. 2013), p. 490-495.

Intelligence, VirusTotal. 2013. « VirusTotal Intelligence ».

Isohara, Takamasa, Keisuke Takemori et Ayumu Kubota. 2011. « Kernel-based behavior analysis for android malware detection ». In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*. p. 1011-1015. IEEE.

Jacob, Grégoire, Hervé Debar et Eric Filiol. 2008. « Behavioral detection of malware: from a survey towards an established taxonomy ». *Journal in Computer Virology,* vol. 4, n° 3, p. 251-266.

Jayasinghe, Gaya K., J. Shane Culpepper et Peter Bertok. 2014. « Efficient and effective realtime prediction of drive-by download attacks ». *Journal of Network and Computer Applications,* vol. 38, n° 0, p. 135-149.

Joly, S. 2014. « TBS2910 Mini PC ARM Matrix ».

Kevin Joshua Abela, Jan Raynier Delas Alas, Don Kristopher Angeles, Robert Joseph Tolentino, Miguel Alberto Gomez. 2013. « An Automated Malware Detection System for Android using Behavior-based Analysis AMDA ». *International Journal of Cyber-Security and Digital Forensics (IJCSDF),* vol. 2, n° 2, p. 1-11.

Kim, Hahnsang, Joshua Smith et Kang G. Shin. 2008. « Detecting energy-greedy anomalies and mobile malware variants ». In *Proceedings of the 6th international conference on Mobile systems, applications, and services*. (Breckenridge, CO, USA), p. 239-252. 1378627: ACM.

Liu, Lei, Guanhua Yan, Xinwen Zhang et Songqing Chen. 2009. « VirusMeter: Preventing Your Cellphone from Spies ». In *Recent Advances in Intrusion Detection*, sous la dir. de Kirda, Engin, Somesh Jha et Davide Balzarotti. Vol. 5758, p. 244-264. Coll. « Lecture Notes in Computer Science »: Springer Berlin Heidelberg. < http://dx.doi.org/10.1007/978-3-642-04342-0_13 >.

Ma, Xiao, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul et Geoffrey M. Voelker. 2013. « eDoctor: automatically diagnosing abnormal battery drain issues on smartphones ». In *Proceedings of the 10th USENIX*

conference on Networked Systems Design and Implementation. (Lombard, IL), p. 57-70. 2482634: USENIX Association.

Maier, Dominik, Tilo Muller et Mykola Protsenko. 2014. « Divide-and-Conquer: Why Android Malware Cannot Be Stopped ». In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*. p. 30-39. IEEE.

Manekari, Rahul. 2013. « Android is Open to Attacks According to a Survey From NQ Mobile (2013) ».

manpagesfr. 2008. « PTRACE ». < http://manpagesfr.free.fr/man/man2/ptrace.2.html >.

Mas' ud, Mohd Zaki, Shahrin Sahib, Mohd Faizal Abdollah, Siti Rahayu Selamat et Robiah Yusof. 2014. « Analysis of features selection and machine learning classifier in android malware detection ». In *Information Science and Applications (ICISA), 2014 International Conference on*. p. 1-5. IEEE.

Michael, Christoph C, et Anup Ghosh. 2002. « Simple, state-based approaches to program-based anomaly detection ». *ACM Transactions on Information and System Security (TISSEC),* vol. 5, n° 3, p. 203-237.

Millard, E. 2004. « Cabir: World's First Wireless Worm ». < http://www.technewsworld.com/story/34542.html >.

Min, Zheng, Sun Mingshen et J. C. S. Lui. 2014. « DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability ». In *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*. (4-8 Aug. 2014), p. 128-133.

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

Moskovitch, Robert, Yuval Elovici et Lior Rokach. 2008. « Detection of unknown computer worms based on behavioral classification of the host ». *Comput. Stat. Data Anal.,* vol. 52, n° 9, p. 4544-4566.

Ng, Gary. 2014. « The 16GB Samsung Galaxy S5 Has Less Than 8GB of Usable Storage ». < http://www.iphoneincanada.ca/news/galaxy-s5-8gbusable-storage >.

Perrucci, Gian Paolo, Frank HP Fitzek et Jörg Widmer. 2011. « Survey on energy consumption entities on the smartphone platform ». In *Vehicular Technology Conference (VTC Spring), 2011 IEEE 73rd*. p. 1-6. IEEE.

Poeplau, Sebastian, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel et Giovanni Vigna. 2014a. « Execute this! analyzing unsafe and malicious dynamic code loading in android applications ». In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*.

Poeplau, Sebastian, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel et Giovanni Vigna. 2014b. « Execute this! analyzing unsafe and malicious dynamic code loading in android applications ». *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*.

Rabek, Jesse C., Roger I. Khazan, Scott M. Lewandowski et Robert K. Cunningham. 2003. « Detection of injected, dynamically generated, and obfuscated malicious code ». In *Proceedings of the 2003 ACM workshop on Rapid malcode*. (Washington, DC, USA), p. 76-82. 948201: ACM.

Rafiqul Islam, Md, Md Saiful Islam et Morshed U. Chowdhury. 2011. « Detecting Unknown Anomalous Program Behavior Using API System Calls ». In *Informatics Engineering and Information Science*, sous la dir. de Abd Manaf, Azizah, Shamsul Sahibuddin, Rabiah Ahmad, Salwani Mohd Daud et Eyas El-Qawasmeh. Vol. 254, p. 383-394. Coll. « Communications in Computer and Information Science »: Springer Berlin Heidelberg. < http://dx.doi.org/10.1007/978-3-642-25483-3_31 >.

Richter, Felix. 2015. « Is Android Becoming the New Windows? ». < http://www.statista.com/chart/1761/connected-device-shipment-forecast/ >. Consulté le January 6th, 2015.

Sabahi, F, et A Movaghar. 2008. « Intrusion detection: A survey ». In *Systems and Networks Communications, 2008. ICSNC'08. 3rd International Conference on*. p. 23-26. IEEE.

Sahs, J., et L. Khan. 2012. « A Machine Learning Approach to Android Malware Detection ». In *Intelligence and Security Informatics Conference (EISIC), 2012 European*. (22-24 Aug. 2012), p. 141-147.

Salman, Alaa, Imad H Elhajj, Ali Chehab et Ayman Kayssi. 2014. « DAIDS: An Architecture for Modular Mobile IDS ». In *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. p. 328-333. IEEE.

Sanz, Borja, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, PabloGarcia Bringas et Gonzalo Álvarez. 2013a. « PUMA: Permission Usage to Detect Malware in Android ». In *International Joint Conference CISIS'12-ICEUTE´12-SOCO´12 Special Sessions*, sous la dir. de Herrero, Álvaro, Václav Snášel, Ajith Abraham, Ivan

Zelinka, Bruno Baruque, Héctor Quintián, José Luis Calvo, Javier Sedano et Emilio Corchado. Vol. 189, p. 289-298. Coll. « Advances in Intelligent Systems and Computing »: Springer Berlin Heidelberg. < http://dx.doi.org/10.1007/978-3-642-33018-6_30 >.

Sanz, Borja, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Javier Nieves, Pablo G. Bringas, Gonzalo, #xC1, lvarez Mara, #xF1 et #xF3. 2013b. « MAMA: MANIFEST ANALYSIS FOR MALWARE DETECTION IN ANDROID ». *Cybern. Syst.,* vol. 44, n° 6-7, p. 469-488.

Schmidt, A. D., R. Bye, H. G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe et S. Albayrak. 2009. « Static Analysis of Executables for Collaborative Malware Detection on Android ». In *Communications, 2009. ICC '09. IEEE International Conference on*. (14-18 June 2009), p. 1-5.

sectechno. 2014. « Ads on Android Claims Fake Update That Install Unwanted Application ». < http://www.sectechno.com/ads-on-android-claims-fake-update-that-install-unwanted-application/ >.

Shabtai, Asaf, Uri Kanonov, Yuval Elovici, Chanan Glezer et Yael Weiss. 2012. « "Andromaly": a behavioral malware detection framework for android devices ». *Journal of Intelligent Information Systems,* vol. 38, n° 1, p. 161-190.

Shabtai, Asaf, Robert Moskovitch, Yuval Elovici et Chanan Glezer. 2009. « Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey ». *Inf. Secur. Tech. Rep.,* vol. 14, n° 1, p. 16-29.

Shabtai, Asaf, Lena Tenenboim-Chekina, Dudu Mimran, Lior Rokach, Bracha Shapira et Yuval Elovici. 2014. « Mobile malware detection through analysis of deviations in application network behavior ». *Computers & Security,* vol. 43, p. 1-18.

Sheen, Shina, R. Anitha et V. Natarajan. 2015. « Android based malware detection using a multifeature collaborative decision fusion approach ». *Neurocomputing,* vol. 151, Part 2, n° 0, p. 905-912.

Somayaji, Anil, et Stephanie Forrest. 2000. « Automated Response Using System-Call Delay ». In *Usenix Security Symposium*. p. 185-197.

Sophos. 2014. « Mobile Security Threat Report 2014 ». < http://www.sophos.com/enus/medialibrary/PDFs/other/sophos-mobile-securitythreat-report.pdf >.

Srinivasan, Raj. 1995. « RPC: Remote procedure call protocol specification version 2 ».

Toupin, D. 2011. « Using Tracing to Diagnose or Monitor Systems ». *Software, IEEE,* vol. 28, n° 1, p. 87-91.

Viennot, Nicolas, Edward Garcia et Jason Nieh. 2014. « A measurement study of google play ». *SIGMETRICS Perform. Eval. Rev.,* vol. 42, n° 1, p. 221-233.

Wang, Chao, Zhizhong Wu, Aili Wang, Xi Li, Feng Yang et Xuehai Zhou. 2013. « SmartMal: a service-oriented behavioral malware detection framework for smartphones ». In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. p. 329-336. IEEE.

webappers. 2008. « Android- The First Complete, Open, and Free Mobile Platform ». < http://www.webappers.com/2008/08/23/android-the-first-complete-open-and-free-mobile-platform/ >.

Xie, Liang, Xinwen Zhang, Jean-Pierre Seifert et Sencun Zhu. 2010. « pBMDS: a behavior-based malware detection system for cellphone devices ». In *Proceedings of the third ACM conference on Wireless network security*. (Hoboken, New Jersey, USA), p. 37-48. 1741874: ACM.

Yuan, Fangfang, Lidong Zhai, Yanan Cao et Li Guo. 2013. « Research of Intrusion Detection System on Android ». In *Services (SERVICES), 2013 IEEE Ninth World Congress on*. p. 312-316. IEEE.

Zhao, Min, Fangbin Ge, Tao Zhang et Zhijian Yuan. 2011. « Antimaldroid: An efficient svm-based malware detection framework for android ». In *Information Computing and Applications*. p. 158-166. Springer.