Une approche sémantique de détection de maliciel Android basée sur la vérification de modèles et l'apprentissage automatique

Mémoire

Souad El Hatib

Maîtrise en informatique - avec mémoire
Maître ès sciences (M. Sc.)

Québec, Canada

Une approche sémantique de détection de maliciel Android basée sur la vérification de modèles et l'apprentissage automatique

Mémoire

Souad El Hatib

Sous la direction de:

Nadia Tawbi, directrice de recherche
Josée Desharnais, codirectrice de recherche

Résumé

Le nombre croissant de logiciels malveillants Android s'accompagne d'une préoccupation profonde liée aux problèmes de la sécurité des terminaux mobiles. Les enjeux deviennent sans conteste de plus en plus importants, suscitant ainsi beaucoup d'attention de la part de la communauté des chercheurs. En outre, la prolifération des logiciels malveillants va de pair avec la sophistication et la complexité de ces derniers. En effet, les logiciels malveillants plus élaborés, tels que les maliciels polymorphes et métamorphiques, utilisent des techniques d'obscurcissement du code pour créer de nouvelles variantes qui préservent la sémantique du code original tout en modifiant sa syntaxe, échappant ainsi aux méthodes de détection usuelles.

L'ambition de notre recherche est la proposition d'une approche utilisant les méthodes formelles et l'apprentissage automatique pour la détection des maliciels sur la plateforme Android. L'approche adoptée combine l'analyse statique et l'apprentissage automatique. En effet, à partir des applications Android en format APK, nous visons l'extraction d'un modèle décrivant de manière non ambiguë le comportement de ces dernières. Le langage de spécification formelle choisi est LNT. En se basant sur le modèle généré, les comportements malicieux exprimés en logique temporelle sont vérifiés à l'aide d'un vérificateur de modèle. Ces propriétés temporelles sont utilisées comme caractéristiques par un algorithme d'apprentissage automatique pour classifier les applications Android.

Abstract

The ever-increasing number of Android malware is accompanied by a deep concern about security issues in the mobile ecosystem. Unquestionably, Android malware detection has received much attention in the research community and therefore it becomes a crucial aspect of software security. Actually, malware proliferation goes hand in hand with the sophistication and complexity of malware. To illustrate, more elaborated malware like polymorphic and metamorphic malware, make use of code obfuscation techniques to build new variants that preserve the semantics of the original code but modify it's syntax and thus escape the usual detection methods. In the present work, we propose a model-checking based approach that combines static analysis and machine learning. Mainly, from a given Android application we extract an abstract model expressed in terms of LNT, a process algebra language. Afterwards, security related Android behaviours specified by temporal logic formulas are checked against this model, the satisfaction of a specific formula is considered as a feature, finally machine learning algorithms are used to classify the application as malicious or not.

Table des matières

Liste des tableaux

Liste des figures

Liste d'abréviations

APK Android Package Kit.

ART Android Runtime.

AUC Area Under The Curve.

BNS Binormal Separation.

CADP Construction and Analysis of Distributed Processes.

CCS Calculus for Communicating Systems.

CFG Control Flow Graph.

CTL Computation Tree Logic.

CTPL Computation Tree Predicate Logic.

DCL Dynamic code loading.

DNS Domain Name System.

DSA Deterministic Symbolic Automaton.

FNR False Negative Rate.

FPR False Positive Rate.

GB Gradient Boosting.

HAL Hardware Abstraction Layer.

HML Hennessy Milner Logic.

ICFG Interprocedural Control Flow Graph.

JNI Java Native Interface.

KNN K-Nearest Neighbors.

LNT LOTOS New Technology.

LTL Linear Temporal Logic.

LTS Labelled Transition Systems.

NDK Native Development Kit.

OBA Object Based Actions.

PDS Pushdown System.

PPV Positive Predictive Value.

RF Random Forest.

ROC Receiver Operating Characteristic.

SOS Structural Operational Semantics.

SVM Support Vector Machine.

TAN Transaction Authentication Number.

TNR True Negative Rate.

TPR True Positive Rate.

URL Uniform Resource Locator.

À la mémoire de ma douce
grand-mère.

Remerciements

La réalisation de mon projet de maîtrise a été rendue possible grâce à la contribution de plusieurs personnes. À cet effet, j'aimerais saisir cette occasion pour remercier toute personne ayant contribuer de près ou de loin au bon déroulement de ce projet.

Je tiens tout d'abord à remercier mes directrices de recherche Nadia Tawbi et Josée Desharnais de m'avoir accueillie au sein de leur équipe. Je leur suis très reconnaissante de leur temps, leur patience et leur accompagnement technique, pédagogique et moral. Ce fût un plaisir de travailler sous votre direction !

Je remercie mes collègues au laboratoire, Andrew Bedford et Loïc Ricaud, pour leurs encouragements, leur sympathie et leurs conseils avisés.

Je tiens également à remercier mes amies pour leur présence, leurs encouragements et leur soutien. Votre joie de vivre et la bonne humeur que vous dégagez m'ont été d'une aide précieuse.

Finalement, j'exprime ma gratitude envers les membres de ma famille pour leur amour inconditionnel et leur soutien indéfectible.

Introduction

L'industrie des maliciels (en anglais « *malwares* ») mobiles continue de s'étendre. En effet, l'utilisation des smartphones ainsi que le nombre de services et d'applications croissent de façon fulgurante. Ces appareils concentrent une partie de nos activités sur Internet et manipulent des données sensibles pouvant être exploitées par des pirates pour mener des attaques de grande envergure.

Android se révèle comme une cible de choix pour les pirates. La hausse permanente du nombre d'attaques visant le système d'exploitation de Google peut être expliqué en partie par l'ouverture de ce système ainsi que sa domination sur le marché des smartphones avec 87% de part de marché pour l'an 2019 [10]. De surcroît, la sophistication et la complexité des maliciels augmente en parallèle avec leur nombre. Selon le rapport de McAfee, l'année 2018 a été marquée par une croissance rapide du nombre de menaces qui planent sur les appareils mobiles et les objets connectés [3]. Face aux menaces persistantes, il devient primordial de se doter de capacités de détection plus précises et plus rapides.

Plusieurs approches ont été proposées pour analyser et détecter les maliciels, parmi ces dernières on cite l'analyse statique et l'analyse dynamique.

L'analyse dynamique consiste à exécuter un programme et observer son comportement dans le but de détecter la présence d'un comportement malicieux. Certes, cette approche nous permet d'obtenir des résultats précis concernant quelques scénarios d'exécution du programme, car aucune approximation ou abstraction n'a lieu. Cependant, elle ne permet pas de garantir qu'un programme donné satisfait une propriété vu qu'on raisonne sur une ou plusieurs exécutions concrètes, et non sur toutes les exécutions possibles [32].

L'analyse statique, quant à elle, se fait sans exécution préalable du programme. En se basant sur le code on considère toutes les exécutions possibles, ce qui implique des calculs complexes mais sans impact sur l'exécution. Toutefois, il y a une marge d'imprécision due au problème d'indécidabilité vu qu'on ne peut pas prédire tous les scénarios et de ce fait, on a recours aux approximations. Bien que l'analyse statique a une meilleure couverture des chemins d'éxécutions en comparaison avec l'analyse dynamique, elle demeure vulnérable face aux techniques d'obscurcissement de code [43].

Par ailleurs, une approche hybride peut être mise en œuvre afin de bénéficier du caractère conservatif de l'approche statique ainsi que la précision et l'efficacité de l'analyse dynamique.

Problématique

Pour parer aux attaques des maliciels, plusieurs approches ont été proposées. Ces dernières reposent sur une panoplie de techniques, qui peuvent être catégorisées comme suit : apprentissage automatique, analyse de flots de données et de flots de contrôle, similitude du code, et recherche de signatures dans le code.

La détection basée sur les signatures fait partie des techniques classiques de détection de logiciels malveillants. Cependant, malgré sa précision, cette technique est surpassée par le grand nombre d'applications soumises dans les plateformes de téléchargements, notamment Google Play ou alternatifs. Selon Statista [51], environ 1666 applications sont ajoutées dans Google Play quotidiennement. En effet, l'efficacité des techniques de détection basées sur les signature repose sur une mise à jour régulière des bases de données de signatures. Étant donné que chaque signature identifie un logiciel malveillant spécifique et ne permet pas de détecter les variantes de ce dernier, il est facile de contourner ces techniques de détection par des techniques d'obscurcissement de code. En pratique, il s'avère très difficile de maintenir une mise à jour régulière de la base de données de signatures, qui va de pair avec l'évolution constante et rapide des maliciels [20].

Compte tenu de la similitude des variantes d'un maliciel, plusieurs approches se sont penchées sur l'analyse de cette similarité afin de repérer des fragments de code malicieux. Dans [26], la similarité a été calculée par des équivalences (bisimulations en logique modale) de modèles exprimés en termes d'algèbre de processus CCS (de l'anglais « *Calculus for Communicating Systems* »).

Une autre approche repose sur l'analyse des flux de données. En effet, des travaux de recherche tels que [15, 27, 59] proposent des outils d'analyse statiques/dynamiques de flux de données pour la détection des logiciels malveillants sur Android. À titre d'exemple, Flowdroid [27] et vetdroid [59] utilisent des techniques d'analyse dynamique basées sur le « *data-tainting* » pour suivre les flux de données tout au long du programme. Clairement, ces approches permettent une détection efficace des fuites de données, mais elles ne détectent pas des comportements malveillants plus élaborés. En outre, bien que prometteuses, les techniques d'analyse dynamique s'avèrent gourmandes en temps, tandis que les approches d'analyse statique reposent sur des approximations.

Par ailleurs, les techniques d'apprentissage automatique sont de plus en plus utilisées pour détecter les maliciels sur Android [14, 21, 45, 58]. Toutefois, la plupart de ces classifieurs n'arrivent pas à détecter les variantes d'un maliciel puisqu'ils se basent sur des caractéristiques

qui ne capturent pas la sémantique du comportement du code malveillant, tels les n-grammes des appels API et les permissions. Bien que la majorité des outils utilisant l'apprentissage automatique atteignent un degré élevé de précision lors de la phase d'entraînement, en réalité ces derniers ne parviennent à détecter que peu de maliciels. Cette chute dramatique de performance et d'efficacité peut être expliquée en partie par le sur-ajustement (en anglais « *overfitting* ») et la non pertinence des caractéristiques [12]. En effet, la sélection des caractéristiques pour la classification joue un rôle très important dans l'amélioration de la précision et de la qualité des prédictions.

Comme énoncé ci-dessus, plusieurs techniques et approches ont été mises en place afin de comprendre, d'analyser, de détecter et de classifier les maliciels Android, cependant le paysage de la menace est en constante effervescence, et ainsi Android est toujours confronté à de nombreux défis et problèmes de sécurité.

La compréhension de la sémantique des maliciels Android permet d'extraire l'essence du code malicieux, permettant ainsi d'illustrer les différences fondamentales entre un maliciel et une application bénigne (une application non malicieuse). Sommes-nous en mesure de bien comprendre et de bien cerner l'essence d'un comportement malicieux ?

L'utilisateur a désormais accès à une panoplie d'applications Android à travers le Google Play Store ainsi que d'autres alternatives, un grand pourcentage de ces applications contient du code malicieux ainsi, devient-il primordial de concevoir des outils flexibles, évolutifs permettant de vérifier rapidement les applications. Nous est-il possible d'obtenir le meilleur compromis entre précision et rapidité ?

Certes, l'analyse statique est une approche prometteuse toutefois, elle ne peut être qu'approximative vu le manque des informations purement dynamiques. Par conséquent dans les cas incertains, on adopte plutôt les scénarios pessimistes, vu qu'il est plus sécuritaire de considérer une application bénigne comme maliciel que l'inverse. Saurons-nous mettre en œuvre des approximations nous permettant de réduire le plus possible le nombre de faux positifs occasionnés par l'analyse statique ?

Objectifs

L'ambition de notre recherche est la proposition d'une approche utilisant les méthodes formelles et l'apprentissage automatique pour la détection des maliciels sur la plateforme Android. Dans le cadre de cette maîtrise, nous nous intéressons particulièrement à :

— Construire un modèle sémantique concis et précis permettant de distinguer les différences majeures entre les maliciels et les applications bénignes. Le niveau de détail du modèle doit être convenable de tel sorte que le modèle puisse capturer une partie essentielle du comportement de l'application qui pourrait être représentative dans le cadre de cette

étude.

— Extraire des comportements malicieux. En général, le comportement malicieux est dissimulé dans une application bénigne. Cela tient au fait que la plupart des maliciels sont des versions modifiées des applications bénignes (appelées en anglais « *repackaged applications* ») ainsi une analyse rigoureuse d'un bon nombre de maliciels s'avère nécessaire pour la compréhension et l'extraction des comportements malicieux.

— Prouver la pertinence des caractéristiques sur lesquelles s'appuie la détection, et ce en s'assurant que nos critères de détection permettent de différencier les applications bénignes de celles exhibant des comportements suspects.

Méthodologie

Dans le présent travail, nous proposons une approche basée sur la vérification de modèles qui combine l'analyse statique et l'apprentissage automatique. La figure A résume les grandes étapes de notre approche. Afin de vérifier formellement les applications Android, il importe de les modéliser. Pour ce faire, une spécification en terme de processus LNT est extraite à partir du format APK (pour « *Android Package Kit* ») des applications Android. Ce format est utilisé par le système d'exploitation Android pour la distribution et l'installation d'applications mobiles. En se basant sur le modèle généré, les comportements malicieux représentés par un formalisme logique vont être par la suite vérifiés par un vérificateur de modèles, notamment l'outil Evaluator4 de la boite à outils CADP (de l'anglais « *Construction and Analysis of Distributed Processes* ») [29].

Afin de souligner les mécanismes utilisés par les auteurs des maliciels Android, il convient d'analyser le code des applications malicieuses pour comprendre le fonctionnement de ces derniers ainsi que les différents éléments qui les déclenchent. Une fois définis, ces comportements seront spécifiés en μ-calcul, une extension de la logique modale de Henessy-Milner avec des opérateurs de points fixes. La pertinence des propriétés sera déterminée à l'aide des algorithmes d'apprentissage automatique.

Pour une classification binaire des applications Android, les algorithmes d'apprentissage automatique ont été utilisées. En effet, en se basant sur la présence des appels aux méthodes de l'API Android protégées par des permissions ainsi que la satisfaction des formules du μ-calcul modal, les algorithmes d'apprentissage automatique se prononcent sur le caractère malicieux d'une application Android donnée.

Plan du mémoire

Le présent document s'organise en quatre chapitres. Dans le premier chapitre, nous présentons les éléments essentiels à la compréhension de la problématique de la détection des maliciels

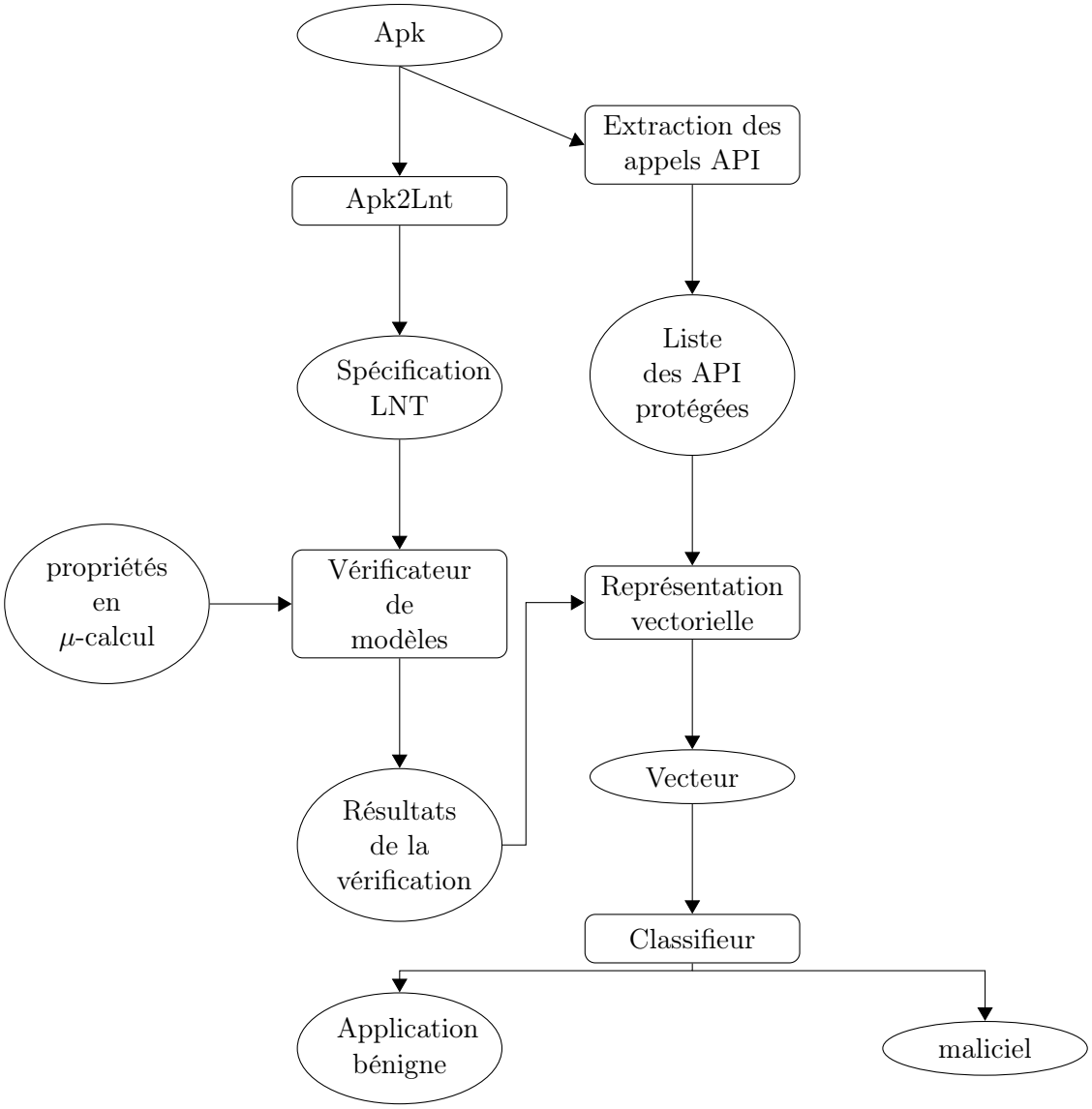

FIGURE A – Le schéma général du processus de détection.

sur la plateforme Android. En effet, à travers ce chapitre nous définissons l'architecture du système Android, les composantes des applications Android ainsi que les différents vecteurs d'attaque des maliciels Android. Par ailleurs, ce chapitre constitue un survol de l'état de l'art portant sur les différentes techniques existantes d'analyse des applications Android.

Le deuxième chapitre est consacré à la présentation de quelques notions de base concernant la vérification de modèles. Une description du processus de la vérification de modèle est fournie ainsi que la définition des langages de spécifications utilisés.

Les chapitres restants sont consacrés à l'explication de notre approche, soit l'application de la vérification de modèles et de l'apprentissage automatique pour la détection de maliciels An-

droid. Dans le troisième chapitre nous décrivons le processus de modélisation des applications Android. De même, nous y présentons les caractéristiques sur lesquelles les algorithmes de classification s'appuient pour détecter les maliciels Android. Finalement, dans le quatrième chapitre, nous présentons les différents modèles d'apprentissage supervisé étudiés, ainsi que les résultats des expérimentations d'apprentissage automatique effectuées.

Chapitre 1

État de l'art

Afin d'analyser et de détecter les maliciels sur la plateforme Android, il convient avant tout de comprendre le fonctionnement des applications Android et d'étudier les différentes techniques de détection existantes. C'est dans cette optique que s'inscrit ce chapitre visant la compréhension de l'architecture du système Android ainsi que les différentes composantes des applications. De même, ce chapitre s'intéresse aux vecteurs d'attaques déployés par les auteurs de logiciels malveillants afin d'infecter les terminaux mobiles et recense les différentes techniques d'analyse des applications Android.

1.1 Le système Android

Android est une plateforme logicielle ouverte (en anglais « *open source* ») destinée à un large éventail d'appareils électroniques. La figure 1.1 illustre l'architecture du système Android. Ce dernier se décline comme une pile logicielle comportant plusieurs couches notamment la couche Kernel (noyau Android), la couche d'abstraction matérielle, un environnement d'exécution, les librairies C/C++ , *Java API Framework* et la couche applications.

Noyau Android : Cette composante de base permet de gérer les ressources matérielles des appareils électronique tout en assurant un fonctionnement harmonieux des composants logiciels et matérielles. Android se base sur une version modifiée du noyau Linux. Plusieurs correctifs ont été apportés y compris *wakelocks* qui est un module de gestion d'alimentation, *Binder* un mécanisme de communication entre processus, *Ashmem* un mécanisme de partage de mémoire, etc. Ces modifications ont été apportées afin de répondre à des besoins spécifiques en matières de gestion d'alimentation, de gestion de mémoire ainsi que de gestion d'exécution. Ces besoins dérivent de la nature des appareils sur lesquels Android est supposé tourner, caractérisée par des ressources limitées.

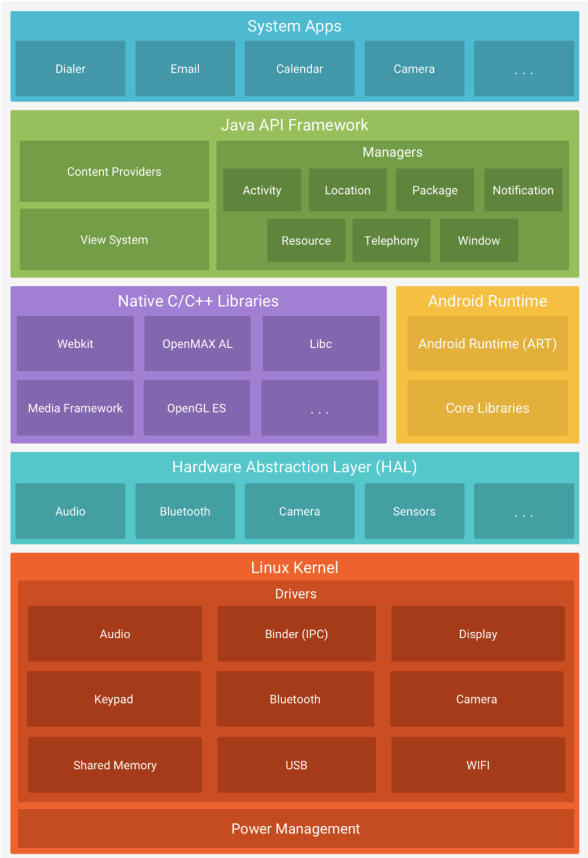

FIGURE 1.1 – L'architecture d'Android, figure extraite de [9]

La couche d'abstraction matérielle ou HAL (de l'anglais « *Hardware Abstraction Layer* ») : Cette couche constitue une interface standard qui met les ressources physiques et matérielles à la disposition de la couche supérieure *Java API Framework*, et ce à travers des fonctions standards de manipulation du matériel informatique. Cette couche regroupe plusieurs modules de librairies implémentant les interfaces des périphériques de l'appareil tels que la caméra, les capteurs et la connexion bluethooth.

Android Runtime : Cette couche représente l'environnement d'exécution des applications Android. Depuis Android 5, version d'Android baptisée Lollipop, la machine virtuelle *Dalvik* a été remplacée officiellement par l'environnement d'exécution ART en guise d'amélioration des performances et d'autonomie tout en préservant une rétrocompatibilité ascendante. ART a été conçu pour exécuter plusieurs machines virtuelles sur des périphériques à faible mémoire. En effet, grâce à la compilation anticipée (en anglais « *ahead of time compilation* »), ART permet de compiler l'application lors de l'installation évitant ainsi la recompilation de l'application à chaque utilisation.

Les librairies C/C++ : Android dispose d'un ensemble de bibliothèques natives écrites en C/C++, parmi ses librairies on peut citer : *libc* une bibliothèque C standard, *Webkit* un moteur de rendu libre et *Sqlite* un moteur de base de données relationnelle.

Java API Framework : Le cadriciel (en anglais « *framework* ») regroupe un ensemble des APIs (interfaces de programmation) écrites en Java. Cette couche fournit un ensemble de services conçus pour simplifier la réutilisation des composants des applications Android. Parmi les services offerts par le cadriciel on trouve *Activity Manager*, qui contrôle la pile des activités et gère le cycle de vie des applications Android.

Applications : Cette couche représente l'ensemble des applications fournies par Android. En effet, les téléphones sous Android sont dotés d'un ensemble riche d'applications par défaut (ex : applications de messagerie électronique, calendriers, applications de messagerie SMS, applications de gestion de contacts, navigateur web,etc.)

1.2 Les applications Android

Bien qu'elles soient implémentées en Java, les applications Android ne possèdent pas un point d'entrée unique. En effet, à l'opposé des applications Java, les applications Android ne disposent pas d'une fonction principale mais elles sont conçues en termes de composantes. Ces dernières, sont listées dans le fichier *manifest* de l'application. Notamment, une application Android est constituée de quatre composantes :

Les activités : Une activité permet de gérer l'interaction d'un écran. Elle permet à l'utilisateur d'interagir avec l'application. Précisément, une application possède une ou plusieurs activités permettant d'assurer l'interactivité. Cependant, une application ne peut lancer qu'une activité à la fois. Les activités sont gérées en pile et possèdent essentiellement quatre états.

- L'état « active » (« *Running* ») : Lorsque l'activité s'exécute au premier plan, autrement dit lorsqu'elle occupe le sommet de la pile des activités.
- L'état « pause » (« *Paused* ») : Quand l'activité n'est plus au premier plan. L'activité demeure visible et reste attachée au gestionnaire de fenêtres. Toutefois, en cas de manque de mémoire, le système peut détruire l'activité.
- L'état « stoppée » (« *Stopped* ») : Dans cet état, l'activité existe toujours en mémoire mais elle est invisible à l'utilisateur. Si nécessaire, les activités stoppées peuvent être détruite afin de libérer les ressources pour les activités actives.
- L'état « détruite » (« *Destroyed* ») : Les activités détruites sont supprimées de la pile des activités et doivent être redémarrées avant de pouvoir être affichées.

La figure 1.2 illustre le cycle de vie d'une activité Android.

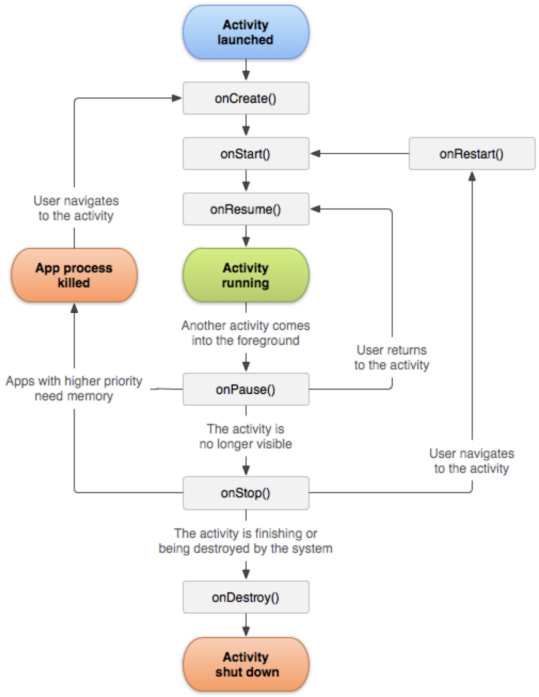

FIGURE 1.2 – Le cycle de vie d'une activité Android, extraite de [8]

Les services : Un service est une composante qui s'exécute en arrière-plan. Il permet l'exécution des tâches ne nécessitant pas l'interaction avec l'utilisateur, à titre d'exemple écouter de la musique en tâche de fond.

Les fournisseurs de contenu (en anglais « *content providers* ») : Les fournisseurs de contenu permettent de partager les données entre différentes applications. Ils offrent une interface standard permettant aux applications de manipuler les données. L'accès aux données est protégé par des permissions. En effet, dans le but de contrôler l'accès aux données, juste les applications possédant les permissions adéquates peuvent y accéder.

Les récepteurs d'événement (en anglais « *Broadcast receivers* ») : Il s'agit d'une composante qui reste à l'écoute des *intents*, un mécanisme de communication entre applications. Elle permet aux applications de recevoir les *intents* diffusées par le système ou par d'autres applications. Précisément, elle permet aux applications de s'inscrire à des événements système ou d'application. Toutes les applications enregistrées pour l'événement vont être averties une fois que cet événement se produit.

1.3 Les maliciels Android

Un maliciel est un logiciel à visée malveillante pouvant compromettre la sécurité d'un système informatique. Les actions effectuées par les maliciels ainsi que les systèmes visés par ces derniers sont multiples et diversifiés.

L'ouverture et la versatilité des terminaux mobiles ont toujours captivé l'attention des attaquants. Les maliciels peuvent être automatiquement téléchargés et cachés dans des applications non malveillantes sans que l'utilisateur ne s'en rende compte, et ainsi, de manière illégale le maliciel aura accès aux données personnelles. La propagation des maliciels entre les terminaux mobiles se fait de plusieurs manières. Dans cette section, nous présentons quelques vecteurs d'attaques sur Android.

1.3.1 L'ingénierie sociale

L'ingénierie sociale se base sur l'exploitation des failles humaines, l'attaquant abusant de la confiance et la crédulité d'une personne afin de récupérer des informations qui vont l'aider à bien mener son attaque par la suite. Pour les plateformes mobiles ce type d'attaque prend plusieurs formes, notamment l'envoi des SMS avec des pièces jointes ou encore la création des applications ou des jeux captivants qui une fois installés, envoient des informations sans la permission de l'utilisateur. De nombreuses techniques d'ingénierie sociale ont été déployées dans les plateformes mobiles afin de faire proliférer les maliciels. Selon Zhou et Jiang [33], le *repackaging*, l'attaque par mise à jour (en anglais « *update attack* ») et l'attaque par téléchargement furtif (en anglais « *drive-by download* ») constituent ces principales techniques.

L'hameçonnage

Cette technique consiste à faire croire à la victime qu'elle s'adresse à la bonne entité par exemple un site web d'une banque, en créant des sites web truqués avec une apparence similaire, et ce dans le but de rassembler des renseignements personnels pour usurper l'identité de la victime.

Le *repackaging*

Dans ce type d'attaque, en se basant sur la rétro-ingénierie, les concepteurs de maliciels modifient des applications populaires et ce en rajoutant du code malicieux. Finalement, ces versions modifiées d'applications légitimes, vont être soumises sur des plateformes de téléchargements d'applications mobiles, notamment Google Play ou ses alternatifs. Afin de duper les utilisateurs et les inciter à télécharger les applications infectées, les auteurs des applications malveillantes utilisent des noms de classes légitimes. À titre d'exemple, dans des variantes de DroidKungFu [4], la charge utile (en anglais « *payload* ») se trouve dans une classe appelée com.google.UpdateService pour apparaître comme une mise à jour officielle de Google.

L'attaque par mise à jour

Cette technique rend la tâche de détection plus difficile. Essentiellement, au lieu de livrer des applications contenant des fragments du code malicieux, ces applications sont plutôt dotées de composantes de mise à jour qui par la suite vont télécharger le code malicieux au moment de l'exécution. Le maliciel Plankton [6] met à jour furtivement certaines composantes de l'application en téléchargeant d'un serveur distant, un fichier « .jar » contenant le code malicieux et ce sans envoyer de notification préalable à l'utilisateur.

L'attaque par téléchargement furtif

Dans ce type d'attaque, on incite l'utilisateur à télécharger des applications malicieuses. Le maliciel Jifake [5] est téléchargé lorsque l'utilisateur est redirigé vers une page web infectée. De même, le maliciel Cgtracker, s'appuie sur la publicité afin de rediriger l'utilisateur vers une fausse plateforme de téléchargement pour l'inciter à télécharger une application qui est censée optimiser l'utilisation de la batterie. En fait, cette application renferme un maliciel qui souscrit l'utilisateur à un service surtaxé.

1.3.2 L'attaque par *botnet*

Un Android *botnet*, est un réseau constitué de plusieurs smartphones compromis (sous Android) contrôlés par un maître, le *botmaster*, à travers un réseau de commandement et de control (en anglais « C&C *network* »). La première génération des Android botnet est arrivée sous la forme d'un cheval de troie SMS, le maliciel est dissimulé dans une application légitime et tourne d'une manière discrète sans que l'utilisateur ne s'en rende compte.
L'Eurograbber, une variante plus élaborée du maliciel Zitmo découverte pour la première fois en 2012, aurait volé plus de 36 millions d'euros auprès de 30000 clients de plusieurs banques en Europe [34]. Notamment, Eurograbber incite les victimes à cliquer sur une adresse URL illicite afin d'infecter leur ordinateur par un cheval de troie. Une fois infecté, la connexion des victimes aux comptes bancaires sera surveillée et le maliciel injecte du Javascript et demande aux utilisateurs de saisir leur numéro de téléphone mobile. Ce dernier sera par la suite utilisé pour infecter les terminaux mobiles des victimes dans le but d'intercepter les SMS. En effet, les SMS envoyés par la banque contiennent le numéro d'autorisation de transaction (TAN, de l'anglais « *Transaction Authentication Number* »). Afin de dérober l'argent, Eurograbber manipule le TAN pour autoriser frauduleusement ses propres transactions.

1.3.3 L'attaque par chevaux de troie

Les chevaux de Troie sont aussi utilisés sur le téléphone intelligent pour prendre le contrôle total d'un équipement. C'est un moyen utilisé pour obtenir des informations privées ou installer d'autres applications malicieuses comme des *botnets* ou des vers. Ils sont utilisés pour transmettre des informations obtenues par l'hameçonnage.

1.4 L'analyse des applications Android

Beaucoup d'efforts ont été déployés afin de prévenir et de détecter rapidement les dysfonctionnements informatiques. En effet, la gravité et l'impact de ces dysfonctionnements varient considérablement. En particulier, un simple bogue informatique peut non seulement engendrer des pertes financières importantes mais aussi des pertes de vies humaines. Dans ce contexte, diverses approches ont été suggérées en vue d'établir une analyse rigoureuse des programmes informatiques et plus particulièrement des applications Android. Parmi ces approches, on distingue l'analyse statique, l'analyse dynamique ainsi que l'analyse hybride.

1.4.1 L'analyse statique

L'analyse statique désigne une famille de méthodes visant l'extraction des informations pertinentes sur le comportement du programme sans l'exécuter. Dans le cadre de la détection des maliciels sur Android, l'analyse statique est utilisée pour déceler les comportements menaçant la sécurité. Cette analyse, assure généralement une bonne couverture du code, vu qu'on considère toute les exécutions possibles. De surcroît, elle permet de détecter les défaillances logicielles ou dans notre cas les comportements malicieux avant qu'ils ne se produisent. Cela dit, elle permet d'éviter la contamination lors de la manipulation des maliciels. Par ailleurs, l'analyse statique se montre vulnérable face aux techniques d'obscurcissement de code et se retrouve toujours face au défi d'indécidabilité. En effet, cette approche se base sur des approximations qui influencent l'exactitude et la précision des résultats qu'elle fournit. En général, on adopte des approximations conservatrices menant à un certain nombre de faux positifs.

1.4.2 L'analyse dynamique

A l'encontre de l'analyse statique, l'analyse dynamique se base sur l'exécution du code. Notamment, l'analyse dynamique permet d'étudier le comportement des applications Android et leurs interactions avec le système en exécutant ces dernières dans un émulateur ou dans un appareil mobile réel. Cette façon de faire expose le système à des risques si l'environnement d'exécution n'est pas suffisamment sécurisé et isolé. Elle permet d'effectuer un examen minutieux d'une ou plusieurs exécutions concrètes. Toutefois, malgré que cette approche corrige les inconvénients de l'analyse statique et plus précisément les limitations occasionnées par le chargement dynamique du code, elle s'avère très coûteuse en temps. De surcroît, l'analyse dynamique ne permet qu'une couverture partielle des exécutions possibles.

1.4.3 L'analyse hybride

Étant donné que les deux approches, statique et dynamique, se montrent complémentaires, il serait alors très utile de déployer une approche combinant les deux analyses afin de bénéficier des avantages de chacune. Une approche hybride visant la détection de malware sur Android, se

base sur l'extraction statique des informations sémantiques relatives à la sécurité sur Android, ainsi que sur l'observation et l'analyse du comportement des applications durant leur exécution.

1.5 Les techniques de détection de maliciel sur Android

La prolifération effervescente des logiciels malveillants Android a suscité un grand intérêt pour le développement des techniques efficaces permettant d'analyser et de détecter automatiquement les nouvelles menaces. L'une des techniques les plus courantes est la détection basée sur les signatures. Remontant aux années quatre-vingts, cette dernière repose sur l'identification de signatures, soit des éléments du code correspondants à un maliciel.

Bien qu'elle soit très précise, cette technique peut être facilement contournée par des techniques d'obscurcissement de code et reste donc vulnérable face aux maliciels métamorphiques (qui réecrivent leur code à chaque réplication) ou inconnus, par exemple les attaques zero day qui exploitent des vulnérabilités logicielles qui ne sont pas encore corrigées, de surcroît il est très difficile de maintenir une actualisation continue des bases de signatures vu l'évolution rapide des logiciels malveillants. Cela dit, il s'avère nécessaire de mettre en œuvre des techniques pour la détection de maliciels. Dans ce contexte, de nombreuses études ont été réalisées.

1.5.1 Détection basée sur l'apprentissage automatique

L'utilisation de l'apprentissage automatique pour la détection de maliciels devient de plus en plus grande. Nombreuses sont les approches qui formulent le problème de détection en un problème de classification. En effet, les approches basées sur l'apprentissage automatique, utilisent des informations extraites soit statiquement soit dynamiquement pour entraîner des classifieurs dans le but détecter les instances de maliciels. Ces informations peuvent être simples ou syntaxiques comme des permissions, des appels à des méthodes spécifiques ou bien plus élaborées comme des graphes représentant la sémantique d'un comportement donné.

Parmi les travaux adoptant cette approche, on cite DREBIN [14], un outil de détection de maliciel Android utilisant l'analyse statique pour l'extraction des caractéristiques à partir du code et du fichier *manifest* de l'application. Les caractéristiques résultantes, tels les appels API, les permissions et les adresses réseaux, sont utilisées par un algorithme d'apprentissage automatique pour une classification binaire des applications.

Canfora et al. [21] présentent une approche dynamique de détection de logiciels malveillants basée sur les appels système. Leur méthode permet de sélectionner automatiquement les séquences d'appels système permettant d'illustrer les différences majeures entre les applications bénignes et les applications malicieuses. Ces séquences sont par la suite utilisées par une machine à vecteurs de support (SVM pour « *Support Vector Machine* ») afin de déterminer si une application donnée est malicieuse.

Similairement dans [45], les auteurs proposent une approche combinant l'analyse dynamique et l'apprentissage automatique qui vise la détection et la classification des maliciels Android. Les caractéristiques (en anglais « *features* ») utilisées par les forêts d'arbres décisionnels englobent les appels API, les mutex, les requêtes DNS, les clés de registre et les fichiers consultés.

Contrairement aux approches de classifcation binaire, dans [35] les auteurs présentent une approche qui vise la détection et la classification des maliciels Android selon les différentes familles. Notamment, à partir des graphes de flots de contrôle des programmes, on extrait les séquences relatives à toutes les traces d'exécutions possibles. Les caractéristiques de classification utilisées correspondent aux trigrammes observés dans les séquences résultantes. L'approche utilise BNS (de l'anglais « *Binormal Separation* ») afin de sélectionner et d'évaluer la pertinence des caractéristiques.

Zhu et al. proposent une approche combinant l'apprentissage automatique et la fouille de données [60]. En se basant sur des graphes d'API, les auteurs extraient les séquences des appels API observées dans des maliciels Android issues de la même famille. Ceci tient au fait que les instances malicieuses de la même famille auront tendance à partager les mêmes motifs malicieux. Ces séquences sont filtrées par des techniques de fouilles de données et utilisées comme caractéristiques de classification.

Kwon et al. présentent une abstraction du graphe de téléchargement « downloader-graph » qui capture l'activité de téléchargement des hôtes [39]. En combinant la télémétrie des anti-virus et des systèmes de prévention des intrusions, ils reconstituent et analysent des milliers de graphes de téléchargement. Cette analyse leur a permis d'extraire des indicateurs de malveillance tel la croissance et le diamètre du graphe. Ces indicateurs sont utilisés comme caractéristiques de classification.

Andrana est présenté dans [18]. C'est un outil léger (en anglais « *lightweight* ») de détection de maliciel sur la plateforme Android. En utilisant l'analyse statique, l'extraction des caractéristiques se fait à partir du code désassemblé des applications Android. L'approche se distingue par les caractéristiques utilisées par l'algorithme d'apprentissage automatique, ces derniers englobent en plus des autres caractéristiques habituelles telles les permissions et les appels API, le repérage des techniques d'obscurcissement de code.

Li et al. [40] présentent Obfusifier, un outil de détection de maliciels Android, basé sur l'apprentissage automatique. En se basant sur Jitana [52], un cadriciel d'analyse hybride de programmes, le comportement des applications Android est modélisé par le graphe d'appel. Ce dernier capture la relation entre les méthodes du programme. À partir du graphe d'appel original, une version plus générale est extraite, notamment en ne gardant que les nœuds représentant les méthodes de l'API Android vu que ces dernières ne peuvent être obfusquées. Cette simplification du graphe vise à construire un graphe d'appel préservant la sémantique du programme et plus résistant aux méthodes d'obscurcissement du code. En se basant sur

des méthodes API critiques, qui peuvent être utilisés pour exhiber des comportements malicieux, Obfusifier génère des chemins d'exécutions critique SAPs (de l'anglais « Sensitive API Path »). Pour la détection, un ensemble de caractéristiques extraites à partir du graphe d'appel original, du graphe simplifié ainsi que les SAPs est utilisé pour construire des modèles de classification.

En effet, la sélection des caractéristiques pour la classification joue un rôle très important sur la précision et la qualité des prédictions. C'est dans cette optique que s'inscrit l'article [23] dans lequel les auteurs visent l'amélioration de la robustesse des classifieurs et ce en utilisant des automates modélisant un comportement non désiré comme caractéristiques pour la classification. À partir du bytecode d'une application malveillante, les auteurs construisent un automate représentant le comportement de l'application. Afin d'éliminer les redondances qui résultent du fait qu'un seul comportement peut être décrit via plusieurs méthodes API, ces dernières ont été regroupées dans un ensemble de désignations similaires aux permissions. L'intersection des automates des maliciels est un ensemble de sous-automates représentant chacun un comportement non désiré. La pertinence de ces caractéristiques est évaluée par des poids estimés par un classifieur linéaire, de surcroît, des techniques de fouille de textes ont été utilisées afin de souligner le comportement malveillant spécifique à chaque famille. L'approche présentée permet de générer automatiquement des propriétés temporelles à partir des maliciels.

Toujours dans la même optique, Meng et al. [41] présentent SMART, un outil de détection et de classification de maliciels. En analysant le bytecode des maliciels issus de la même famille, les auteurs construisent le graphe de flots de contrôle, noté CFG (de l'anglais « *Control Flow Graph* »), de chaque méthode dans le but de repérer les méthodes dupliquées relatives aux comportements malicieux. La similarité des méthodes est mesurée à l'aide du 3D-CFG typé. Ces méthodes sont par la suite regroupées dans des ensembles et analysées par un algorithme de différence dans le but d'identifier les parties communes. Finalement des automates déterministes symboliques, notés DSA (de l'anglais « *Deterministic Symbolic Automaton* »), et des coupes en avant d'un DSA notées OBA (de l'anglais « *Object Based Actions* ») sont générés. La détection est faite en deux phases. La première phase se base sur l'apprentissage automatique, l'utilité de cette phase réside dans la réduction du nombre d'application à analyser statiquement. La deuxième phase se fait par analyse statique via des tests d'inclusion des DSA et des OBA.

Wen et al. [57] proposent un outil de détection de maliciel Android. L'approche utilisée combine l'analyse statique et l'analyse dynamique pour l'extraction des caractéristiques. Ces dernières sont filtrées par analyse en composantes principales avant d'être utilisées par une machine à vecteurs de support.

1.5.2 Détection basée sur la vérification du modèle

Kinder et al. [36] présentent une approche de détection de code malicieux basée sur la vérification de modèles. L'outil construit et vérifie le graphe de flots de contrôle des fichiers exécutables désassemblés. Ils proposent un algorithme de vérification explicite de modèle permettant de vérifier l'absence de motifs malicieux, ces derniers étant spécifiés en CTPL, une extension de la logique temporelle arborescente prenant en compte le renommage des registres. En d'autres termes si le graphe de flots de contrôle d'un programme donné respecte une propriété φ décrivant le motif malicieux, alors ce dernier est jugé malveillant.

Dans [44] Battista et al. présentent une approche visant la détection et l'identification des différentes familles des logiciels malveillants Android via la vérification de modèle. L'idée de base est de représenter les applications par un modèle spécifié en CCS, une algèbre de processus, et les propriétés à vérifier, identiquement les différents comportements malicieux caractérisant chaque famille, par un formalisme logique et plus précisément le μ-calcul modal.

Vu que la majorité des maliciels sont des versions modifiées d'applications bénignes, il s'avère très utile de détecter et d'analyser les fragments de code dupliqués. En effet, cette piste de réflexion a été adoptée par Cuomo et al. [26]. L'approche consiste à traduire le bytecode des programmes Java par le langage d'algèbre de processus CCS dans le but de vérifier l'équivalence des processus résultants via des bisimulations. Leur prototype vise la détection de fragments syntactiquement identiques tolérant une variation des identifiants, des littéraux, des types, d'espaces, des layouts et des commentaires.

Mercaldo et al. [42] présentent une approche de détection de rançogiciel, un maliciel qui chiffre les données personnelles. L'approche fait usage des méthodes formelles. Elle consiste à exploiter des algorithmes de vérification de modèles afin de vérifier les propriétés malicieuses d'un rançogiciel exprimées en logique modale. C'est l'une des premières approches utilisant la vérification de modèles pour identifier et détecter les logiciels de rançon sur Android.

Dans [49], les auteurs présentent un détecteur de logiciels malveillants basé sur la vérification des systèmes à pile. L'outil présenté détecte les maliciels dissimulés dans les programmes binaires et ce via l'extraction d'un modèle exprimé en termes de PDS (systèmes à pile), et la vérification de propriétés spécifiées en SCTPL et SLTPL, des extensions de CTL et LTL permettant une meilleure expressivité. Les performances de leur outil dépassent dans certains cas ceux d'autres antivirus.

L'approche présentée par Song et Touili [50] utilise la vérification de modèles pour la détection des maliciels Android. L'outil développé est doté d'un générateur de modèle basé sur le désassembleur d'applications Android Smali, permettant de traduire les applications par des PDS. Le caractère hostile des applications est déterminé par des algorithmes de vérification de modèle qui détectent la présence de certains comportements malicieux formulés en SCTPL

et SLTPL. SCTP (respectivement SLTPL) est une extension de la logique temporelle arborescente (respectivement la logique temporelle linéaire) avec des variables, des quantificateurs et des prédicats sur la pile.

Ladarola et al. [31] présentent un outil de détection de maliciel bancaire sur Android basé sur la vérification de modèles. En se basant sur les fichiers « .class », les applications Android sont modélisées par des modèles exprimés en termes de processus CCS. Les comportements malicieux exhibés par les maliciels bancaire sont spécifiés en μ-calcul modal. Le vérificateur de modèle CWB-NC est utilisé pour vérifier si une application donnée exhibe un comportement malicieux. Notamment, une application est considérée malicieuse si son modèle respecte une des propriétés temporelles utilisées pour spécifier les comportements malicieux.

JPF-Android [53], un outil de vérification d'applications Android basé sur JavapathFinder, un vérificateur du Java bytecode développé au centre de recherche NASA AMES. JPF-Android fournit un modèle simplifié du framework Android sur lequel les applications Android peuvent s'exécuter. La vérification de toute application via cet outil requiert un fichier *.jpf, définissant un fichier où les propriétés à vérifier ont été spécifiées, ainsi qu'un script contenant une séquence d'événements permettant de simuler l'exécution des applications.

Conclusion

La prolifération de maliciels visant la plateforme Android a nécessité le développement des outils d'analyse d'application Android. C'est dans cette perspective que s'inscrit ce travail visant le développement d'une méthode de détection de maliciels Android. Nous avons présenté dans ce chapitre le contexte de ce travail. Par ailleurs, nous y avons introduit les concepts de bases liés au système Android, les différentes techniques d'analyse des applications Android ainsi que les approches de détection se basant sur la vérification de modèles et l'apprentissage automatique.

Chapitre 2

Définitions et concepts de base

Vérifier la bonne fonctionnalité des systèmes informatiques représente un véritable défi et ce en particulier dans les domaines où la sécurité et la justesse des systèmes informatiques est très importante. Dans ce contexte, plusieurs méthodes formelles ont été élaborées afin d'analyser et de vérifier rigoureusement la cohérence des systèmes par rapport à leurs spécifications. Parmi ces méthodes, nous citons la vérification de modèles. Dans le présent chapitre, nous présentons quelques notions de base liées à cette méthode. Par ailleurs, nous y définissons brièvement les systèmes de transition étiquetées, le langage de spécification formel LNT ainsi que le μ-calcul modal.

2.1 Le processus de la vérification de modèles

La vérification de modèles (en anglais « *model-checking* ») est une technique de vérification qui explore tous les états possibles du système, il s'agit d'une technique automatisée qui, étant donné un modèle ayant un nombre fini d'états et une propriété formelle, vérifie si le modèle satisfait la propriété en question. En d'autres termes, le problème de la vérification de modèles s'énonce comme suit :

> Soit le modèle M, une représentation du système informatique par un système de transitions. Soit φ une formule logique.
> Il s'agit de vérifier que M satisfait φ, noté $M \models \varphi$. (2.1)

Selon Baier et Katoen [17] le processus de vérification comporte trois étapes : la modélisation, l'exécution et l'analyse.

La modélisation : Cette étape est très importante car les résultats de la vérification dépendent du modèle construit. En effet, un algorithme de vérification de modèle prend en entrée un modèle du système et une spécification de la propriété qu'on veut vérifier. Spécifiquement,

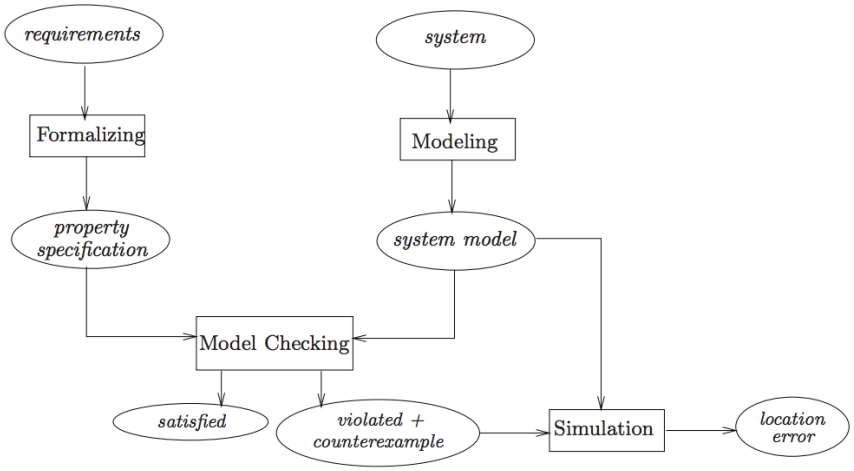

FIGURE 2.1 – Le processus de la vérification de modèles [17].

les modèles décrivent de manière précise et non ambiguë le comportement des systèmes. Ils sont principalement exprimés au moyen d'automates à états finis, constitués d'un ensemble fini d'états et d'un ensemble de transitions nommés dans ce contexte, systèmes de transitions étiquetées (LTS, de l'anglais « *Labelled Transition Systems* »). Le modèle correspond à une abstraction du système à analyser, c'est un moyen de simplifier la tâche de la vérification en éliminant les détails non pertinents dans le cadre de l'analyse effectuée. Par conséquent, l'extraction d'un modèle conservant le bon niveau d'abstraction est une tâche difficile qui dépend fortement de ce qu'on veut analyser.

Tout comme le système, les propriétés à vérifier doivent être spécifiées de manière non ambiguë. En l'occurrence, les logiques temporelles, et spécifiquement le μ-calcul modal, ont été utilisées comme langage de spécification des propriétés. La logique temporelle est une extension de la logique conventionnelle avec des opérateurs exprimant la notion du temps. Elle permet de spécifier un large éventail de propriétés des systèmes telles que la vivacité (quelque chose de bon finit par arriver), l'accessibilité (une certaine situation peut être atteinte), la sûreté (quelque chose de mauvais n'arrive jamais), etc.

L'exécution du vérificateur de modèles : Après la modélisation du système et des propriétés à vérifier vient la phase de la vérification. Notamment, à l'aide d'un algorithme on vérifie en explorant tous les états du système si le modèle satisfait la propriété en question.

L'analyse des résultats : Une fois les deux étapes précédentes accomplies, il en résulte trois scénarios possibles : Le modèle satisfait la propriété en question, la propriété n'est pas vérifiée par le modèle ou bien le modèle est trop grand pour être vérifié et dépasse les limites physiques de l'ordinateur.

Comme illustré sur la figure 2.1, le vérificateur explore les états du modèle pour vérifier si ce dernier satisfait la propriété souhaitée. Si un état qui viole la propriété considérée est détecté, le vérificateur de modèle fournit un contre-exemple indiquant comment le modèle peut atteindre l'état non souhaité. Le contre-exemple décrit un chemin d'exécution, en général sous de forme d'une séquence d'états, menant de l'état initial du système à un état non conforme à la propriété en cours de vérification.

Dans la pratique, le model-checking souffre du problème d'explosion combinatoire d'états. En effet, la taille du modèle augmente exponentiellement avec le nombre de variables du système. Ce problème est d'autant plus accentué lors de la vérification des systèmes concurrents. En effet, la complexité et le temps d'exécution des algorithmes de vérification dépend de la taille du modèle ainsi que de celle de la propriété à vérifier. À titre d'exemple, dans un programme dont toutes les instructions peuvent s'exécuter en parallèle, constitué de n *threads* contenant m instructions chacun, on aura $\frac{(nm)!}{(m!)^n}$ entrelacements possibles. Afin de pallier ce problème, il existe plusieurs techniques permettant la réduction d'espaces d'états, telles que la vérification à la volée (en anglais « *on-the-fly verification* »), les diagrammes de décision binaire et la réduction d'ordre partiel.

2.2 La modélisation des systèmes concurrents

En informatique théorique, les systèmes informatiques logiciels et matériels sont modélisés par des systèmes de transitions. Ce modèle de base consiste à considérer que le système passe d'un état à un autre en fonction de l'action effectuée par l'environnement ou par le système lui-même. Un système de transition peut être visualisé sous forme de graphe où les sommets sont les états et les arêtes représentent les transitions entre les états. Les arêtes peuvent être étiquetées. Une étiquette représente une action qui déclenche un changement d'état. Parmi les systèmes de transitions les plus utilisés, on trouve les structures de Kripke et les systèmes de transitions étiquetées. Dans le cadre de notre étude, nous nous servons de la boite à outil CADP et ainsi nous utilisons les systèmes de transitions étiquetées.

Définition 2.2.1 *Systèmes de transition étiquetées*
Un système de transitions étiquetées LTS (de l'anglais « Labelled Transition System ») est un 5-uplet $(S, Act, \rightarrow, s_0, T)$ *tel que :*

- S *est un ensemble d'états ;*
- Act *est un ensemble d'actions excluant l'action silencieuse* i *;*
- $\rightarrow \subseteq S \times (Act \cup \{i\}) \times S$ *est la relation de transition ;*
- $s_0 \in S$ *est l'état initial ;*
- $T \subseteq S$ *l'ensemble des états finaux.*

Si $(s, a, s') \in \rightarrow$ *on écrit* $s \xrightarrow{a} s'$

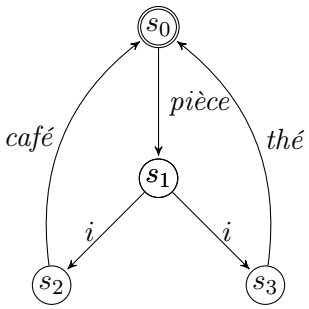

FIGURE 2.2 – LTS d'un distributeur de boissons.

Exemple 2.2.1 *La figure 2.2 représente le système de transitions étiquetées d'un distributeur de boissons. Rappelons que « i » est l'action silencieuse. L'ensemble des états est* $S = \{s_0, s_1, s_2, s_3\}$*, l'état initial est* s_0 *et l'ensemble des actions est* $Act = \{pièce, thé, café\}$.

2.3 La logique de Hennessy Milner

La logique de Hennessy Milner [30] est l'une des logiques modales utilisées pour spécifier les propriétés des systèmes informatiques décrits par des systèmes de transitions étiquetées. Elle permet d'exprimer des propriétés sur les états et les transitions. Cette section présente la syntaxe et la sémantique de cette logique.

2.3.1 La syntaxe et la sémantique de la logique de Hennessy Milner

Une formule de la logique de Hennessy Milner (HML) sur un ensemble d'actions Act est définie par la syntaxe suivante :
$\varphi, \psi ::= true \mid false \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid [a]\varphi \mid \langle a\rangle\varphi$, avec $a \in Act$.
Pour ce qui est de la sémantique, on dit qu'un LTS satisfait une formule de Hennessy Milner si son état initial satisfait la propriété. Étant donné un LTS $L = (S, Act, \rightarrow, s_0, T)$, s un état, et φ une formule HML. On définit $L, s \models \varphi$, qui se lit « s satisfait φ dans L », par induction sur φ, comme suit :

- $L, s \models true$ est satisfaite dans tous les états ;
- $L, s \models false$ n'est satisfaite dans aucun état ;
- $L, s \models \neg\varphi$ ssi $L, s \not\models \varphi$;
- $L, s \models \varphi \wedge \psi$ ssi $L, s \models \varphi$ et $L, s \models \psi$;
- $L, s \models \varphi \vee \psi$ ssi $L, s \models \varphi$ ou $L, s \models \psi$;
- $L, s \models \varphi \Rightarrow \psi$ ssi $L, s \models \varphi$ implique $L, s \models \psi$;
- $L, s \models \langle a\rangle\varphi$ ssi il existe $t \in S$ tq $s \xrightarrow{a} t$ et $L, t \models \varphi$;
- $L, s \models [a]\varphi$ ssi pour tout $t \in S$ $s \xrightarrow{a} t \Rightarrow L, t \models \varphi$.

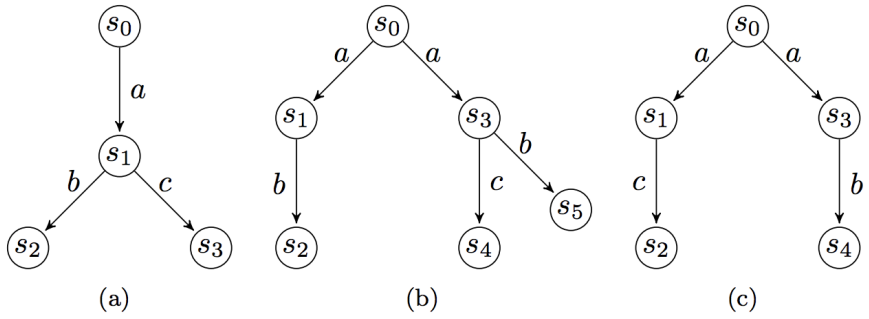

FIGURE 2.3 – Exemples de systèmes de transitions.

2.3.2 La sémantique dénotationnelle de HML

Soit L un LTS et φ une formule HML. On définit inductivement $[\![\varphi]\!] \subseteq S$ comme suit :

- $[\![\mathit{true}]\!]_L = S$
- $[\![\mathit{false}]\!]_L = \emptyset$
- $[\![\neg\varphi]\!]_L = S \setminus [\![\varphi]\!]_L$
- $[\![\varphi \wedge \psi]\!]_L = [\![\varphi]\!]_L \cap [\![\psi]\!]_L$
- $[\![\varphi \vee \psi]\!]_L = [\![\varphi]\!]_L \cup [\![\psi]\!]_L$
- $[\![\varphi \Rightarrow \psi]\!]_L = (S \setminus [\![\varphi]\!]_L) \cup [\![\psi]\!]_L$
- $[\![\langle a \rangle \varphi]\!]_L = \{ s \in S \mid (\exists t \in S \mid s \xrightarrow{a} t \wedge t \in [\![\varphi]\!]_L) \}$
- $[\![[a] \varphi]\!]_L = \{ s \in S \mid (\forall t \in S \mid s \xrightarrow{a} t \Rightarrow t \in [\![\varphi]\!]_L) \}$

Exemple 2.3.1 *Soit ψ_1, ψ_2 deux propriétés de la logique HML, tel que $\psi_1 = \langle a \rangle (\langle b \rangle \mathit{true} \wedge \langle c \rangle \mathit{true})$ et $\psi_2 = \langle a \rangle \langle b \rangle \mathit{true} \wedge \langle a \rangle \langle c \rangle \mathit{true}$. Pour qu'un modèle satisfasse la propriété ψ_1, il faut qu'il puisse effectuer l'action a et que celle-ci mène vers un état à partir duquel il peut effectuer l'action b et l'action c. À titre d'exemple, les modèles* (a) *et* (b) *de la figure 2.3 satisfont cette propriété. La propriété ψ_2 exprime une transition immédiate étiquetée par l'action a menant à un état qui admet une transition immédiate étiquetée par l'action b, ou une transition immédiate étiquetée par l'action a menant à un état qui admet une transition immédiate étiquetée par l'action c. On remarque que le modèle* (c) *satisfait la propriété ψ_2 mais ne satisfait pas ψ_1 car après avoir effectué l'action a, ce dernier n'a plus le choix entre l'action b et l'action c.*

2.4 Treillis et théorèmes de point fixe

Définition 2.4.1 *Une relation d'ordre*
Soit E un ensemble et soit $\leq$ une relation binaire sur E. On dit que $\leq$ est une relation d'ordre si :

- $\leq$ *est réflexive,* $(\forall x \in E \mid x \leq x)$ *;*
- $\leq$ *est transitive,* $(\forall x, y, z \in E \mid x \leq y \text{ et } y \leq z \Rightarrow x \leq z)$ *;*
- $\leq$ *est antisymétrique,* $(\forall x, y \in E \mid x \leq y \text{ et } y \leq x \Rightarrow x = y)$.

Définition 2.4.2 *Un ensemble partiellement ordonné (poset)*
On appelle poset (de l'anglais « Partially Ordered Set ») ou ensemble partiellement ordonné tout ensemble muni d'une relation d'ordre.

Définition 2.4.3 *Un treillis complet*
Un treillis complet est un ensemble partiellement ordonné $(E, \leq)$ dans lequel chaque sous ensemble A possède une borne supérieure, notée $\vee A$, et une borne inférieure, notée $\wedge A$.

Exemple 2.4.1 *L'ensemble des parties d'un ensemble S muni de la relation d'inclusion est un treillis complet. En effet, on a $\subseteq$ est une relation d'ordre, ainsi $(2^S, \subseteq)$ est ensemble partiellement ordonné. Soit $X \subseteq 2^S$, montrons que $\vee X = \bigcup\{C \mid C \in X\}$. Supposons que U est un majorant de X, c'est-à-dire $\forall C \in X,\ C \subseteq U$. Ainsi $\bigcup\{C \mid C \in X\} \subseteq U$. De plus $\bigcup\{C \mid C \in X\}$ est un majorant de X car $\forall C \in X,\ C \subseteq \bigcup\{C \mid C \in X\}$ et ainsi par définition $\bigcup\{C \mid C \in X\}$ est la borne supérieure de X. Similairement, supposons que V est un minorant de X. On a $\forall C \in X,\ V \subseteq C$. Donc $V \subseteq \bigcap\{C \mid C \in X\}$ et $\bigcap\{C \mid C \in X\}$ est un minorant de X car $\forall C \in X,\ \bigcap\{C \mid C \in X\} \subseteq C$ et ainsi $\bigcap\{C \mid C \in X\}$ est la borne inférieure de X. Donc $(2^S, \subseteq)$ est un poset, et chaque sous ensemble de 2^S possède une borne supérieure et une borne inférieure ainsi l'ensemble des parties d'un ensemble S est un treillis complet.*

Théorème 2.4.1 *Knaster-Tarski*
Soit $(E, \leq)$ un treillis complet et $f : E \to E$ une fonction monotone, c'est-à-dire pour tout x et y dans E $x \leq y \implies f(x) \leq f(y)$. Alors f a un plus petit point fixe qui est le plus petit élément x de E tel que $f(x) \leq x$ et un plus grand point fixe qui est le plus grand élément x de E tel que $x \leq f(x)$. De plus, l'ensemble des points fixes de f est un treillis complet (pour $\leq$).

Définition 2.4.4 *$\vee$-continuité et $\wedge$-continuité*
Soit $(E, \leq)$ un treillis complet. On dit que C est une chaîne si elle est non vide et chaque deux éléments de C sont comparables par $\leq$.
Une fonction $f : E \to E$ est dite $\vee$-continue si $\forall C \subseteq E,\ f(\vee C) = \vee\{f(c) \mid c \in C\}$. Dualement f est dite $\wedge$-continue si pour tout $C \subseteq E,\ f(\wedge C) = \wedge\{f(c) \mid c \in C\}$. La $\vee$-continuité ou la $\wedge$-continuité impliquent la monotonie de la fonction. $\bot$ et $\top$ indiquent respectivement le plus petit et le plus grand élément du treillis complet.

Lemme 2.4.1 *Soit S un ensemble fini et $f : 2^S \to 2^S$ une fonction monotone. Alors f est $\vee$-continue et $\wedge$-continue.*

Théorème 2.4.2 *théorème du point fixe de Kleene*
Soit $(E, \leq)$ un ordre partiel complet et $f : E \to E$ une fonction $\vee$-continue alors le plus petit point fixe de f noté μf est défini comme suit :

$$\mu f = \vee\{f^i(\bot) \mid i \geq 0\}.$$

Dualement, le plus grand point fixe de f noté νf est défini comme suit :

$$\nu f = \wedge\{f^i(\top) \mid i \geq 0\}.$$

2.5 Le μ-calcul modal

Le μ-calcul modal [38] est une extension de la logique temporelle de Hennessy Milner avec des opérateurs de points fixes. Similairement à la logique HML, le μ-calcul est défini sur un ensemble d'actions Act et un ensemble de variables Var. Étant donné $a \in Act$ et $X \in Var$, les formules du μ-calcul modal sont construites selon la grammaire suivante :
$\varphi, \psi ::= true \mid false \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid [a]\varphi \mid \langle a\rangle\varphi \mid \mu X.\varphi \mid \nu X.\varphi \mid X$
Tout état satisfait *true* et aucun état ne satisfait *false*. Intuitivement, un état s satisfait la formule $\langle a\rangle\varphi$ s'il existe un a-successeur de s, identiquement une a-transition immédiate, satisfaisant la formule φ. Dualement, un état s satisfait la formule $[a]\varphi$ si tout a-successeur de s satisfait la formule φ. Afin de garantir l'existence du plus petit et plus grand points fixes, on applique une restriction dite de monotonie syntaxique. Cette restrition stipule que les occurences de X dans φ ne peuvent apparaître que sous un nombre pair de négations. Dans le contexte du μ-calcul modal, les formules sont interprétées dans un environnement $\eta : Var \to 2^S$ défini comme suit :

$$\eta[X := S'](Y) = \begin{cases} S' & \text{si } X = Y \\ \eta(Y) & \text{sinon.} \end{cases} \tag{2.2}$$

En appliquant les théorèmes du point fixe dans le treillis $(2^S, \subseteq)$, on définit inductivement la sémantique du μ-calcul modal comme suit :

- $[\![\, true \,]\!]^\eta_L = S$
- $[\![\, false \,]\!]^\eta_L = \emptyset$
- $[\![\, X \,]\!]^\eta_L = \eta(X)$
- $[\![\, \neg\varphi \,]\!]^\eta_L = S \setminus [\![\, \varphi \,]\!]^\eta_L$
- $[\![\, \varphi \wedge \psi \,]\!]^\eta_L = [\![\, \varphi \,]\!]^\eta_L \cap [\![\, \psi \,]\!]^\eta_L$
- $[\![\, \varphi \vee \psi \,]\!]^\eta_L = [\![\, \varphi \,]\!]^\eta_L \cup [\![\, \psi \,]\!]^\eta_L$
- $[\![\, \varphi \Rightarrow \psi \,]\!]^\eta_L = (S \setminus [\![\, \varphi \,]\!]^\eta_L \,) \cup [\![\, \psi \,]\!]^\eta_L$
- $[\![\, \langle a\rangle\varphi \,]\!]^\eta_L = \{s \in S \mid \exists t \in S,\, s \xrightarrow{a} t \wedge t \in [\![\, \varphi \,]\!]^\eta_L \,\}$

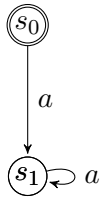

FIGURE 2.4 – Un LTS à deux états satisfaisant la formule $\nu X.\langle a \rangle X$.

- $[\![[a]\varphi]\!]_L^\eta = \{s \in S \mid \forall t \in S,\, s \xrightarrow{a} t \Rightarrow t \in [\![\varphi]\!]_L^\eta \}$
- $[\![\mu X.\varphi]\!]_L^\eta = \bigcap \{S' \subseteq S \mid [\![\varphi]\!]_L^{\eta[X:=S']} \subseteq S'\}$
- $[\![\nu X.\varphi]\!]_L^\eta = \bigcup \{S' \subseteq S \mid S' \subseteq [\![\varphi]\!]_L^{\eta[X:=S']} \}$

Pour faciliter le calcul des plus petit et plus grand points fixes, on applique le lemme 2.4.1 et le théorème de Kleene 2.4.2 à la fonction $\Phi_\eta : 2^S \to 2^S$ définie par $\Phi_\eta(T) = [\![\varphi]\!]_L^{\eta[X:=T]}$.
Ainsi, la sémantique du plus petit et plus grand points fixes est définie comme suit :

- $[\![\mu X.\varphi]\!]_L^\eta = \bigcup \{\Phi_\eta^i(\emptyset) \mid i \in \mathbb{N}\}$
- $[\![\nu X.\varphi]\!]_L^\eta = \bigcap \{\Phi_\eta^i(S) \mid i \in \mathbb{N}\}$

avec $\Phi_\eta^0(T) = T$ et pour tout $n, \in \mathbb{N}^*\ \Phi_\eta^{n+1}(T) = \Phi(\Phi_\eta^n(T))$.

Exemple 2.5.1 *Soit $\varphi = \nu X.\langle a \rangle X$. Vérifions si le système de transition représenté par la figure 2.4 satisfait la propriété φ. Calculer $[\![\nu X.\langle a \rangle X]\!]_L$ revient à calculer le plus grand point fixe de $\Phi(T)$, avec $\Phi(T) = [\![\langle a \rangle X]\!]_L^{\eta[X:=T]}$.*

$$
\begin{aligned}
\Phi^0(S) &= S \\
\Phi^1(S) &= [\![\langle a \rangle X]\!]_L^{[X:=S]} \\
&= \{s \in S \mid \exists t \in S\, s \xrightarrow{a} t \wedge t \in [\![X]\!]_L^{\eta[X:=S]} \} \\
&= \{s \in S \mid \exists t \in S\, s \xrightarrow{a} t \wedge t \in \eta[X := S](X)\} \\
&= \{s \in S \mid \exists t \in S\, s \xrightarrow{a} t \wedge t \in S\} \\
&= S
\end{aligned}
$$

Ainsi le système de transition en question satisfait la propriété φ. En général, tout système de transition qui à partir de son état initial peut effectuer une infinité d'actions « a » satisfait cette propriété.

2.6 Le langage LNT

LNT est un langage de spécification formel de haut niveau supporté par CADP, une boîte à outils qui implémente les techniques de vérification de modèles. Il s'agit d'une version simplifiée du langage E-Lotos avec des notations plus conviviales empruntées aux langages fonctionnels

⟨*definition_processus*⟩ : := **process** Π [[$fg_0, \ldots, fg_m$]]
[[$fp_1, \ldots, fp_n$]] **is**
$pp_1, \ldots, pp_l$
⟨*comportement*⟩
end process

⟨*comportement*⟩ : := **null**	sans effet (sans continuation)
| **stop**	(avec continuation)
| B_1 **;** B_2	composition séquentielle
| $G[(O_0, \ldots, O_n)]$[**where** V]	communication
| X := V	assignation déterministe
| **var** $vd_0, \ldots, vd_n$ **in**	déclaration des variables
B_0	
end var	
| **loop**	boucle infinie
B_0	
end loop	
| **while** V **loop**	boucle while
B_0	
end loop	
| **break** L	interruption d'une boucle
| Π [[actual_gates]]	
[($ap_1, \ldots, ap_n$)]	appel de processus
| **select**	choix non déterministe
B_0 [ ] ... [ ] B_n	
end select	
| **par** [$G_0, \ldots, G_n$ **in**]	composition parallèle
[$G_{(i,0)}, \ldots, G_{(i,n_i)}$ ->] B_0	
‖ ... ‖	
[$G_{(i,0)}, \ldots, G_{(i,n_i)}$ ->] B_m	
end par	

FIGURE 2.5 – Syntaxe des processus en LNT

et impératifs. Une spécification LNT correspond à un ensemble de modules et possède des composants de contrôle et de données, identiquement des types, des fonctions, des canaux et des processus définis dans un module. Comme dans d'autres langages d'algèbre de processus, en LNT les comportements de système sont exprimés en termes de processus. Un processus LNT est défini par un identifiant, un comportement, une liste optionnelle de portes formelles, une liste optionnelle de paramètres formels et une liste optionnelle de pragmas de processus. Les pragmas de processus ne sont utilisés que pour donner des indications sur la manière dont la traduction du code source en Lotos et en C doit être effectuée. Dans ce qui suit nous présentons brièvement le langage LNT en nous concentrant uniquement sur les éléments essentiels liés à la définition des processus. Pour plus de détails sur le langage LNT, nous

invitons le lecteur à se reporter au manuel de référence [22].

La figure 2.5 présente une syntaxe partielle des processus en LNT. Dans cette grammaire, *ap* représente *paramètre effectif* (en anglais « *actual parameter* »), *fp* représente *paramètre formel* (en anglais « *formal parameter* »), *fg* représente *porte formelle* (en anglais « *formal gate* »), *pp* représente *pragma de processus* (en anglais « *process pragma* »), *vd* représente *déclaration d'une variable* (en anglais « *variable declaration* »), X est une variable, V est une expression, B_i dénote un comportement et G_i représente une porte.

La sémantique dynamique des comportements

La sémantique dynamique des comportements est définie comme un système de transitions étiquetées, dont les états sont des couples, souvent appelés configurations dans la littérature, de la forme $\langle B, \sigma \rangle$, où B est un comportement et σ est un état de la mémoire. L'état initial d'un programme LNT B_0 est $\langle B_0, [\,] \rangle$, où $[\,]$ dénote une mémoire vide. Dans ce qui suit, en se basant sur la sémantique opérationnelle structurée (SOS), nous définissons les transitions d'un LTS de la forme $\langle B, \sigma \rangle \xrightarrow{a} \langle B', \sigma' \rangle$. L'étiquette « a » peut prendre l'une des formes suivantes :

- L'étiquette ✓ désigne une terminaison normale. Toute transition étiquetée par ✓ mène à un état de blocage. Autrement dit, si $a = \checkmark$ alors $B' = \mathbf{stop}$.
- *brk(L)* désigne une terminaison sur un comportement « break L ». Ce dernier permet d'interrompre la boucle L.
- Une étiquette de communication est soit **i** ou « $G(v_1, ..., v_n)$ », G étant une porte et $v_1, ..., v_n$ des valeurs. En LNT, les processus communiquent via des portes (en anglais « *gates* ») permettant l'échange des données. Les portes peuvent être typées, au moyen d'un canal de communication, ou non typées. Soit a dans C, l'ensemble des étiquettes de communications. La fonction $gate(a)$ est définie comme suit :
  $gate(\mathbf{i}) = \mathbf{i}$
  $gate(G(v_1, ..., v_n)) = G$.

Une mémoire, notée, σ correspond à une fonction partielle qui associe des valeurs aux variables. La notation « $[X_1 \leftarrow v_1, \ldots, X_n \leftarrow v_n]$ » ($n \geq 0$ et $i \neq j \Rightarrow X_i \neq X_j$) dénote la mémoire σ où pour tout $X \notin \{X_1, \ldots, X_n\}$, $\sigma(X)$ est non définie et $\sigma(X_1) = v_1, \ldots \sigma(X_n) = v_n$. Étant donné deux mémoires σ_1 et σ_2, on définit $\sigma_1 \oslash \sigma_2$, $\sigma_1 \oplus \sigma_2$ et $\sigma_1 \ominus \sigma_2$, représentant respectivement la mise à jour de σ_1 par rapport à σ_2, l'union disjointe de σ_1 et σ_2 et la différence entre σ_1 et σ_2.

$$(\sigma_1 \oslash \sigma_2)(X) = \begin{cases} \sigma_2(X) & \text{si } \sigma_2(X) \text{ est définie} \\ \sigma_1(X) & \text{si } \sigma_2(X) \text{ est non définie et } \sigma_1(X) \text{ est définie} \\ \text{non définie} & \text{sinon.} \end{cases} \tag{2.3}$$

Formellement, la définition de l'union disjointe de σ_1 et σ_2 notée $\sigma_1 \oplus \sigma_2$ correspond à celle de $\sigma_1 \oslash \sigma_2$ si les ensembles des variables de σ_1 et σ_2 sont disjoints. Autrement, $\sigma_1 \oplus \sigma_2$ n'est

pas définie.

$$(\sigma_1 \ominus \sigma_2)(X) = \begin{cases} \sigma_1(X) & \text{si } \sigma_1(X) \text{ est définie et } \sigma_2(X) \text{ est non définie ou } \sigma_2(X) \neq \sigma_1(X) \\ \text{non définie} & \text{sinon.} \end{cases} \tag{2.4}$$

Ci-dessous, nous définissons la sémantique de certains comportements à l'aide des règles de la SOS.

L'inaction. Aucune règle de la SOS n'est associée à $\langle \mathbf{stop}, \sigma \rangle$, qui représente l'inaction d'un processus.

La terminaison. L'instruction **null** se termine normalement et n'affecte pas l'état de la mémoire.

$$\langle \mathbf{null}, \sigma \rangle \xrightarrow{\checkmark} \langle \mathbf{stop}, \sigma \rangle.$$

La composition séquentielle. Le comportement « $B_1; B_2$ » commence par exécuter B_1. Si B_1 se termine normalement, B_2 est exécuté dans la mémoire, mise à jour par B_1.

$$\frac{\langle B_1, \sigma \rangle \xrightarrow{\checkmark} \langle \mathbf{stop}, \sigma' \rangle \qquad \langle B_2, \sigma' \rangle \xrightarrow{a} \langle B_2', \sigma'' \rangle}{\langle B_1; B_2, \sigma \rangle \xrightarrow{a} \langle B_2', \sigma'' \rangle}$$

Si B_1 se termine par une instruction « **break** », alors « $B_1; B_2$ » se termine également sur cette instruction.

$$\frac{\langle B_1, \sigma \rangle \xrightarrow{brk(L)} \langle B_1', \sigma' \rangle}{\langle B_1; B_2, \sigma \rangle \xrightarrow{brk(L)} \langle B_1', \sigma' \rangle}$$

Si B_1 propose une étiquette de communication, l'exécution de B_1 doit continuer jusqu'à la fin.

$$\frac{\langle B_1, \sigma \rangle \xrightarrow{a} \langle B_1', \sigma' \rangle \qquad a \in C}{\langle B_1; B_2, \sigma \rangle \xrightarrow{a} \langle B_1'; B_2, \sigma' \rangle}$$

Assignation déterministe. L'exécution d'une assignation déterministe se termine normalement après avoir modifié dans la mémoire la valeur de la variable par la valeur qui lui a été assignée dans la partie droite de cette assignation.

$$\frac{\langle V, \sigma \rangle \to v}{\langle X := V, \sigma \rangle \xrightarrow{\checkmark} \langle \mathbf{stop}, \sigma \oslash [X \leftarrow v] \rangle}$$

Choix non déterministe. Un choix non déterministe parmi les comportements $B_1, ..., B_n$ se comporte comme l'un de ces derniers.

$$\frac{\langle B_i, \sigma \rangle \xrightarrow{a} \langle B_i', \sigma' \rangle}{\langle \mathbf{select}\ B_1[\,]...[\,]B_n\ \mathbf{end\ select}, \sigma \rangle \xrightarrow{a} \langle B_i', \sigma' \rangle} (i \in 1..n)$$

La composition parallèle. Soit S un ensemble de synchronisation correspondant à une étiquette de communication a dans C. La sémantique de la composition parallèle est comme suit :

$$\frac{a \in C \qquad S \in sync(a) \qquad \langle B_i, \sigma\rangle \xrightarrow{a} \langle B'_i, \sigma_i\rangle \, (i \in S) \qquad \langle B'_j, \sigma\rangle = \langle B_j, \sigma\rangle \, (j \in \{1, ..., m\} \setminus S)}{\left\langle \begin{array}{l} \textbf{par}\, G_0, \ldots, G_n \\ G_{(1,0)}, ..., G_{(1,n_1)} \text{->} B_1 \\ \| \ldots \| \qquad\qquad , \sigma \\ G_{(m,0)}, \ldots, G_{(m,n_n)} \text{->} B_m \\ \textbf{end par} \end{array} \right\rangle \xrightarrow{a} \left\langle \begin{array}{l} \textbf{par}\, G_0, ..., G_n \,\textbf{in} \\ G_{(1,0)}, ..., G_{(1,n_1)} \text{->} B'_1 \\ \| \ldots \| \qquad\qquad , \sigma' \\ G_{(m,0)}, \ldots, G_{(m,n_m)} \text{->} B'_m \\ \textbf{end par} \end{array} \right\rangle}$$

Avec $\sigma' \triangleq \sigma \oslash ((\sigma_1 \ominus \sigma) \oplus ... \oplus (\sigma_m \ominus \sigma))$ [*] et $sync(a)$ une fonction qui retourne un ensemble de sous-ensembles de $1..m$ dénotant les indices des comportements B_i, $i = 1, ..., m$ qui synchronisent sur a.

$$sync(a) = \begin{cases} \{\{1..m\}\} & \text{si } gate(a) \in \{G_0, ..., G_n\} \\ \{\{i \mid i \in 1..m \wedge gate(a) \in \{G_{(i,0)}, ..., G_{(i,n_i)}\}\}\} & \text{si } gate(a) \in \bigcup\limits_{i \in 1..m} \{G_{(i,0)}, ..., G_{(i,n_i)}\} \\ \{\{i\} \mid i \in 1..m\} & \text{sinon} \end{cases} \tag{2.5}$$

Si tous les comportements parallèles se terminent normalement, alors la composition parallèle se termine normalement.

$$\frac{\langle B_i, \sigma\rangle \xrightarrow{\checkmark} \langle B'_i, \sigma_i\rangle \, (i \in 1..m)}{\left\langle \begin{array}{l} \textbf{par}\, G_0, \ldots, G_n \,\textbf{in} \\ G_{(1,0)}, \ldots, G_{(1,n_1)} \text{->} B_1 \\ \| \ldots \| \qquad\qquad , \sigma \\ G_{(m,0)}, ..., G_{(m,n_n)} \text{->} B_m \\ \textbf{end par} \end{array} \right\rangle \xrightarrow{\checkmark} \langle \ \textbf{stop}, \sigma' \rangle}$$

Exemples

Choix non déterministe. Comme mentionné précédemment, le mot clé « **select** » permet à un processus d'effectuer un choix non déterministe. À titre d'exemple le processus P, spécifié ci-dessous, effectue un choix non déterministe entre une action sur la porte A ou sur celle de D. On note que dans le LTS (Figure 2.6) deux branches d'exécutions sont possibles. Une fois le choix effectué, le processus exécute le reste de cette branche. En LNT, le mot clé « any » est utilisé pour déclarer les portes formelles non typées.

[*] Le symbole $\triangleq$ se lit : « égal par définition ».

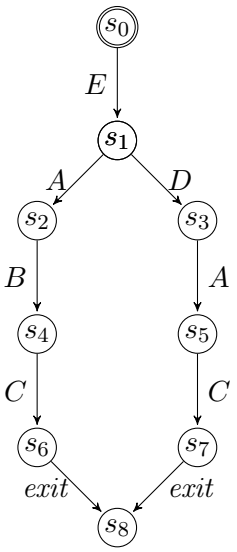

FIGURE 2.6 – LTS correspondant au processus P

module NDC **is**
process MAIN[E,A,B,C,D : any] **is**
  P[E,A,B,C,D]
end process
process P[E,A,B,C,D : any] **is**
  E;
  select
        A;
        B;
        C;
        null
  [ ]
        D;
        A;
        C;
        null
  end select
end process
end module

Composition parallèle. Le processus spécifié ci-dessous représente la composition parallèle des processus P_1 et P_2. Notamment, à travers le mot clé « **par** » on a spécifié les portes sur lesquelles ces deux processus doivent synchroniser leurs actions, soit la porte A. Les actions sur les autres portes s'entrelacent. Il est à noter que l'action de terminaison exit est toujours synchronisée entre tous les processus. Le système de transitions étiquetées résultant de cette composition parallèle est illustré par la figure 2.7.

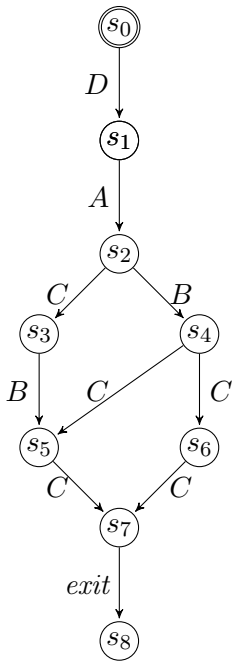

FIGURE 2.7 – LTS correspondant au processus *MAIN* de la composition parallèle

module PC **is**
process MAIN[A,B,C,D : any] **is**
  par A **in**
    P_1[A,B,C]
  ||
    P_2[D,A,C]
  end par
end process
process P_1[A,B,C : any] **is**
  A;
  B;
  C;
  null
end process
process P_2[D,A,C: any] **is**
  D;
  A;
  C;
  null
end process
end module

Conclusion

La vérification de modèles est une technique permettant de vérifier rigoureusement la cohérence des systèmes informatiques par rapport à leur spécification. Ceci dit, cette technique s'avère utile pour la détection de maliciels Android vu qu'elle nous permet de vérifier si une application Android exhibe des comportements malicieux.

À travers ce chapitre, nous avons vu que la modélisation constitue une étape très importante du processus de vérification de modèles. En effet, les modèles ainsi que les propriétés qu'on souhaite vérifier doivent être exprimés dans des langages de spécification. À cet effet, des langages de spécification formelles ont été présentés, notamment le langage LNT et le μ-calcul modal.

Chapitre 3

La vérification des applications Android

Dans le but de détecter le caractère malicieux des applications Android, nous avons recours à la vérification de modèles qui nous permet de vérifier si un modèle respecte une propriété donnée. À cet effet, les applications Android sont spécifiées par des modèles exprimés en termes de processus LNT. En outre, les propriétés encodant des comportements Android potentiellement malicieux sont modélisables par des formules de μ-calcul modal.

Dans ce chapitre, nous expliquons le processus de la modélisation des applications Android. Par ailleurs, nous présentons les différentes propriétés sur lesquelles les algorithmes d'apprentissage automatique se basent pour une classification binaire des applications Android.

3.1 La modélisation des applications Android

Comme nous l'avons précisé dans le chapitre précédent, la première étape de la vérification de modèle correspond à la modélisation, autrement dit l'extraction d'un modèle abstrait capturant la sémantique des programmes à vérifier. En l'occurrence, une abstraction des applications Android. Dans cette section, nous présentons Apk2Lnt, le programme assurant la construction d'un modèle formel en termes de processus LNT.

3.1.1 La construction du modèle

En nous basant sur un extracteur de modèle en CCS, développé par mon collègue Loïc Ricaud dans le cadre de son stage, nous avons développé Apk2Lnt. C'est un programme Java qui prend en entrée une application Android sous la forme APK et retourne sa spécification en LNT.

L'extraction du modèle passe par la création du graphe de flot de contrôle inter-procédural (ICFG, de l'anglais « *Interprocedural Control Flow Graph* ») de l'application Android. C'est

```
main(){
      x=2;
      y=f(x);
      x=y;
      z=f(x+1);
          }
f(int  n){
      if(n>10)
         p=2;
      else
         p=4;
      return p;
          }
```

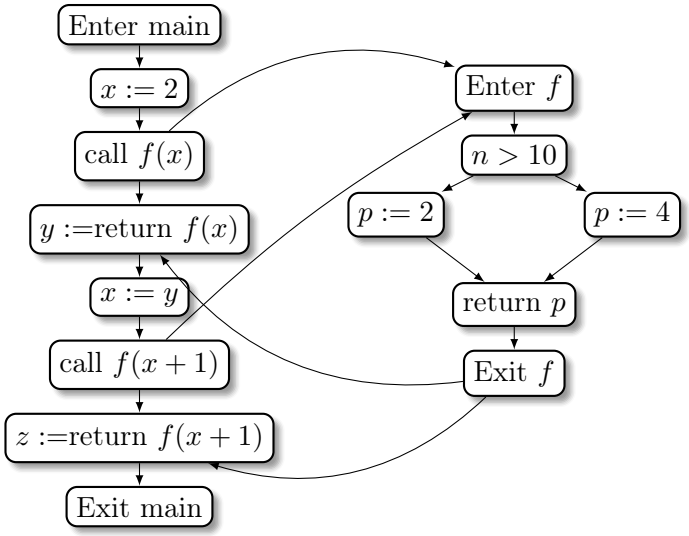

FIGURE 3.1 – Exemple de graphe de flot de contrôle inter-procédurale.

une représentation sous forme de graphe de tous les chemins d'exécution possibles. Dans ce graphe, les nœuds correspondent aux instructions du code et les arcs représentent les sauts dans le flot de contrôle. La figure 3.1 représente un exemple d'ICFG.

Pour construire l'ICFG, nous procédons par la création des graphes de flot de contrôle intra-procéduraux, soit les graphes de flot de contrôle de chaque fonction du programme. Une fois construits ces graphes sont regroupés dans le but d'obtenir l'ICFG de l'application. Les graphes intra-procéduraux ont été construits par Soot [47], un cadriciel dédié à l'analyse des programmes Java et Android.

L'algorithme conçu pour l'extraction du modèle est présenté ci-dessous. Notamment, en visitant chaque nœud du ICFG, nous créons un processus et nous le définissons par ses successeurs.

Algorithme 1 Création du code LNT

```
1: Construction du ICFG.
2: Création d'une pile contenant la première unité (unit) du ICFG.
3: tant que pile non vide faire
4:     unit ← l'élément au sommet de la pile.
5:     succs ← successeurs de unit
6:     si unit est un point de sortie ∨ (succs ≠ ∅ ∧ unit non définie) alors
7:         Définir unit.
8:         Empiler succs dans la pile.
```

Afin de faciliter la compréhension de la façon dont le ICFG a été traduit en LNT, nous citons quelques exemples de fragments de code d'applications Android ainsi que leurs modèles LNT correspondants. Ces bouts de code correspondent à une séquence d'instructions, une séquence comportant une instruction conditionnelle, une séquence comportant une boucle et une séquence exécutant des *threads*.

Par souci de clarté, nous introduisons des exemples en Java. Cependant, il est à noter que dans le processus de la traduction vers LNT nous utilisons plutôt Jimple, une représentation intermédiaire typée de Soot. Cette représentation permet de simplifier l'analyse du code. En effet, elle permet de manipuler 15 types d'instructions au lieu de 200 et quelques instructions du bytecode rendant ainsi la tâche d'extraction du modèle plus simple.

Une séquence

Dans les programmes Android, on retrouve des instructions dites de contrôle ainsi que des instructions qui ne modifient pas le contrôle, soit l'ordre d'exécution des instructions du programme. La traduction en LNT d'un bloc d'instructions Java, n'agissant pas sur l'ordre d'exécution du programme, est réalisée par la composition séquentielle de processus.

Soient A et B deux instructions qui ne modifient pas le contrôle ; à titre d'exemples des affectations, des déclarations de variables, etc.

---

1 A;
2 B;
3 ...

Soit π la fonction qui, à partir d'une instruction, extrait l'action correspondante. De manière simplifiée, la traduction en LNT du bloc d'instructions ci-dessus se fait comme suit :

process $P_0[\pi(A),\pi(B)$: any] **is**
$\pi(A)$; $P_1[\pi(B)]$
end process
process $P_1[\pi(B)$: any]
$\pi(B)$; P_2
end process

Chaque instruction est traduite par un processus LNT effectuant une action. Par simplification, les instructions restantes du code sont modélisées par le processus P_2. Pour plus de détail, nous citons l'exemple de traduction d'un bout de code Java.

Le fragment de code ci-dessous correspond à une séquence d'instructions simples (ne contenant aucune instruction de contrôle). Notamment, l'exécution de ce morceau de code résulte en une exécution séquentielle d'instructions Java permettant de retourner des informations sur l'appareil mobile utilisé.

```
1 TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.
      TELEPHONY_SERVICE);
2 String imsi= tm.getSubscriberId();
3 String imei= tm.getImei();
```

```
4  int phoneType = tm.getPhoneType();
```

La composition séquentielle de processus est utilisée pour traduire le séquencement simple d'instructions Java. La traduction de cette séquence en LNT est comme suit :

process P_1[androidcontentContextgetsystemservice,androidtelephonyTelephonyManagergetSubscriberID,androidtelephonyTelephonyManagergetImei,androidtelephonyTelephTelephonyManagergetPhoneType : any] **is**
androidcontentContextgetsystemservice; P_2[androidtelephonyTelephonyManagergetSubscriberID,androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManagergetPhoneType]
end process
process P_2[androidtelephonyTelephonyManagergetSubscriberID,androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManagergetPhoneType : any] **is**
androidtelephonyTelephonyManagergetSubscriberID; P_3[androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManagergetPhoneType]
end process
process P_3[androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManagergetPhoneType : any] **is**
androidtelephonyTelephonyManagergetImei; P_4[androidtelephonyTelephonyManagergetPhoneType]
end process
process P_4[androidtelephonyTelephonyManagergetPhoneType : any] **is**
androidtelephonyTelephonyManagergetPhoneType;P_5
end process

En effet, les actions effectuées par les processus correspondent au noms des méthodes de l'API d'Android.

Une instruction conditionnelle

Pour la spécification d'une instruction conditionnelle en LNT, nous avons utilisé l'opérateur « **select** » pour exprimer le choix entre les branches possibles.

```
1  if(condition)
2  A;
3  else
4  B;
```

La structure conditionnelle **if else** est modélisée par un processus LNT permettant de réaliser le comportement de l'une des branches possibles.

process $P_1[\pi(A), \pi(B) : \text{any}]$ **is**
select
i; P_2
[ ]
i; P_3
end select
end process
process $P_2[\pi(A) : \text{any}]$ **is**
$\pi(A)$; P_4
end process
process $P_3[\pi(B) : \text{any}]$ **is**
$\pi(B)$; P_5
end process

Comme illustré dans le fragment de code ci-dessous deux branches sont possibles, la branche du **if** et celle du **else**.

```
1  TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.
       TELEPHONY_SERVICE);
2  Random rand = new Random();
3  int randNumber = rand.nextInt(10);
4  String str = "";
5  if(randNumber == 4){
6                 str = tm.getSubscriberId();
7                 str.concat("␣IMSI");
8  } else {
9                 str = tm.getImei();
10 }
11 Log.i(TAG, str);
```

La traduction vers LNT des quatre premières lignes du code se fait, comme expliqué dans la section précédente, par la composition séquentielle des processus. Les instructions du code qui ne correspondent pas à un appel d'une méthode de l'API sont traduites par des processus effectuant l'action silencieuse, dénotée par « **i** » en LNT. À titre d'exemple, le processus P_4 effectue l'action silencieuse. Ceci tient au fait que l'instruction correspondante en Java, soit l'instruction à la quatrième ligne du fragment de code, est une assignation. Dans la spécification LNT proposée, le choix entre les branches du **if** est exprimé par le processus P_5, et ce, par le biais de l'opérateur « **select** ».

process P_1[androidcontentContextgetsystemservice,javautilRandomRandom,javautilRandomnextInt,androidtelephonyTelephonyManagergetImei,androidtelephonyTelephony-

ManageretSubscriberID, javalangStringconcat,androidutilLog : any] **is**
androidcontentContextgetsystemservice; P_2[javautilRandomRandom,javautilRandomnextInt,androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManageretSubscriberID, javalangStringconcat,androidutilLogi]
end process
process P_2[javautilRandomRandom,javautilRandomnextInt,androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManageretSubscriberID, javalangStringconcat,androidutilLogi : any]
javautilRandomRandom; P_3[javautilRandomnextInt,androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManageretSubscriberID, javalangStringconcat,androidutilLogi]
end process
process P_3[javautilRandomnextInt,androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManageretSubscriberID, javalangStringconcat,androidutilLogi- : any] **is**
javautilRandomnextInt; P_4[androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManageretSubscriberID, javalangStringconcat,androidutilLogi]
end process
process P_4[androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManageretSubscriberID, javalangStringconcat,androidutilLogi : any] **is**
i; P_5[androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManagergetSubscriberID, javalangStringconcat,androidutilLogi]
end process
process P_5[androidtelephonyTelephonyManagergetImei,androidtelephonyTelephonyManageretSubscriberID, javalangStringconcat,androidutilLogi : any] **is**
select
i; P_6[androidtelephonyTelephonyManagergetSubscriberID,javalangStringconcat,androidutilLogi]
[ ]
i; P_7[androidtelephonyTelephonyManagergetImeit,androidutilLogi]
end select
end process
process P_6[androidtelephonyTelephonyManagergetSubscriberID,javalangStringconcat, androidutilLogi : any] **is**
androidtelephonyTelephonyManagergetSubscriberID; P_8[javalangStringconcat,androidutilLogi]
end process
process P_8[javalangStringconcat,androidutilLogi : any] **is**
javalangStringconcat; P_9[androidutilLogi]

end process
process P_7[androidtelephonyTelephonyManagergetImeit,androidutilLogi : any] **is**
androidtelephonyTelephonyManagergetImeit; P_9[androidutilLogi]
end process
process P_9[androidutilLogi : any] **is**
androidutilLogi; P_{10}
end process

Une boucle

Les boucles sont des structures permettant d'exécuter un bloc d'instructions à plusieurs reprises. En Java, il existe plusieurs types d'instructions répétitives, entre autres on retrouve la boucle while.

```
1 while(condition){
2 A;
3 }
4 B;
```

Les processus récursifs et le choix non déterministe ont été utilisés pour spécifier les boucles.

Process $P_1[\pi(A), \pi(B) : \text{any}]$ **is**
i;P_2
end process
Process $P_2[\pi(A), \pi(B) : \text{any}]$ **is**
select
$\pi(A)$;P_1
[ ]
$\pi(A)$;P_3
end select
end process
process $P_3[\pi(B) : \text{any}]$ **is**
$\pi(B)$;P_4
end process

Dans l'exemple suivant, nous avons défini le processus P_5 de manière récursive afin de spécifier la boucle **while**.

```
1 TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.
    TELEPHONY_SERVICE);
2 Random rand = new Random();
3 int randNumber = rand.nextInt(10);
```

```
4  String str = "";
5  while(randNumber >= 4){
6                 String str1 = tm.getSimCountryIso();
7                 str.concat(str1);
8                 randNumber=randNumber-1;
9  }
10 str.concat("␣SimCountryIso");
```

La traduction vers LNT du fragment de code ci-haut est comme suit :

process P_1[androidcontentContextgetsystemservice,javautilRandomRandom,javautilRandomnextInt,androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat : any] **is**
androidcontentContextgetsystemservice; P_2[javautilRandomRandom,javautilRandomnextInt,androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat]
end process
process P_2[javautilRandomRandom,javautilRandomnextInt,androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat : any]
javautilRandomRandom; P_3[javautilRandomnextInt,androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat]
end process
process P_3[javautilRandomnextInt,androidtelephonyTelephonyManagergetSimCountryIso ,javalangStringconcat : any] **is**
javautilRandomnextInt; P_4[androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat]
end process
process P_4[androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat : any] **is**
i; P_5[androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat]
end process
process P_5[androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat : any] **is**
i; P_6[javalangStringconcat]
end process
process P_6[androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat : any] **is**
androidtelephonyTelephonyManagergetSimCountryIso; P_7[javalangStringconcat]
end process
process P_7[androidtelephonyTelephonyManagergetSimCountryIso,javalangStringco-

ncat : any] **is**
javalangStringconcat; P_8[androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat]
end process
process P_8[androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat : any] **is**
select
i; P_5[androidtelephonyTelephonyManagergetSimCountryIso,javalangStringconcat]
[]
i; P_9[javalangStringconcat]
end select
end process
process P_9[javalangStringconcat : any] **is**
javalangStringconcat; P_{10}
end process

On remarque que le processus P_8 offre le choix entre deux branches. La branche à l'extérieur du **while** spécifiée par le processus P_9 et la réexécution du bloc **while** en rebouclant sur le processus P_5.

Les *threads*

La modélisation du parallélisme en LNT se fait par la composition parallèle des processus. En effet, les processus modélisant les threads sont mis en parallèle par l'opérateur « **par** » permettant ainsi de générer tous les entrelacements possibles. Prenons, à titre d'exemple, deux *threads* exécutant différentes instructions dans leur méthode `run()`. Le premier *thread* T_1 exécute le bloc d'instructions ci-dessous :

```
1 TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.
      TELEPHONY_SERVICE);
2 String imei= tm.getImei();
```

Et un deuxième *thread* T_2 qui exécute les instructions suivantes :

```
1 LocationManager locationManager = (LocationManager) getSystemService(
      Context.LOCATION_SERVICE);
2 List<String> providers = locationManager.getProviders(true);
```

L'exécution parallèle des deux *threads* est spécifiée par le processus MAIN du code LNT suivant :

process T_1[androidcontentContextgetsystemservice,androidtelephonyTelephonyM-

anagergetImei: any] **is**
androidcontentContextgetsystemservice; P_1[androidtelephonyTelephonyManagergetImei]
end process
process P_1[androidtelephonyTelephonyManagergetImei : any]
androidtelephonyTelephonyManagergetImei; **null**
end process
process T_2[androidcontentContextgetSystemService,androidlocationLocationManagergetProviders : any] **is**
androidcontentContextgetSystemService; P_2[androidlocationLocationManagergetProviders]
end process
process P_2[androidlocationLocationManagergetProviders : any] **is**
androidlocationLocationManagergetProviders; **null**
end process

process MAIN[androidcontentContextgetsystemservice,androidtelephonyTelephonyManagergetImei,androidcontentContextgetSystemService,androidlocationLocationManagergetProviders : any] **is**
par
T_1[androidcontentContextgetsystemservice,androidtelephonyTelephonyManagergetImei]
||
T_2[androidcontentContextgetSystemService,androidlocationLocationManagergetProviders]
end par
end process

3.1.2 Le raffinement du modèle

Le temps d'exécution d'Apk2Lnt dépend de la taille de l'application donnée en entrée. Dans le but d'accélérer l'étape de la vérification du modèle, il est important que notre constructeur de modèle produise un modèle LNT à la fois précis et concis. Pour ce faire, nous visons à réduire le mieux possible la taille du modèle tout en préservant la sémantique de l'application à analyser. Le processus de traduction d'un APK en LNT décrit dans la section précédente est direct et non-optimisé. En pratique, afin de raffiner le modèle construit, les transformations décrites dans les paragraphes suivants ont été apportées à cette traduction de base.

La réduction du modèle

Lors de la génération du modèle LNT, le constructeur du modèle extrait la liste des actions à considérer dans le cadre de cette analyse, et ce à partir d'un fichier contenant toutes les propriétés que nous souhaitons vérifier. Les actions non pertinentes, soit celles ne figurant pas dans la liste des propriétés qu'on souhaite vérifier, sont remplacées par des actions silencieuses représentées par « **i** » dans le langage LNT. L'objectif étant d'atténuer le problème de l'explosion de l'espace d'états, nous visons la réduction de la taille du modèle, notamment en compactant les chaînes d'actions silencieuses tout en conservant le choix interne.
Soit P_1 le processus suivant :

process P_1 **is**
    i; P_2
end process
process P_2 **is**
    i; P_3
end process
        ...
        ...
        ...
process P_{n-1} **is**
    i; P_n
end process

En appliquant cette méthode de réduction, le processus P_1 sera défini comme suit :

process P_1 **is**
    i; P_n
end process

Le déroulement des boucles

Les outils de CADP n'autorisent pas les spécifications contenant des processus instanciés récursivement via un opérateur de composition parallèle. En effet, les processus ayant une récursion comportant la parallélisation conduisent à un espace d'états infini. Pour illustrer ce constat, prenons l'exemple suivant où A et B sont des actions :

module LU **is**
process $P_1[A, B : \text{any}]$ **is**
    A; $P_2[A, B]$
end process
process $P_2[A, B : \text{any}]$ **is**
    select

```
        par A; P1[A, B] || B; null end par
    [ ]
        i ; P1[A, B]
    end select
end process

process MAIN[A, B : any] is
    P1[A,B]
end process
end module
```

Dans cette spécification, on observe que le processus P_2 instancie de manière récursive le processus P_1 via l'opérateur de composition parallèle. Pour pallier le problème d'explosion d'états occasionné par cette combinaison, le déroulement de la boucle a été appliqué. Principalement, on garde la récursion mais on n'exécute la parallélisation qu'un nombre fini d'itérations. Ce nombre est un paramètre indiqué lors de la génération du modèle, s'il n'est pas défini, la valeur par défaut de ce paramètre est égale à 1. En appliquant la transformation suggérée, la spécification résultante est la suivante :

```
module LU is
process P1[A, B : any] is
    A; P2[A, B]
end process
process P2[A, B : any] is
    par A; P3[A, B] || B; null end par
end process
process P3[A, B : any] is
    A; P4[A, B]
end process
process P4[A, B : any] is
    i ; P3[A, B]
end process
process MAIN[A, B : any] is
    P1[A,B]
end process
end module
```

Il est à noter que cette combinaison de récursion et de parallélisation est occasionnée par la façon dont laquelle le cycle de vie des applications Android est modélisé et ainsi, la modification suggérée ne s'applique pas au code de l'application.

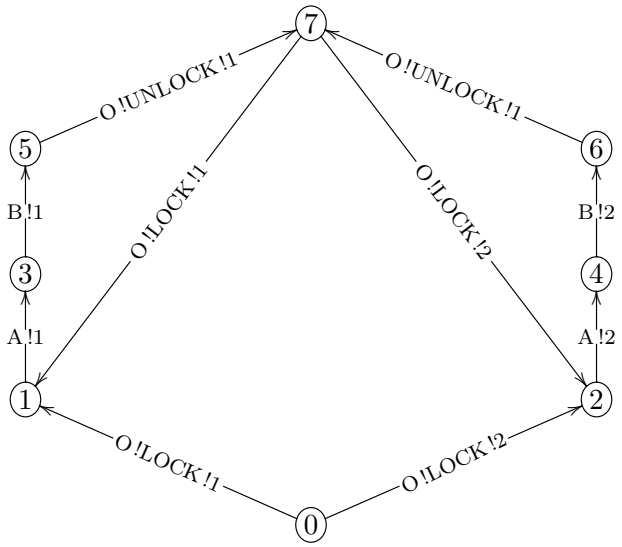

FIGURE 3.2 – Représentation graphique du LTS associé au processus MAIN de la synchronisation par verrous.

Les mécanismes de synchronisation

La majorité des applications Android utilisent des techniques de programmation concurrente, notamment les *threads*. La prise en compte des mécanismes de synchronisation est alors cruciale pour la génération d'un modèle précis. Puisque nous cherchons la présence de comportements malicieux, il nous serait très utile de considérer une approximation conservatrice envisageant tous les entrelacements possibles entre les actions. Cependant, cela pourrait introduire des comportements impossibles, pouvant conduire à de fausses évaluations des caractéristiques de sélection, soit à des faux positifs. En effet, si deux *threads* sont synchronisés, nous nous attendons à ce que leurs processus correspondants effectuent les actions sur lesquels ils synchronisent simultanément. Dans certains cas, la synchronisation peut ainsi réduire la taille du modèle généré. Lors de la génération du modèle LNT, Apk2Lnt prend en charge deux types de synchronisation : la synchronisation par verrous et les méthodes synchronisées. Nous nous sommes particulièrement concentrés sur ces routines de synchronisation car elles sont les plus utilisées. Il existe d'autres mécanismes de synchronisation, tels que la barrière de synchronisation et la synchronisation temporelle (*wait* et *notify*), qui sont laissés pour des travaux futurs.

Synchronisation par verrous Cette synchronisation est utilisée pour résoudre le problème de l'exclusion mutuelle. Essentiellement, à chaque appel d'une méthode de l'API Java ou Android relative à la synchronisation par verrous (à titre d'exemple les fonctions `lock()` et `unlock()` de la classe java.util.concurrent.locks.ReentrantLock), nous construisons une action qui concatène l'objet de la synchronisation ainsi que la méthode de synchronisation en question. Considérons deux *threads* P_1 et P_2 qui s'exécutent en parallèle et qui se synchronisent sur le

même objet o. Notamment, les méthodes `run()` de ces deux *threads* exécutent un bloc similaire à celui ci-dessous :

```
1 ReentrantLock o = new ReentrantLock();
2 try{
3     o.lock();
4     a_i;
5     b_i;
6 }finally{
7    o.unlock();
8 }
```

Les expressions a_i et b_i représentent des instructions Java gardées par l'objet o dans P_i, $i = 1, 2$. Pour le modèle LNT avec synchronisation, nous utilisons la spécification ci-dessous.

```
type M_ACTION is LOCK, UNLOCK end type
channel MONITOR is (M_ACTION, NAT) end channel
channel ACTION is (NAT) end channel

process P[O : MONITOR, A, B : ACTION](id : NAT) is
    loop
        O(LOCK, id);
        A(id);
        B(id);
        O(UNLOCK, id)
    end loop
end process

process MO[O : MONITOR] is
    loop
        var pid : NAT in
            O(LOCK, ?pid);
            O(UNLOCK, pid);
        end var
    end loop
end process

process MAIN[O : MONITOR, A, B : ACTION] is
    par O in
        MO[O]
    ||
```

```
        par
            P[O, A, B](1)
        ||
            P[O, A, B](2)
        end par
    end par
end process
```

Le système de transitions étiquetées résultant est illustré à la figure 3.2.

Les méthodes synchronisées En Java, une méthode est déclarée comme synchronisée lorsqu'elle est marquée par le mot clé « `synchronized` ».

```
1  class Hello {
2    synchronized void foo(){
3    // Le corps de foo
4   }
5 }
```

La spécification des méthodes synchronisées en LNT se fait de la même façon que la synchronisation par verrous, la seule différence réside dans l'objet de synchronisation qui devient l'instance de la classe où la méthode a été déclarée. Par conséquent, pour éviter de répéter le même code LNT de la synchronisation par verrous, nous ne citerons pas la spécification LNT des méthodes synchronisées. Dans cet exemple, le paramètre de synchronisation correspond à l'instance de la classe `Hello`. Lorsque la méthode `foo()` est invoquée dans la méthode `run()` d'un *thread*, ce dernier obtient une clé sur cette instance est ainsi aucun autre *thread* ne peut exécuter la méthode `run()` tant que cette instance n'est pas encore déverrouillée (la clé n'est pas libérée par le *thread*).

3.2 La spécification des comportements malicieux en μ-calcul modal

La majorité des approches de classification des applications Android utilisent les appels API comme caractéristiques. Il est toutefois possible que ce choix de caractéristiques soit accompagné par des taux de faux positifs élevés. Dans l'optique de détecter des applications Android contenant des comportements malicieux plus élaborés tout en minimisant le mieux possible le taux de faux positifs, nous optons pour l'utilisation des propriétés temporelles encodant des comportements potentiellement malicieux. En effet, au lieu de nous baser uniquement sur les appels API, nous considérons la relation entre ces appels en tant qu'indicateur de malveillance des comportements Android.

Afin de classifier les applications Android, deux ensembles de caractéristiques ont été utilisés. Le premier ensemble est construit en extrayant statiquement les méthodes de l'API Android protégées par des permissions. Ceci est réalisé en détectant la présence des appels aux méthodes de l'API nécessitant la détention d'une permission pour pouvoir être invoquées. Pour la correspondance (en anglais « *mapping* ») entre les méthodes de l'API et les permissions, nous avons utilisé les résultats de Pscout [16].
Le deuxième ensemble de caractéristiques comporte des formules en μ-calcul modal spécifiant des comportements Android potentiellement malicieux. Ces deux ensembles ont été utilisés par des algorithmes d'apprentissage automatique pour effectuer une classification binaire des applications Android. Le reste de cette section est consacré à la présentation de certaines caractéristiques ainsi que leurs spécifications en μ-calcul modal.

3.2.1 Les techniques d'évasion

La détection de l'environnement d'analyse est l'un des enjeux de l'analyse dynamique. En effet, dans le but de contourner les outils d'analyse dynamique les auteurs de logiciels malveillants exploitent la différence entre un émulateur et un appareil réel, et ce en ayant recours aux techniques de « *fingerprinting* » pour recueillir furtivement des informations à l'égard de l'environnement d'exécution. Une fois l'émulation détectée, le maliciel désactive les fonctionnalités liées aux comportements malveillants, échappant ainsi à la détection.

Pour détecter la virtualisation, les méthodes de la classe TelephonyManager peuvent être utilisées pour récupérer des informations pertinentes sur l'environnement d'exécution. À titre d'exemple, la valeur de *getdeviceId*() est nulle dans le cas d'un émulateur. Vidas et Christin [54] citent plusieurs méthodes de l'API Android qui renvoient des valeurs particulières lors de l'exécution sous un périphérique émulé. Un logiciel malveillant évasif tente de détecter la différence entre les émulateurs et les appareils réels en vérifiant ces valeurs à l'aide des méthodes de comparaison des chaînes de caractères. Ensuite, il dissimule le comportement malveillant en restant inactif, ce qui peut être fait à l'aide de la classe javautilTimer. Le comportement évasif peut être spécifié à l'aide de la formule suivante :

DormantFunctionality = **mu** X.((DetectionofVirtualization **and** φ2) **or** ⟨true⟩(X))

DetectionOfVirtualization = **mu** X.((⟨androidosBuildgetRadioVersion⟩true **or** ⟨androidtelephonyTelephonyManagergetDeviceID⟩true **or** ⟨androidtelephonyTelephonyManagergetNetworkCountryIso⟩true **or** ⟨androidtelephonyTelephonyManagergetLine1Number⟩true **or** ⟨androidtelephonyTelephonyManagergetNetworkType⟩true **or** ⟨androidtelephonyTelephonyManagergetNetworkOperator⟩true **or** ⟨androidtelephonyTelephonyManagergetPhoneType⟩true **or** ⟨androidtelephonyTelephonyMangergetSimCountryIso⟩true **or** ⟨androidtelephonyTelephonyManagergetSimSerialNumber⟩true **or** ⟨androidtelephonyTelephonyManagergetSubscriberId⟩true **or** ⟨androidtelephonyTelephonyManagergetVoice-

MailNumber⟩true) **and** $\varphi 1$) **or** ⟨true⟩(X))

$\varphi 1$ = **mu** X.((⟨javalangStringEquals⟩true **or** ⟨javalangStringcontains⟩true **or** ⟨ javalangStringstartsWith⟩true) **or** ⟨true⟩(X))

$\varphi 2$ = **mu** X.((⟨javautilTimerschedule⟩true **or** ⟨javautilTimercancel⟩true **or** ⟨javautilTimerscheduleAtFixedRate⟩true) **or** ⟨true⟩(X))

3.2.2 Le bloquage des SMS entrants

Les chevaux de Troie mobiles comme la famille Opfake sont connus pour leurs abonnements aux services Premium SMS. Ils envoient des messages SMS à des numéros surtaxés sans le consentement des utilisateurs, puis interceptent et bloquent les messages SMS entrants, pour que l'utilisateur ne soit pas averti. À l'aide de la méthode androidtelephonygsmSmsManagerSendTextMessage, le maliciel envoie un SMS à un numéro payant. Ensuite, il surveille les messages entrants en vérifiant que la valeur de androidcontentIntentgetAction est égale à « android.provider.Telephony.SMS_RECEIVED » finalement, il bloque le SMS à l'aide de androidcontentBroadcastReceiverabortBroadcast.

BlockIncomingSms = **mu** X.((⟨androidtelephonygsmSmsManagerSendTextMessage⟩ true **and** $\psi 1$) **or** ⟨true⟩(X))

$\psi 1$ = **mu** X.((⟨androidcontentIntentgetAction⟩true **and** $\psi 2$) **or** ⟨true⟩(X))

$\psi 2$ = **mu** X.((⟨javalangStringequals⟩true **and** $\psi 3$) **or** ⟨true⟩(X))

$\psi 3$ = **mu** X.(⟨androidcontentBroadcastReceiverabortBroadcast⟩true **or** ⟨true⟩(X))

3.2.3 Les rançogiciels

Les rançongiciels Android posent une menace sérieuse à la sécurité des smartphones. Ce type de programme malveillant est en mesure de compromettre les données de l'utilisateur. Ces maliciels ciblent l'utilisateur et procèdent généralement soit en verrouillant l'appareil de la victime ou en cryptant ses données. Les attaques par rançogiciels ont généralement pour objectif de contraindre les victimes à payer des rançons afin de regagner l'accès à leurs données.

Pour le verrouillage de l'appareil, les auteurs des rançogiciels peuvent utiliser la méthode androidappadminDevicePolicyManagerlockNow pour verrouiller l'appareil et faire en sorte que l'affichage passe en mode veille.

LockTheDevice = **mu** X.((⟨androidcontentContextgetSystemService⟩true **and** $\theta 1$) **or** ⟨true⟩(X))

$\theta 1$ = **mu** X.(⟨androidappadminDevicePolicyManagerlockNow⟩true **or** ⟨true⟩(X))

Le deuxième type de rançogiciel, appelé crypto-rançogiciel, fait usage de la cryptographie pour empêcher l'utilisateur d'accéder à ses données. Ce type de maliciel accède aux répertoires de stockage, puis chiffre tous les fichiers contenus dans ces répertoires. Le cryptage peut être effectué à l'aide des méthodes de l'API Android dédiées à la cryptographie. Après avoir crypté les données, le maliciel supprime les fichiers d'origine et ne conserve que leur version cryptée.

UsesCryptoApis = **mu** X.((⟨javaxcryptoCiphergetInstance⟩true **or** ⟨javaxcryptoCipherdoFinal⟩true **or** ⟨javaxcryptospecDESKeySpecDESKeySpec⟩true **or** ⟨javaxcryptospecSecretKeySpec⟩true **or** ⟨javasecurityMessageDigestgetInstance⟩true **or** ⟨javasecurityMessageDigestupdate⟩true **or** ⟨javasecurityMessageDigestdigest⟩true) **or** ⟨true⟩(X))

ExternalStorageEncryption = **mu** X.((⟨androidosEnvironmentgetExternalStorageDirectory⟩true **and** $\theta 2$) **or** ⟨true⟩(X))

$\theta 2$ = **mu** X.(⟨javaioFileFile⟩true **and** $\theta 3$ **or** ⟨true⟩(X))

$\theta 3$ = **mu** X.((⟨javaioFileInputStream⟩true **and** $\theta 4$) **or** ⟨true⟩(X))

$\theta 4$ = **mu** X.((⟨javaioInputStreamread⟩true **and** $\theta 5$) **or** ⟨true⟩(X))

$\theta 5$ = **mu** X.(⟨javaxcryptoCipherOutputStream⟩true **or** ⟨true⟩(X))

DeleteExternalStorageData = **mu** X.((⟨javaioFileFile⟩true **and** $\theta 6$) **or** ⟨true⟩(X))

$\theta 6$ = **mu** X.((⟨androidosEnvironmentgetExternalStorageDirectory⟩true **and** $\theta 7$) **or** ⟨true⟩(X))

$\theta 7$ = **mu** X.(⟨javaioFiledelete⟩true **or** ⟨true⟩(X))

Ainsi, pour spécifier les comportements de rançogiciels, nous proposons la formule suivante :

Ransomware = ((UsesCryptoApis **or** ExternalStorageEncryption) **and** Deleteexternalstoragedata) **or** LockTheDevice

3.2.4 Les logiciels espions (Spyware)

Il existe un bon nombre de logiciels espions sur Android. Le caractère espion de ces maliciels peut prendre plusieurs formes. Certains tracent la localisation géographique, d'autres écoutent et enregistrent les appels téléphoniques. L'enregistrement des vidéos et la prise des photos à

l'insu de l'utilisateur sont, entre autres, des comportements malicieux caractérisant les logiciels espions sur Android.

Comme expliqué dans [49], un appel de la méthode androidhardwareCameraTakePicture non précédé par androidhardwareCameraSetPreviewDisplay ou androidhardwareCameraSetPreviewTexture prendra une photo sans que l'utilisateur ne s'en rende compte. De même, en n'appelant pas la méthode androidmediaMediaRecorderSetPreviewDisplay, l'utilisateur ne sera pas averti lors de l'enregistrement de vidéos ou des audios.

TakingPictures = **mu** X.((⟨androidhardwareCameraopen⟩true **and** ω1) **or** ⟨true⟩(X))

ω1 = **mu** X.((⟨androidhardwareCameraTakePicture⟩true) **or** ((**not** (⟨androidhardwareCameraSetPreviewDisplay⟩true **or** ⟨androidhardwareCameraSetPreviewTexture⟩true)) **and** ⟨true⟩(X)))

BackgroundRecording = **mu** Y.((⟨androidmediaMediaRecorderSetAudioSource⟩true **or** ⟨androidmediaMediaRecorderSetVideoSource⟩true) **or** ([androidmediaMediaRecorderSetPreviewDisplay]false **and** ⟨true⟩(Y)))

3.2.5 Le code natif

Le développement natif est possible sous Android. Afin d'améliorer les performances des applications, le Kit de développement natif Android (NDK) [11] et JNI peuvent être utilisés pour compiler et exécuter du code en C et C ++. Certes, le code natif n'est pas malveillant en soi, cependant, il peut être utilisé pour dissimuler le code malveillant. Ainsi, l'exécution du code natif constitue une menace pour la sécurité sur Android. Au niveau de l'API Android, les applications Android peuvent charger des bibliothèques natives à l'aide de la méthode javalangSystemloadLibrary. De la même manière, javalangProcessBuilderstart et javalangRuntimeexec peuvent être utilisés pour exécuter du code natif. Par conséquent, pour détecter l'utilisation du code natif, nous utilisons la formule suivante :

NativeCode = **mu** X.((⟨javalangSystemloadLibrary⟩true **or** ⟨javalangRuntimeexec⟩true **or** ⟨javalangProcessBuilderstart⟩true) **or** ⟨true⟩(X))

3.2.6 Les techniques d'obscurcissement de code

Les cybercriminels sont en quête permanente de techniques permettant de défier la détection de maliciels. L'une des techniques couramment utilisées pour échapper à la détection est la transformation du code, notamment en réécrivant les logiciels malveillants existants dans le but de rendre l'analyse statique difficile. Cela peut être réalisé par des techniques d'obscurcissement de code, qui offrent la possibilité de changer la syntaxe du programme tout en

préservant sa sémantique. Bien que l'obscurcissement de code vise également à protéger la propriété intellectuelle des programmes, les auteurs de programmes malveillants en profitent pour contourner les outils de détection de maliciels. Dans cet esprit, nous avons inclus l'utilisation de l'obscurcissement de code dans nos propriétés. Rastogi et al. [48] expliquent les techniques de transformation du code utilisées pour le rendre plus difficile à analyser. La réflexion et le chargement de code dynamique en font partie.

La réflexion

En Java, grâce à la réflexion on peut introspecter les classes, les interfaces, les champs et les méthodes lors de l'exécution. L'invocation de méthodes par réflexion complique l'analyse des méthodes invoquées et ainsi représente un défi majeur pour l'analyse statique. Dans le but de détecter les appels de méthode par réflexion, nous utilisons la formule suivante :

Reflection = **mu** X.((⟨javalangClassforName⟩true **and** $\psi 1$) **or** ⟨true⟩(X))

$\psi 1$ = **mu** X.(⟨javalangClassgetMethod⟩true **or** ⟨true⟩(X))

Le chargement dynamique du code

La plateforme Android offre la possibilité de charger dynamiquement du code pendant l'exécution. Poeplau et al. [46] soulignent les risques liés à l'utilisation du chargement de code dynamique (DCL). C'est non seulement utile pour les auteurs de maliciels dans la mesure où ça leur permet d'échapper à la détection, mais l'utilisation inappropriée de DCL rend également les applications bénignes vulnérables aux attaques par injection de code. Les méthodes DexClassLoader et ClassLoader sont utilisées pour effectuer un chargement dynamique de classes.

Dynamicclassloading = **mu** X.((⟨dalviksystemDexClassLoaderDexClassLoader⟩true **or** ⟨javalangClassLoaderClassLoader⟩true) **or** ⟨true⟩(X))

3.2.7 La vérification des adresses IP

Certains logiciels malveillants sous Android collectent des informations sensibles sur le périphérique de la victime, les informations sur les réseaux en font partie. En effet, le cheval de Troie Acnetdoor envoie l'adresse IP de la victime à un serveur distant après avoir ouvert une porte dérobée sur le périphérique infecté [1].

CheckIpAddress = **mu** X.((⟨javanetNetworkInterfacegetNetworkInterfaces⟩true **and** $\omega 1$) **or** ⟨true⟩(X))

$\omega 1$ = **mu** X.((((⟨javautilEnumerationhasMoreElements⟩true **or** ⟨javanetNetworkInterfacegetInetAddresses⟩true)**and** $\omega 2$) **or** ⟨true⟩(X))

$\omega 2$ = **mu** X.((javanetInetAddressisLoopbackAddress⟩true **or** ⟨javanetInetAddressget-HostAddress⟩true) **or** ⟨true⟩(X))

3.2.8 Les requêtes DNS

La communication réseau joue un rôle important dans le fonctionnement des maliciels. En se basant sur un ensemble de donnée constitué de 24650 applications malveillantes regroupées en 135 types, Wei et al. [56] observent que 90% de ces applications malicieuses ainsi que 64% des types de maliciels, dérivés des comportements des applications malicieuses, se servent des serveurs de commande et contrôle C&C. Les logiciels malveillants Android ont plusieurs stratégies pour communiquer avec des serveurs distants. En plus des adresses IP codées en dur, des requêtes DNS peuvent être utilisées pour établir une connexion avec le serveur C&C. En effet, les maliciels effectuent des requêtes DNS pour obtenir l'adresse IP du serveur distant. Sous Android, les méthodes getByName et getAllByName de la classe java.net.InetAddress peuvent être utilisées pour effectuer des recherches DNS.

DnsLookups = **mu** X.(⟨javanetInetAddressgetByName⟩true **or** ⟨javanetInetAddressget-AllByName⟩true **or** ⟨true⟩(X))

Conclusion

Dans ce chapitre, nous avons présenté le processus de modélisation des applications Android ainsi que les propriétés temporelles sur lesquelles se basent les classifieurs. En effet, en se basant sur Apk2Lnt, un programme java assurant l'extraction du modèle, les applications Android sont exprimées en termes de processus LNT. Dans le but de vérifier si une application Android donnée exhibe des comportements malicieux, ces derniers ont été exprimés par des formules en μ-calcul modal avant d'être soumis au vérificateur de modèle evaluator4. À cet effet, les comportements malicieux des applications Android, tel que les comportements évasifs, ainsi que leur spécification en mu-calcul modal ont été présentés dans la deuxième partie du chapitre.

Chapitre 4

La détection des maliciels par apprentissage automatique

L'apprentissage automatique est une branche de l'intelligence artificielle qui vise à résoudre des tâches sans avoir à programmer explicitement l'ensemble des instructions menant à la solution. En effet, les algorithmes d'apprentissage automatique visent la construction de modèles permettant d'effectuer des prédictions en se basant sur un jeu de données.

Les techniques d'apprentissage automatique sont utilisées pour résoudre une multitude de problèmes qui touchent différents domaines. Entre autres, le problème de détection de maliciels Android peut être formulé comme un problème de classification. Les techniques d'apprentissage automatique peuvent se décliner en deux catégories : l'apprentissage supervisé et l'apprentissage non supervisé.

Dans le présent chapitre nous présentons brièvement des techniques d'apprentissage automatique supervisé ainsi que les métriques permettant de choisir parmi plusieurs modèles le modèle menant à la solution optimale du problème étudié. En l'occurrence, nous cherchons le classifieur optimal permettant de prédire si une application donnée est malicieuse ou non. Par ailleurs, nous allons présenter les résultats de la classification.

4.1 L'apprentissage supervisé

Les méthodes d'apprentissage automatique supervisé visent à apprendre une fonction de prédiction à partir d'exemples préalablement étiquetés. Il s'agit d'algorithmes qui se basent sur des observations antérieures pour prédire la solution d'un problème dont on ne connaît pas la réponse.

Formellement, les techniques d'apprentissage supervisé utilisent un jeu de données D sous

forme de couples (x_i, y_i) où x_i est un vecteur d'entrée et y_i est la valeur qu'on veut prédire.

$$D = \{(x_i, y_i) \mid i \in \{1 \dots n\}\} \tag{4.1}$$

À partir de ce jeu de données, l'algorithme cherche à construire une fonction f qui représente bien D, soit un modèle, qui associe à chaque vecteur entrée x une valeur y.

$$y = f(x) \tag{4.2}$$

Dépendamment du type de la valeur de sortie y, on distingue trois sous-catégories d'apprentissage supervisé : la classification, la régression et la prédiction structurée.

La classification
Dans les problèmes de classification (ou discrimination), la valeur de sortie y est une valeur qualitative. À titre d'exemple, à partir d'une base de données d'images de chiens et de chats, on veut savoir si l'animal en photo correspond à un chien ou à un chat.

La régression
En régression, la valeur de sortie y est une valeur quantitative. Par exemple, on veut prédire le prix d'une maison ou le salaire d'un individu.

La prédiction structurée
On parle de prédiction structurée si y prend une valeur dans un ensemble de données sctructurées. Par exemple, on veut localiser dans une image la région correspondant aux visages. Dans ce cas la valeur de sortie correspond à un sous-ensemble de pixels de l'image.

4.2 Le processus de la classification

Le processus de sélection du modèle décisionnel approprié au problème étudié est un processus itératif à plusieurs étapes [37]. Dans cette section, nous allons décrire les différentes étapes de ce processus.

Comme le montre la figure 4.1, la première étape de la résolution d'un problème de classification correspond à la collecte des données. Cette étape est très importante car la qualité et la quantité des données collectées influencent significativement la qualité du modèle prédictif.

Les données collectées sont souvent représentées par des vecteurs avant d'être utilisées par un algorithme de classification. Dans le cadre de notre projet, on vise la résolution d'un problème de classification binaire. Chaque instance du jeu de données correspond à un vecteur dont le dernier élément est la variable y qu'on veut prédire. C'est une variable qualitative qui peut prendre deux valeurs : 0 ou 1. La valeur 1 correspond à l'étiquette de la classe positive, la classe des applications malicieuses. La valeur 0 correspond à l'étiquette de la classe négative, la classe des applications non malicieuses.

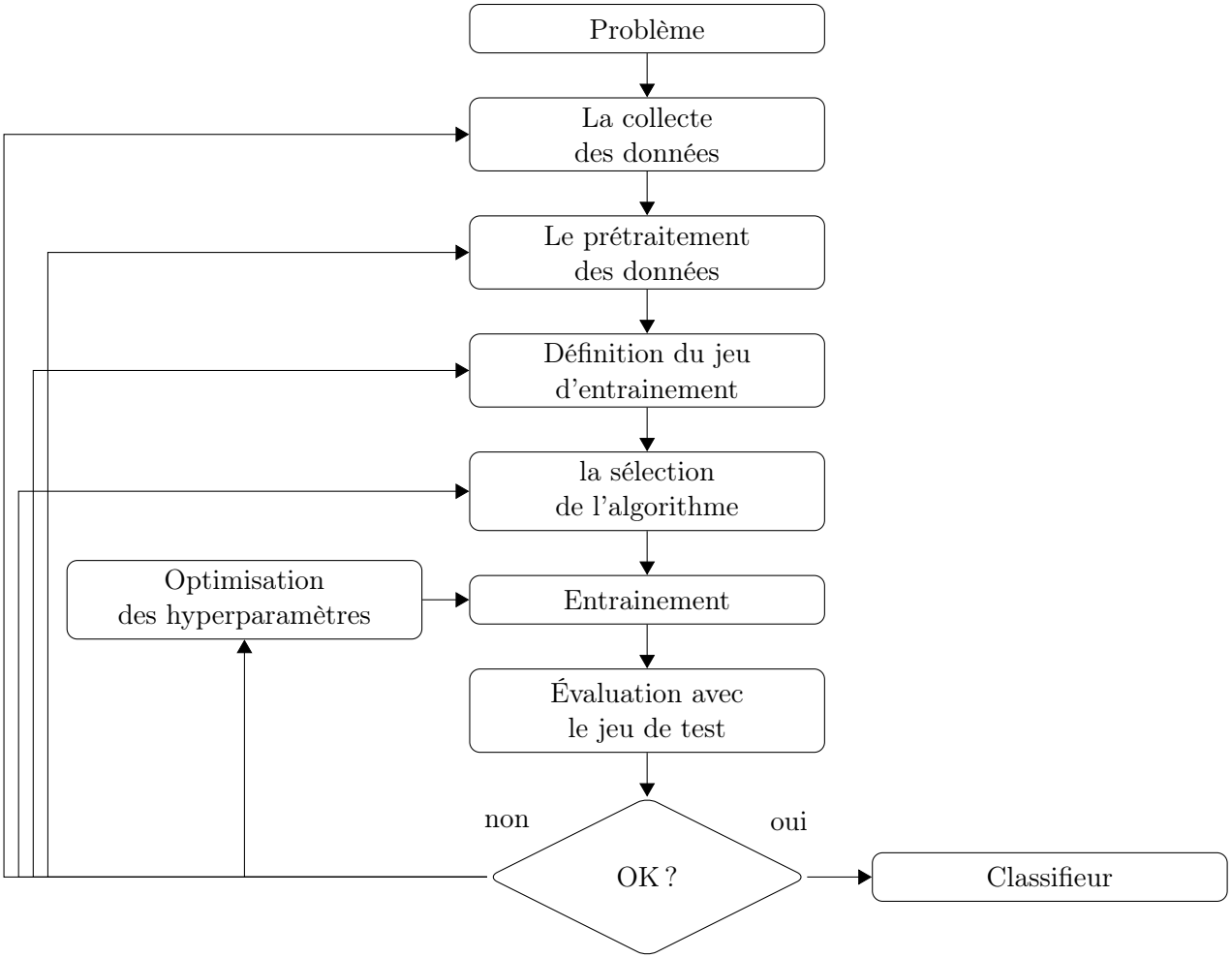

FIGURE 4.1 – Le processus d'apprentissage supervisé.

Les autres éléments du vecteur représentent les variables explicatives, qui fournissent des informations sur le problème étudié. Ultérieurement, les valeurs de ces variables vont être utilisées par le modèle prédictif pour obtenir une estimation de la valeur de la variable à expliquer. Dans ce travail, ces variables correspondent à la présence d'une méthode de l'API Android ou à la satisfaction de certaines propriétés temporelles.

Les données reçues en entrée par les algorithmes d'apprentissage automatique nécessitent généralement des prétraitements. En effet, certains algorithmes d'apprentissage automatique tel les SVMs, nécessitent l'utilisation de données normalisées, en d'autres termes des données centrées (moyenne = 0) et réduites (écart-type = 1).

Bien souvent, il arrive que les instances de données soient incomplètes. Une façon simplifiée de traiter le manque des valeurs d'une ou de plusieurs variables est la suppression des instances incomplètes. Cette méthode consiste à ignorer les instances incomplètes. Une approche alternative est l'imputation des données. Cette approche complète les données en attribuant des

valeurs de remplacement aux données manquantes.

La sélection du modèle optimal passe par l'évaluation et la comparaison de plusieurs modèles prédictifs. Pour chaque algorithme d'apprentissage supervisé, on fournit un jeu de données réparti en deux sous-ensembles : le jeu d'entraînement et le jeu de test.

La construction d'un modèle prédictif se fait à partir d'un jeu de données, notamment à partir d'un ensemble d'observations dont les valeurs de la variable expliquée sont connues. Cet ensemble est appelé le jeu d'entraînement. En plus du jeu d'entraînement, l'algorithme de classification prend en entrée les valeurs des hyperparamètres. Ces derniers affectent la performance du classifieur, d'où la nécessité de bien choisir les valeurs de ces hyperparamètres. Il s'agit des paramètres de l'algorithme dont la valeur est fixée avant le début de l'apprentissage. Le processus de sélection de la valeur optimale de chaque hyperparamètre, appelé optimisation des hyperparamètres, est souvent réalisé en effectuant une validation croisée. En explorant plusieurs valeurs d'hyperparamètres, on calcule la mesure de performance moyenne. La valeur d'hyperparamètre choisie est celle qui donne la meilleure performance.

Une fois construit, la capacité de généralisation du modèle est évaluée en utilisant le jeu de test. Si la performance du modèle n'est pas satisfaisante, on revisite les étapes précédentes du processus de sélection du modèle. Une mauvaise performance peut être due à un choix non pertinent d'hyperparamètres, à un jeu de données trop petit, comme elle peut être occasionnée par le mauvais choix de l'algorithme d'apprentissage.

4.3 Les modèles étudiés

Le choix du classifieur adéquat au problème étudié est effectué en entraînant plusieurs classifieurs et en comparant leurs performances. Dans cette optique, nous avons évalué la performance de quatre différents modèles, à savoir les k-plus proches voisins (KNN, de l'anglais « *K-Nearest Neighbors* »), les séparateurs à vastes marges (SVM, de l'anglais « *Support Vector Machine* »), les forêts aléatoires (RF, de l'anglais « *Random Forest* »), et le *gradient boosting* (noté GB).

4.3.1 Les k-plus proches voisins

L'algorithme des k-plus proches voisins [25] est un algorithme d'apprentissage automatique non paramétrique qui construit des modèles prédictifs pour la classification et la régression en se basant sur la similarité. Une méthode d'apprentissage est dite non paramétrique lorsqu'elle ne fait aucune hypothèse sur la distribution des données.

Pour prédire la classe d'un échantillon non vu dans le jeu de données, l'algorithme se base sur des fonctions de calcul de distance telles que la distance euclidienne et la distance de Min-

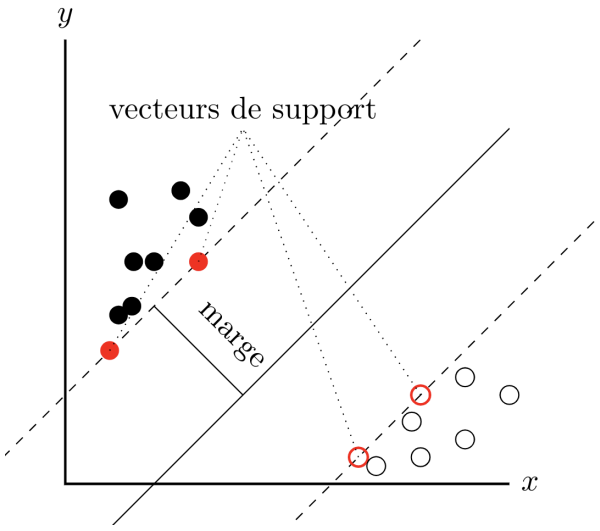

FIGURE 4.2 – Exemple de classification de données linéairement séparables.

kowsky [55] pour localiser les k instances du jeu de données les plus proches de cet échantillon. L'étiquette attribuée à ce dernier correspond au mode des étiquettes de ces k instances.

4.3.2 Séparateurs à Vastes Marges

Les séparateurs à vastes marges [24] sont une technique d'apprentissage automatique initialement conçue pour résoudre des problèmes de classification binaire. L'algorithme SVM se base sur la recherche d'un hyperplan optimal permettant de séparer les deux classes. Il existe une multitude d'hyperplans valides permettant la séparation de données, toutefois l'algorithme calcule, en se basant sur un ensemble de points, l'hyperplan dont la distance minimale aux exemples d'apprentissage est maximale. Ce choix se base sur un sous ensemble de points d'entraînement appelés des vecteurs de support. Comme le montre la figure 4.2, les vecteurs de support sont les points d'entraînement les plus proches de la frontière.

En général plus la distance entre l'hyperplan et les exemples (la marge) est élevée, plus la capacité de généralisation du classifieur est grande.

Dans le cas des données non linéairement séparables, une transformation non linéaire des données est réalisée via des fonctions appelées fonctions de noyau pour représenter les données dans un espace de plus grande dimension permettant une séparation linéaire des données.

4.3.3 Forêt d'arbres aléatoires

Les forêts aléatoires [19] sont une technique d'apprentissage ensembliste qui repose sur la combinaison des prédictions de plusieurs classifieurs dans le but de réduire la variance et le biais des classifieurs individuels. Notamment, l'algorithme des forêts aléatoires propose une

optimisation des arbres de décision. Il construit plusieurs arbres de décision sur des sous-ensembles aléatoires de données tirées de la base d'apprentissage.

Afin de prédire l'étiquette d'une observation, chaque arbre prédit une étiquette. Le classifieur des forêts aléatoires choisit l'étiquette la plus dominante, à savoir l'étiquette de la classe ayant obtenu la majorité des votes.

4.3.4 Le *gradient boosting*

Le *gradient boosting* [28] est une méthode ensembliste qui combine de manière itérative plusieurs modèles simples entraînés sur différentes versions de données.

L'apprentissage est effectué d'une manière séquentielle afin que les modèles ajoutés apprennent des erreurs commises par les modèles précédents. À chaque itération, on vise à focaliser l'apprentissage sur les observations mal classées à l'étape précédente et ce en attribuant des pondérations plus élevée à ces derniers. C'est un modèle additif par étape qui vise à minimiser la fonction de perte.

4.4 L'évaluation de l'apprentissage

Les performances des modèles prédictifs sont évaluées par un ensemble de métriques calculées sur le jeu de test. Dans cette section, nous allons présenter le jeu de données utilisé ainsi que les métriques d'évaluation considérées lors de la sélection du modèle.

4.4.1 Le jeu de données

Pour l'entraînement et l'évaluation des algorithmes d'apprentissage automatique, nous avons utilisé un jeu de données de 5009 applications Android, d'une taille pouvant aller jusqu'à deux mégaoctets. Ce jeu de données est tiré d'Androzoo [13], une collection d'applications Android recueillies à partir de plusieurs plateformes de téléchargement tels Google Play et ses alternatifs. La provenance des applications colléctées dans Androzoo est détaillée dans le tableau 4.1. Notamment, dans notre ensemble de données, les applications ont été collectées à partir de Google Play, d'Appchina, d'Anzhi et du jeu de données du projet Génome. Les applications malicieuses utilisées dans le cadre de notre étude représentent 63% de notre ensemble de données et ont été identifiées comme malicieuses par au moins 10 antivirus de VirusTotal [7].

4.4.2 Les métriques de performance

Le but de la classification étant de prédire l'étiquette des nouvelles données (non préalablement vues par le modèle), dans le cadre d'une classification binaire le résultat de la prédiction peut prendre quatre formes :

TABLE 4.1 – Les sources des applications collectées dans Androzoo [2].

Source	Nombre d'applications
play.google.com	7,397,979
PlayDrone	1,445,205
anzhi	832,997
appchina	771,970
VirusShare	115,956
mi.com	113,583
1mobile	57,530
angeeks	55,818
slideme	52,467
fdroid	18,304
praguard	10,186
torrents	5,294
freewarelovers	4,145
proandroid	3,683
hiapk	2,512
genome	1,247
apk_bang	363
non connue	165

— Un vrai positif, lorsqu'un échantillon de la classe positive reçoit une étiquette positive.

— Un faux positif, lorsqu'un échantillon de la classe négative reçoit une étiquette positive.

— Un vrai négatif, lorsqu'un échantillon de la classe négative reçoit une étiquette négative.

— Un faux négatif, lorsqu'un échantillon de la classe positive reçoit une étiquette négative.

Dans le cadre de notre étude, la classe positive correspond à la classe des applications malicieuses et la classe négative correspond à celle des applications non malicieuses. Dans ce qui suit, TP désigne le nombre de vrais positifs, FP désigne le nombre de faux positifs, TN désigne le nombre de vrais négatifs et FN désigne le nombre de faux négatifs. Ces quatre cas sont représentés par la matrice de confusion illustrée par la table 4.2. Pour évaluer la performance des algorithmes de classification, nous avons utilisé plusieurs mesures de performances, à savoir l'exactitude, le taux de faux positif le taux de faux négatifs, le taux de vrais positifs, le taux de vrais négatifs et la courbe ROC (de l'anglais « *Receiver Operating Characteristic* ») que nous expliquerons ci-dessous.

Le taux de vrais positifs

Le taux de vrais positifs (TPR, de l'anglais « True Positive Rate »), également appelé rappel positif ou sensibilité, mesure la proportion des applications malicieuses correctement identifiées comme malicieuses par le classifieur.

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP+FN}} \tag{4.3}$$

TABLE 4.2 – Matrice de confusion.

		Classe estimée par le classifieur	
		Application malicieuse	Application non malicieuse
Classe réelle	Application malicieuse	Vrai positif (TP)	Faux négatif (FN)
	Application non malicieuse	Faux positif (FP)	Vrai négatif (TN)

Le taux de vrais négatifs

Le taux de vrais négatifs (TNR, de l'anglais « True Negative Rate »), également appelé rappel négatif ou spécificité, mesure la proportion d'applications bénignes correctement identifiées comme non malicieuses par le classifieur.

$$\text{TNR} = \frac{\text{TN}}{\text{TN}+\text{FN}} \tag{4.4}$$

Le taux de faux positifs

Le taux de faux positifs (FPR, de l'anglais « False Positive Rate ») mesure la proportion d'applications non malicieuses classées comme malicieuses par le classifieur.

$$\text{FPR} = \frac{\text{FP}}{\text{FP}+\text{TN}} \tag{4.5}$$

Le taux de faux négatifs

Le taux de faux négatifs (FNR, de l'anglais « False Negative Rate ») mesure la proportion d'applications malicieuses non détectées par le classifieur.

$$\text{FNR} = \frac{\text{FN}}{\text{FN}+\text{TP}} \tag{4.6}$$

L'exactitude

L'exactitude est le rapport entre le nombre d'échantillons prédits correctement et le nombre total de prédictions réalisées. Cette métrique mesure la proportion d'applications correctement classées par le classifieur.

$$\text{Exactitude} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4.7}$$

La F1-Mesure

La F1-Mesure est la moyenne harmonique de la précision et du rappel.

$$\text{F1} = 2 \times \frac{\text{PPV} \times \text{TPR}}{\text{PPV} + \text{TPR}} \tag{4.8}$$

Où PPV désigne la précision ou la valeur prédictive positive.

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.9}$$

La valeur des mesures citées ci-dessus varie entre 0 et 1. Plus la valeur est proche de 1, mieux est la performance.

La courbe ROC
La courbe ROC (de l'anglais « *Receiver Operating Characteristic* ») illustre la relation qui existe entre la sensibilité et la spécificité d'un test discriminant pour chaque valeur seuil considérée. C'est la représentation graphique du taux de vrais positifs en fonction du taux de faux négatifs. L'aire sous la courbe est un indicateur du pouvoir discriminant du modèle. Si l'aire sous la courbe est égale à 1 (AUC = 1), la séparation entre les deux classes est parfaite.

4.5 Résultats

Comme mentionné à la section 4.2, en apprentissage automatique supervisé le jeu de données est divisé en deux sous-ensembles : un jeu d'entraînement et un jeu de test.

Pour ce faire, nous avons divisé le jeu de données en deux parties disjointes : le jeu d'entraînement, représentant 67% du total des échantillons, est utilisé pour la construction du modèle d'apprentissage, tandis que les échantillons restants, soit 33% de l'ensemble de données, sont utilisés pour évaluer la performance du modèle sur de nouvelles données non vues au cours de l'apprentissage. L'échantillonnage est effectué de manière stratifiée afin de garantir que la proportion d'applications non malicieuses ainsi que celle d'applications malicieuses soit la même dans le jeu d'entraînement et le jeu de test.

Dans le processus de sélection du modèle décisionnel approprié, l'optimisation des hyperparamètres est une étape importante. En effet, le choix des hyperparamètres a un impact significatif sur les performances du modèle. Afin de choisir la valeur optimale d'un hyperparamètre, la validation croisée à 5-plis sur le jeu d'entraînement est utilisée. Cette technique consiste à diviser le jeux d'entraînement en 5 plis de tailles égales. À la première itération, le premier pli est utilisé pour tester les performances du modèle tandis que les quatre derniers plis sont utilisés pour l'entraînement du modèle. Pour la deuxième itération, le deuxième pli est utilisé pour évaluer le modèle et les plis restant constituent le nouveau jeu d'entraînement. Ce processus est répété jusqu'à ce que chacun des cinq plis ait été utilisé comme un jeu de test. En effectuant une recherche aléatoire explorant les combinaisons possibles de valeurs d'hyperparamètres citées dans la table 4.3, nous avons sélectionné la combinaison d'hyperparamètres rapportant la F1-mesure moyenne optimale. Le choix de la F1-mesure comme mesure de performance durant l'optimisation des hyperparamètres est expliqué par le débalancement des classes.

Les algorithmes d'apprentissage supervisé présentés à la section 4.3 sont utilisés pour décider si une application Android est malicieuse. Le processus de construction et d'évaluation des modèles décisionnels, a été mené en utilisant deux ensembles de caractéristiques ou de variables explicatives. Le premier ensemble contient uniquement la présence ou non des appels de l'API

TABLE 4.3 – Les valeurs utilisées lors de l'optimisation des hyperparamètres

Classifieur	Hyperparamètre	Valeurs
Random Forest	n_estimators	{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500}
	max_depth	{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, None}
SVM	gamma	{0.01, 0.1, 1,10, 100}
	C	{0.001, 0.01, 0.1, 2, 10, 100}
KNN	n_neighbors	{5, 7, 9, 11, 13, 15, 17, 19, 21, 25, 31, 35, 41, 45, 51, 55, 61, 66, 71, 77, 80, 85, 91, 99}
GBC	n_estimators	{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000}

TABLE 4.4 – Les résultats de la classification en utilisant les appels API

	l'exactitude %	FNR %	FPR %	TPR %	TNR %	F1-score %
RF	94.29	5	6	95	94	95.72
SVM	92.76	6	9	94	91	94.59
KNN	92.39	7	10	93	90	94.31
GBC	93.07	6	9	94	91	94.82

TABLE 4.5 – Les résultats de la classification en utilisant les appels API et les propriétés temporelles

	l'exactitude %	FNR %	FPR %	TPR %	TNR %	F1-score %
RF	94.60	5	6	95	94	95.94
SVM	92.94	5	11	95	89	94.77
KNN	93.62	5	9	95	91	95.24
GBC	94.17	5	7	95	93	95.64

Android protégés par des permissions, le deuxième ensemble, quant à lui, en rajoute au premier des variables explicatives relatives à la satisfaction des formules structurées formulées en μ-calcul modal.

La représentation vectorielle des applications dans l'espace des caractéristiques peut induire des redondances de données. En effet, certaines applications Android, malgré leurs différences, peuvent être représentées par le même vecteur, introduisant ainsi des redondances dans le jeu de données. Afin de s'assurer que cela n'affecte pas la capacité de généralisation des classifieurs, la répartition du jeu de données est faite de sorte que le jeu d'entraînement et celui du test soient disjoints.

La table 4.4 présente les résultats de la classification sur le jeu de test. Ces résultats sont obtenus en utilisant le premier ensemble de caractéristiques et englobent l'exactitude, le taux

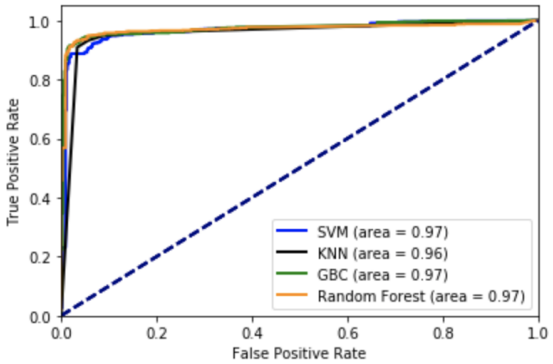

FIGURE 4.3 – Receiver operating characteristic curves

de faux négatifs, le taux de faux positifs, le taux de vrais positifs, le taux de vrais négatifs et la F1-mesure.

Dans la deuxième expérimentation, en plus des appels de l'API Android protégés par les permissions, les propriétés temporelles encodant des comportements Android potentiellement malicieux sont également utilisées par les classifieurs. Comme le montre la table 4.5, il y a une légère amélioration par rapport au premier ensemble de caractéristiques. Nous notons une diminution du taux de faux négatifs et du taux de faux positifs ainsi q'une amélioration de la F1-mesure.

Pour une comparaison plus poussée, nous utilisons la courbe ROC. La surface sous la courbe mesure la capacité discriminative du modèle. En fait, plus la courbe ROC se rapproche du coin supérieur gauche, plus le modèle est apte à différencier entre les applications Android bénignes et celles malicieuses. La figure 4.3 montre la courbe ROC correspondante à chaque classifieur utilisant le deuxième ensemble de caractéristiques.

Les résultats montrent que le meilleur classifieur est celui des forêts aléatoires. Ce dernier surpasse les autres algorithmes d'apprentissage avec une exactitude de $94,60\%$ et une F1-mesure de $95,94\%$.

La collecte des caractéristiques structurées est effectuée en deux phases : la modélisation des applications Android et la vérification formelle du modèle.

Concernant la durée de la première phase, l'extraction du modèle formel est effectuée en quelques secondes pour certaines applications, en minutes pour d'autres, avec une moyenne de $62,88$ secondes par application. De même, bien que la vérification du modèle puisse être effectuée en deux minutes pour certaines applications, cette durée varie considérablement en fonction de la taille du modèle. Dans certains cas, la durée de la vérification peut prendre une trentaine de minutes par application. Dans le but d'alléger la tâche de vérification, des algorithmes de vérification à la volée ont été utilisés. De surcroît, la taille des modèles générés

a été réduite en compactant les chaînes des actions silencieuses.

Le nombre considérable d'applications à vérifier combiné avec le nombre de propriétés temporelles rend la tâche de vérification longue et gourmande en mémoire surtout que la plupart des applications en question utilisent du multithreading. Pour accélérer le processus de la vérification, les calculs ont été effectués sur un supercalculateur afin de vérifier simultanément plusieurs applications.

Conclusion

L'apprentissage automatique est de plus en plus utilisé dans la détection de maliciels Android. Dans ce chapitre, nous avons présenté les algorithmes d'apprentissage automatique supervisé appliqués à la résolution de notre problématique. Pour une classification binaire des applications Android, les k-plus proches voisins, les séparateurs à vastes marges, les forêts aléatoires et le gradient boosting ont été entraînés et évalués. La comparaison des différents modèles étudiés montre que les forêts aléatoires atteignent les meilleures performances en terme d'exactitude et de F1-mesure.

Conclusion

Android est sans conteste le système d'exploitation mobile le plus répandu. Cette popularité a suscité un vif intérêt de la part des développeurs de logiciels malveillants. En effet, le grand nombre d'appareils tournant sur Android combiné à la diversité des données qui résident dans ces appareils a fait de ces derniers une source importante de données sensibles. Android est confronté à de nombreux défis et problèmes de sécurité ainsi, la sécurité sur Android constitue un sujet de recherche captivant l'attention de plusieurs chercheurs. Pour faire face aux enjeux liés à la sécurité sur Android, il devient primordial de mettre en œuvre de nouvelles approches de détection de logiciels malveillants visant la plateforme mobile.

Synthèse des résultats

Dans ce mémoire, nous avons proposé et évalué une méthode de détection de maliciels Android basée sur la vérification de modèles et l'apprentissage automatique. En effet, la construction d'un modèle capturant la sémantique du comportement des applications Android est l'un de nos objectifs de recherche. À cette fin, nous avons développé un programme java baptisé Apk2Lnt permettant de construire un modèle formel exprimé en termes de processus LNT et ce, à partir du fichier APK d'une application Android. Le modèle généré fait l'objet d'une analyse rigoureuse permettant de vérifier si l'application en question exhibe des comportements suspects.

Notre deuxième objectif de recherche est la définition et la spécification des comportements suspects des applications Android. Pour répondre à cet objectif, nous nous sommes basés sur l'analyse de l'évolution des maliciels Android ainsi que l'analyse des différents vecteur d'attaques exploités par les concepteurs de logiciels malveillants Android. Une fois définis, ces comportements ont été spécifiés en μ-calcul modal.

Pour vérifier si une application Android exhibe un comportement pouvant porter atteinte à la confidentialité, l'intégrité ou la disponibilité des données, nous nous sommes tournés vers l'apprentissage automatique. En effet, en se basant sur le modèle LNT des applications Android les formules en μ-calcul encodant les comportements suspects sont vérifiées par un vérificateur de modèle. Le résultat de la vérification de chacune des propriétés constitue une

caractéristique (*feature*) sur laquelle le modèle décisionnel s'appuie pour classifier l'application. Outre les propriétés temporelles, le classifieur se base sur la présence ou l'absence des appels aux méthodes de l'API Android nécessitant la détention d'une permission. Concernant notre troisième et dernier objectif de recherche, les résultats des expérimentations d'apprentissage automatique montrent que le classifieur des forêts aléatoires nous permet de différencier entre les applications bénignes et celles malicieuses avec une exactitude de $94,60\%$ et une F1-mesure de $95,94\%$. L'utilisation des propriétés temporelles comme caractéristiques apporte une légère amélioration par rapport à l'utilisation des appels API seules. Nous notons une amélioration de la F1-Mesure et de l'exactitude souvent accompagnées par une diminution du taux de faux négatifs et du taux de faux positifs.

Limitations et travaux futurs

Malgré les résultats encourageants, les travaux de recherche effectués dans le cadre de cette maîtrise présentent certaines limitations et demeurent sujet à amélioration. Ces limitations constituent un terrain fertile pour de nouvelles pistes de recherche ultérieure.

La vérification de modèle constitue une phase importante dans le processus de détection proposé. En effet, le temps nécessaire à la vérification dépend de la taille du modèle, qui à son tour dépend de la taille de l'application. En raison du problème d'explosion d'états et la contrainte du temps, nous avons limité la taille des applications Android analysées à 2 mégaoctets. Cette limitation nous a permis d'alléger et d'accélérer le processus de vérification. Cependant, cette contrainte a un impact nuisible sur les performances du classifieur, dans la mesure où elle limitait le nombre ainsi que la variété des applications Android sur lesquelles se base l'apprentissage. Pour pallier cette limitation, nous considérons le raffinement du modèle généré et ce en travaillant sur la précision et la concision du modèle. La prise en compte d'autres mécanismes de synchronisation telle la barrière de synchronisation serait bénéfique en termes de précision. De même, l'exploration et la mise en œuvre d'autres techniques de réduction du modèle aideront à atténuer cette contrainte.

Une autre avenue de recherche à considérer est l'élargissement de l'ensemble des propriétés temporelles. En effet, cela permettra de couvrir plus de comportement suspects et contribuera ainsi à l'amélioration des performances prédictives du modèle.

Bibliographie

[1] Current Android malware, Accessed : 2018-07-07. https://forensics.spreitzenbarth.de/android-malware/.

[2] Androzoo markets, Accessed : 2019-06-10. https://androzoo.uni.lu/markets.

[3] MCafee mobile threat report, the next 10 years, Accessed : 2019-10-31. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf.

[4] Droidkungfu and the exploits rage against the cage and exploid for Android, Accessed : 2019-12-08. https://www.malware.unam.mx/en/content/droidkungfu-and-exploits-rageagainstthecage-and-exploid-android.

[5] Trojan : Android/jifake, Accessed : 2019-12-08. https://www.f-secure.com/v-descs/trojan_android_jifake.shtml.

[6] Trojan : Android/plankton, Accessed : 2019-12-08. https://www.f-secure.com/v-descs/trojan_android_plankton.shtml.

[7] Virustotal free online virus, malware and url scanner, Accessed : 2019-12-08. https: www.virustotal.com.

[8] Activity lifecycle, Accessed : 2019-03-18. https://developer.android.com/reference/android/app/Activity.

[9] Android platform architecture, Accessed : 2019-03-18. https://developer.android.com/guide/platform.

[10] IDC smartphone os market share, Accessed : 2019-10-31. https://www.idc.com/promo/smartphone-market-share/os.

[11] Android NDK | Android developers, Accessed : 2019-12-08. https://developer.android.com/ndk/.

[12] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer : Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[13] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo : Collecting millions of Android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[14] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin : Effective and explainable detection of Android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[15] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid : Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM Sigplan Notices*, 49(6) :259–269, 2014.

[16] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout : analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[17] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[18] Andrew Bedford, Sébastien Garvin, Josée Desharnais, Nadia Tawbi, Hana Ajakan, Frédéric Audet, and Bernard Lebel. Andrana : Quick and accurate malware detection for Android. In *International Symposium on Foundations and Practice of Security*, pages 20–35. Springer, 2016.

[19] Leo Breiman. Random forests. *UC Berkeley TR567*, 1999.

[20] Gerardo Canfora, Andrea Di Sorbo, Francesco Mercaldo, and Corrado Aaron Visaggio. Obfuscation techniques against signature-based detection : a case study. In *2015 Mobile Systems Technologies Workshop (MST)*, pages 21–26. IEEE, 2015.

[21] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Detecting Android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20. ACM, 2015.

[22] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Frédéric Lang, Christine McK-inty, Vincent Powazny, Wendelin Serwe, and Gideon Smeding. Reference manual of the LNT to LOTOS translator, 2017.

[23] Wei Chen, David Aspinall, Andrew D Gordon, Charles Sutton, and Igor Muttik. On robust malware classifiers by verifying unwanted behaviours. In *International Conference on Integrated Formal Methods*, pages 326–341. Springer, 2016.

[24] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3) :273–297, 1995.

[25] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1) :21–27, 1967.

[26] Antonio Cuomo, Antonella Santone, and Umberto Villano. A novel approach based on formal methods for clone detection. In *2012 6th International Workshop on Software Clones (IWSC)*, pages 8–14. IEEE, 2012.

[27] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid : an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2) :5, 2014.

[28] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1) :119–139, 1997.

[29] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010 : A toolbox for the construction and analysis of distributed processes. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 372–387. Springer, 2011.

[30] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *International Colloquium on Automata, Languages, and Programming*, pages 299–309. Springer, 1980.

[31] Giacomo Iadarola, Fabio Martinelli, Francesco Mercaldo, and Antonella Santone. Formal methods for android banking malware analysis and detection. In *2019 Sixth International Conference on Internet of Things : Systems, Management and Security (IOTSMS)*, pages 331–336. IEEE, 2019.

[32] Mohd Ishrat, Manish Saxena, and Mohd Alamgir. Comparison of static and dynamic analysis for runtime monitoring. *International Journal of Computer Science & Communication Networks*, 2(5), 2012.

[33] Xuxian Jiang and Yajin Zhou. Dissecting Android malware : Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012.

[34] Eran Kalige, Darrell Burkey, and IPS Director. A case study of eurograbber : How 36 million euros was stolen via malware. *Versafe (White paper)*, 35, 2012.

[35] Akshay Kapoor and Sunita Dhavale. Control flow graph based multiclass malware detection using bi-normal separation. *Defence Science Journal*, 66(2) :138–145, 2016.

[36] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 174–187. Springer, 2005.

[37] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning : A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160 :3–24, 2007.

[38] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical computer science*, 27(3) :333–354, 1983.

[39] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitraş. The dropper effect : Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1118–1129. ACM, 2015.

[40] Zhiqiang Li, Jun Sun, Qiben Yan, Witawas Srisa-an, and Yutaka Tsutano. Obfusifier : Obfuscation-resistant android malware detection system. In *International Conference on Security and Privacy in Communication Systems*, pages 214–234. Springer, 2019.

[41] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. Semantic modelling of Android malware for effective malware comprehension, detection, and classification. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 306–317. ACM, 2016.

[42] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Ransomware steals your phone. formal methods rescue it. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 212–221. Springer, 2016.

[43] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[44] Vittoria Nardone, Antonella Santone, CORRADO AARON Visaggio, Battista Pasquale, et al. Identification of Android malware families with model checking. In *ICISSP 2016*, pages 542–547. SciTePress, 2016.

[45] Radu S Pirscoveanu, Steven S Hansen, Thor MT Larsen, Matija Stevanovic, Jens Myrup Pedersen, and Alexandre Czech. Analysis of malware behavior : Type classification using machine learning. In *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, pages 1–7. IEEE, 2015.

[46] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this ! analyzing unsafe and malicious dynamic code loading in Android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[47] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing java using attributes. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 8. IBM Press, 2000.

[48] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon : evaluating Android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[49] Fu Song and Tayssir Touili. PoMMaDe : pushdown model-checking for malware detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 607–610. ACM, 2013.

[50] Fu Song and Tayssir Touili. Model-checking for Android malware detection. In *Asian Symposium on Programming Languages and Systems*, pages 216–235. Springer, 2014.

[51] Statista, The Statistics Portal. Number of available applications in the google play store from december 2009 to june 2018, Accessed : 2019-12-08. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ .

[52] Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-an, Gregg Rothermel, and Jackson Dinh. Jitana : A modern hybrid program analysis framework for android platforms. *Journal of Computer Languages*, 52 :55–71, 2019.

[53] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Execution and property specifications for jpf-Android. *ACM SIGSOFT Software Engineering Notes*, 39(1) :1–5, 2014.

[54] Timothy Vidas and Nicolas Christin. Evading Android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[55] Janett Walters-Williams and Yan Li. Comparative study of distance functions for nearest neighbors. In *Advanced Techniques in Computing Sciences and Software Engineering*, pages 79–84. Springer, 2010.

[56] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current Android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer, 2017.

[57] Long Wen and Haiyang Yu. An android malware detection system based on machine learning. In *AIP Conference Proceedings*, volume 1864, page 020136. AIP Publishing LLC, 2017.

[58] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware Android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[59] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[60] Jiawei Zhu, Zhengang Wu, Zhi Guan, and Zhong Chen. Api sequences based malware detection for Android. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 673–676. IEEE, 2015.