UNIVERSITÉ DU QUÉBEC À MONTRÉAL

DÉTECTION AUTOMATIQUE D'ANTI-FONCTIONNALITÉS DANS LES APPLICATIONS MOBILES POUR ENFANTS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAITRISE EN INFORMATIQUE

PAR

EMMANUEL MERLO

MAI 2026

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.12-2023). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

REMERCIEMENTS

Je tiens à exprimer ma profonde gratitude à mes parents, Ettore et Martine, ainsi qu'à ma sœur, Raphaëlle, pour leur soutien, leurs encouragements et leur patience tout au long de mes études. J'adresse aussi mes remerciements à Zac, dont la présence et la joie de vivre m'ont été d'un grand réconfort tout au long de la maitrise.

Je tiens aussi à exprimer mes remerciements les plus sincères à mon directeur de recherche Jean Privat de m'avoir guidé et encadré tout au long de la maitrise.

Merci aussi à Maude Bonenfant et son équipe de recherche d'avoir fourni l'échantillon utilisé dans le cadre de ce mémoire.

Table des matières

Liste des figures

LISTE DES TABLEAUX

RÉSUMÉ

Un grand nombre d'enfants interagissent avec un appareil intelligent (Auxier et al., 2020). En raison de ce nombre d'usagers potentiels, beaucoup d'applications destinées aux enfants sont disponibles sur des boutiques d'applications telles que le Google Play Store, l'App Store ou le Samsung Galaxy Store. Ces applications, dont la presque totalité (96.99% sur le Google Play Store) est gratuite (42matters, 2025a), peuvent parfois utiliser des stratégies néfastes pour les enfants afin de générer des revenus, telles que la collecte abusive d'informations privées, l'utilisation de publicité, l'emploi de certaines pratiques visant à inciter le visionnement des dites publicités ou un emploi malhonnête des microtransactions.

Ce projet vise à:
1) Développer des algorithmes et des stratégies permettant la détection automatique de certaines de ces anti-fonctionnalités, telle que la présence de publicité ou de microtransactions.
2) Valider les dites stratégies en les implémentant dans un outil.

Plus spécifiquement, les approches développées portent sur les aspects suivants: les publicités, les microtransactions ainsi que les notifications poussées.
Parmi les microtransactions, l'outil développé vise à détecter les microtransactions non consommables (c'est-à-dire les microtransactions offrant un bien permanent), les microtransactions consommables (soit les microtransactions offrant un bien qui ne peut être utilisé qu'un nombre limité de fois) et les abonnements (qui est un bien disponible pour une durée prédéterminée au moment de l'achat).

Parmi les publicités, l'outil développé vise à détecter les publicités récompensées (soit des publicités offrant une récompense au joueur pour le visionnement de la publicité) et les publicités interstitielles (soit des publicités s'affichant sur la totalité de l'écran).

Ce projet a permis de développer une stratégie simple permettant la détection des publicités récompensées et interstitielles ainsi que, dans une moindre mesure, la présence de microtransactions.

Mots clés: application mobile, Android, anti-fonctionnalité, décompilation, analyse statique, détection

CHAPITRE 1
INTRODUCTION

1.1 Mise en contexte

L'accès des enfants à un appareil mobile a connu une forte augmentation depuis 2011 (Shah & Phadke, 2023). De plus, en raison de la facilité d'utilisation des boutiques d'applications telles que le Google Play Store ou l'App Store, l'accès aux applications mobiles est devenu significativement plus aisé – tout comme le développement et la mise en marché de ces applications en raison de la facilité d'accès aux ressources matérielles et didactiques, autant pour les développeurs que pour de possibles studios de développement envisageant la mise en marché d'une nouvelle application. En conséquence, le marché des applications mobiles destinées aux enfants s'est étendu, ce qui a eu comme conséquence une forte hausse du nombre de ces applications. Parmi celles-ci, les applications gratuites représentent la quasi-totalité des applications disponibles (42matters, 2025a) – en 2025, 96.99% des applications sur le Google Play Store étaient gratuites. Ces applications, étant gratuites, doivent donc utiliser une stratégie de monétisation autre que le prix d'achat. Parmi ces stratégies, on retrouve entre autres la vente de produits dans l'application, c'est-à-dire des microtransactions, les publicités ainsi que la vente de données personnelles par les développeurs.

Bien que les boutiques d'applications les plus populaires indiquent les applications contenant des microtransactions, celles-ci ne donnent pas d'informations quant aux catégories de ces microtransactions. Conséquemment, bien qu'un consommateur potentiel soit en mesure de savoir si une application peut contenir des microtransactions, celui-ci n'est pas en mesure de savoir si les microtransactions en question sont des microtransactions non consommables, consommables ou encore des abonnements. De la même façon, lorsque la boutique d'applications indique la présence de publicité, la catégorie de ces publicités n'est généralement pas indiquée et, conséquemment, le consommateur n'est pas en mesure de savoir si ces publicités sont des publicités récompensées (soit des publicités dont le visionnement optionnel accorde une récompense à l'utilisateur) ou des publicités interstitielles (soit des publicités s'affichant sur la totalité de l'écran). Ce genre de fonctionnalités sont souvent considérées comme des anti-fonctionnalités, c'est-à-dire des fonctionnalités allant à l'encontre de l'intérêt de l'utilisateur.

De plus, les renseignements donnés par les boutiques d'applications au sujet de ces anti-fonctionnalités ne sont pas consistants entre eux. Celles-ci n'indiquent ni les mêmes anti-fonctionnalités, ni les mêmes informations sur les applications qu'elles offrent. Le Galaxy Store (Samsung, n.d.) et l'App Store (Apple, n.d.), par exemple, indiquent tous les deux la présence de microtransactions, mais seul l'App Store indique le nom utilisé dans l'application pour ces microtransactions. Similairement, le Google Play Store indique la présence de microtransactions ainsi que la présence de publicités, mais le Galaxy Store indique uniquement la présence de microtransactions (Google, n.d.).

Cependant, les boutiques d'applications populaires comptent souvent sur le développeur pour déclarer la présence de microtransactions ainsi que, lorsque cela est requis par la boutique d'applications, la présence de publicités. Conséquemment, bien que cela contrevienne aux règles d'utilisation de la boutique d'application et crée un risque que l'application soit retirée, un développeur pourrait ne pas déclarer la présence de certaines anti-fonctionnalités afin de donner une fausse impression aux consommateurs potentiels pour les inciter à télécharger l'application.

boutique	Déclare la présence de publicités	Déclare la présence de microtransactions	Nombre d'utilisateurs
Amazon Appstore	non	oui	Fermé le 20 août 2025
Google play Store	oui	oui	> 2.5 milliards
Samsung Galaxy Store	non	oui	> 400 millions
Huawei AppGallery	non	oui	580 millions

Table 1.1: Anti-fonctionnalitées indiquées par quelques boutiques d'applications populaires

1.2 Problématiques

La présence de ces anti-fonctionnalités est un problème d'autant plus important en raison de la taille importante de la population touchée. En effet, la proportion d'enfants interagissant régulièrement avec un appareil mobile est élevée: 49% pour les enfants en bas de 2 ans, 62% pour les enfants entre trois et quatre ans et 59% des enfants de cinq à huit ans (Auxier et al., 2020). De plus, cette population est

particulièrement vulnérable aux enjeux rencontrés dans les applications mobiles pour enfants. Notamment, au niveau de la confidentialité, la compréhension des enfants au sujet de la collecte des informations personnelles et leurs conséquences n'est pas suffisante pour faire des choix éclairés; le contenu des publicités apparaissant dans l'application n'est pas directement géré par l'application en elle-même, mais par un réseau de publicité externe et peut donc être inapproprié pour les enfants (Chen et coll., 2013). Quant aux microtransactions, un nombre important d'enfants font de tels achats à l'insu de leurs parents, avec 40.8% des parents déclarant que leurs enfants ont fait des achats à leur insu (Mahipal, 2020).

Bien que la présence de ce type de comportement abusif est un problème reconnu, il n'y a pas d'outils permettant de détecter automatiquement la présence de certaines anti-fonctionnalités et de les classifier.

La détection de telles anti-fonctionnalités est un réel défi. En effet, bien qu'une application puisse être analysée manuellement en l'utilisant, certains aspects restent difficiles à détecter et peuvent demander beaucoup de temps – d'autant plus que certains aspects peuvent n'être "accessibles" qu'après un certain temps de jeu ou être invisibles à l'utilisateur. Par exemple, une microtransaction consommable permettant de recevoir une vie supplémentaire dans un jeu vidéo est typiquement disponible uniquement après que l'utilisateur ait connu un certain nombre d'échecs et, de manière similaire, une publicité récompensée (soit une publicité offrant une récompense après son visionnement par l'utilisateur) ne s'affichera que lorsque l'utilisateur se retrouve dans une situation qui bénéficierait de la récompense. Ainsi, bien que simple, une telle approche demanderait trop de ressources et de temps pour être utilisée à large échelle.

Cette limite, en raison du grand nombre de jeux mobiles sur les boutiques d'application (1 797 743 applications dont 245 525 jeux sur Google Play (42matters, 2025a) et 1 755 095 applications dont 205 903 jeux sur l'App Store (42matters, 2025b) en 2025 avec, respectivement, 1301 et 1524 applications ajoutées chaque jour en moyenne), rend l'analyse manuelle non envisageable dans un contexte réel.

1.3 Objectifs

Le but principal du projet est donc de proposer un algorithme visant à détecter certaines anti-fonctionnalités présentes dans une application mobile. Cet algorithme sera ensuite utilisé dans un outil

permettant la détection automatique de certaines anti-fonctionnalités au niveau du code, tel que la présence de certains types de publicité ou de microtransactions. Plus spécifiquement, l'outil développé porte sur les aspects suivant: les publicités, les microtransactions, les notifications poussées ainsi que certaines métadonnées telles que le nom de la compagnie de développement.

Parmi les microtransactions, l'outil vise plus spécifiquement à détecter trois types de microtransactions: les microtransactions consommables, les microtransactions non consommables et les abonnements.
Les microtransactions consommables sont un type de microtransaction qui offre un bien ne pouvant être utilisé qu'un nombre limité de fois. Les biens offerts par ce type de microtransaction sont consommés lors de leur utilisation par l'utilisateur et doivent, pour être utilisés à nouveau, être rachetés. On retrouve dans ce type de microtransaction des offres telles que de la monnaie dans l'univers du jeu, des potions de vie, des coffres à butin, ou encore la possibilité de réessayer un niveau sans pénalités après un échec.
Il y a, à l'opposé, les microtransactions non consommables. Ce type de microtransaction offre des biens qui, contrairement aux offres des microtransactions consommables, sont permanents et peuvent être utilisés un nombre illimité de fois par l'utilisateur. On retrouve dans ce type de microtransaction des offres telles que des éléments cosmétiques, des niveaux supplémentaires ou encore des personnages ou objets pouvant être débloqués.

Pour les publicités, il vise à détecter deux types de publicités, soit les publicités récompensées et les publicités interstitielles.
Les publicités récompensées sont un type de publicité dont le visionnement n'est pas obligatoire et est volontairement visionné par l'utilisateur de l'application. Afin d'inciter le joueur à visionner la publicité, ce type de publicité est accompagné d'une récompense qui sera offerte à l'utilisateur après le visionnement de la publicité, par exemple en accordant au joueur une vie supplémentaire ou de la monnaie dans l'univers du jeu.
Les publicités interstitielles sont un type de publicité s'affichant sur la totalité de l'écran de l'appareil. Contrairement aux publicités récompensées, ce type de publicité n'est pas optionnel et ne donne rien au joueur après le visionnement.

Finalement, pour les mécaniques persuasives de l'application, l'outil vise à détecter les notifications poussées.

L'échantillon fourni pour la validation de l'outil ne spécifiant pas la catégorie des notifications poussées, l'outil ne cherchera pas à classifier les notifications poussées, mais uniquement la présence des notifications.

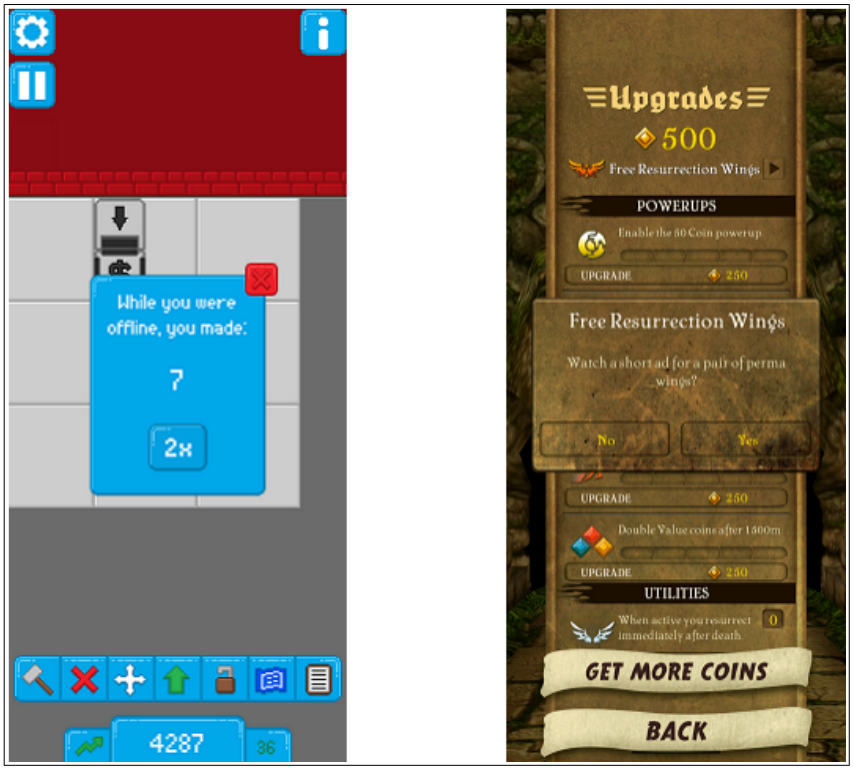

Figure 1.1: Exemples de proposition de visionnement d'une publicité récompensée dans les jeux Assembly Line (gauche) et Temple Run (droite)

Le projet cherchera aussi à répondre aux questions de recherche suivantes:

RQ1 : « Pouvons-nous identifier les motifs courants utilisés dans le code des applications contenant des anti-fonctionnalités ? »

Nous souhaitons savoir si des motifs peuvent être utilisés pour distinguer les applications présentant une anti-fonctionnalité donnée des applications qui n'en présentent pas. Ces motifs pourraient, par exemple, être le nom d'une bibliothèque, le nom d'une méthode, ou encore certaines phrases souvent utilisées dans l'interface utilisateur par les applications utilisant l'anti-fonctionnalité en question. Ces motifs, si

identifiés, seront ensuite utilisés par l'outil pour détecter la présence des anti-fonctionnalités associées dans les applications mobiles pour enfants.

RQ2 : « Parmi les motifs pouvant être utilisés pour détecter la présence d'anti-fonctionnalités dans une application, lesquels permettent la meilleure discrimination ? »
Il est probable que plusieurs motifs permettront la détection des anti-fonctionnalités similaires, mais n'offriront pas un même rappel ou une même précision lorsqu'utilisés par l'outil. Nous voulons savoir, parmi ces motifs, lesquels offrent une précision supérieure au prix d'un rappel inférieur et, vice-versa, quels motifs offrent un rappel supérieur au prix d'une précision inférieure et ainsi être en mesure de, selon les besoins, prioriser la précision de la détection ou le rappel de celle-ci.

1.4 Plan du mémoire

Le mémoire commence par faire le portrait de la situation concernant certaines problématiques en lien avec les applications mobiles au chapitre 2 avant de présenter certains concepts clés de l'architecture Android au chapitre 3. Puis, les concepts clés des applications Android sont présentés ainsi que le modèle de permissions utilisé par les systèmes Android au chapitre 4. La collecte de l'échantillon utilisé pour la validation des stratégies développées est ensuite expliquée ainsi que les stratégies utilisées pour obtenir le code source des applications dudit échantillon dans le chapitre 5 avant de présenter les anti-fonctionnalités qui feront l'objet de la recherche dans le code en plus de l'approche utilisée pour leur détection et de la conception de l'outil développé pour l'automatisation de cette détection au chapitre 6. Les résultats du projet sont ensuite présentés au chapitre 7 avant de conclure le mémoire au chapitre 8.

CHAPITRE 2
ÉTAT DE L'ART

Il y a peu de recherches portant sur la détection d'anti-fonctionnalités dans leur ensemble dans les applications mobiles. Bien qu'il existe des études visant la détection de certains comportements dans les applications mobiles, celles-ci se concentrent surtout sur un aspect ou comportement spécifique dans les applications – notamment les aspects de la confidentialité des informations personnelles et la détection de comportement malveillant. C'est principalement sur cet aspect que se distinguent les travaux effectués dans le cadre de ce mémoire. Au lieu de se concentrer sur un comportement ou aspect spécifique des applications mobiles, ceux-ci visent à détecter plusieurs types d'anti-fonctionnalités distincts.

Ce chapitre présente un aperçu de l'état de l'art et est divisé en quatre sections, chacune présentant les travaux portant sur un aspect spécifique. Il commence avec les travaux portant sur la confidentialité dans les applications mobiles (section 2.1) avant de présenter les travaux portant sur les publicités à la section 2.2 pour ensuite traiter des travaux portant sur les microtransactions (section 2.3). Finalement, la section 2.4 présentera des travaux qui, bien que ne portant pas spécifiquement sur les anti-fonctionnalités présentées dans ce mémoire, utilisent une approche similaire à celle utilisée dans ce mémoire pour détecter certains aspects dans le code source des applications Android.

2.1 Confidentialité

Le traitement des informations personnelles est un sujet de recherche populaire. Les objectifs de ces études sont très variés et peuvent autant avoir pour but de protéger le consommateur, d'offrir un aperçu de la situation actuelle ou de permettre aux développeurs de vérifier la conformité de leurs applications à certaines normes ou législations.

Par exemple Alamri et al. (2022) étudie la conformité des applications mobiles Apple vis-à-vis des politiques de l'App Store – qui demande entre autres que les applications fournissent une politique de confidentialité (Apple, 2025). En se basant sur l'analyse des métadonnées de deux millions d'applications présentes sur cette boutique, il apparaît que seulement 58.5% des applications fournissent un lien

amenant supposément vers une politique et que, parmi ceux-ci, 61.6% amènent réellement vers une politique de confidentialité.
Dans la boutique d'applications de Google, les travaux de Story et al. (2018) montrent que la proportion d'applications offrant une politique de confidentialité n'est que de 50% - tout comme l'Apple Store, le Google Play Store oblige la publication d'une politique de confidentialité. (Google, n.d.)

Les politiques de confidentialité étant un aspect très important pour la problématique du traitement des informations personnelles, de nombreuses études se penchent sur l'analyse des politiques de confidentialité. On retrouve Yu et al. (2018) proposant l'outil TapVerifier qui utilise une analyse statique de l'application et une analyse sémantique de la politique de confidentialité pour s'assurer que le comportement de l'application soit fidèle à ce qui est déclaré dans la politique de confidentialité. Similairement, certaines études telles que Gorla et al. (2014) cherchent à s'assurer que le comportement de l'application soit fidèle à la description de l'application dans la boutique d'applications dont elle fait partie, d'autres cherchent à s'assurer que la description de l'application liste correctement les permissions requises par l'application - par exemple, la position de l'utilisateur ou la liste de contacts avec Qu et al. (2014) - alors que d'autres cherchent, par l'analyse de la description de l'application ainsi que des permissions requises, à expliquer les raisons de la présence de chaque permission de ladite application (Pandita et al., 2013).

L'interface utilisateur de l'application est parfois utilisée. On retrouve Huang et al. (2014) qui propose un outil, AsDroid, utilisant une analyse statique des applications mobiles dans le but de détecter la présence de comportements malhonnêtes en recherchant des contradictions entre l'interface usager de l'application et son comportement. De plus, les problèmes du respect des informations personnelles ne sont pas uniquement causés par le développeur de l'application, mais aussi par les bibliothèques utilisées. En effet, certaines bibliothèques utilisées pour l'usage de publicité transmettent des informations privées dont l'utilisation est difficilement justifiable: bien que la position de l'appareil puisse être une information légitimement utilisée pour fournir des publicités pertinentes, l'historique d'appels ou la liste des applications présentes sur l'appareil sont plus difficilement justifiables (Grace et al., 2012).

Les aspects légaux sont aussi un enjeu étudié. Reyes et al. (2018), par exemple, utilisent une analyse dynamique des applications mobiles pour vérifier la conformité de l'application à la COPPA, loi

américaine portant sur la confidentialité et réglementant la façon dont les applications mobiles, jeux et sites Internet peuvent collecter et utiliser les informations privées des enfants de 13 ans ou moins. L'analyse en tant que telle a utilisé une version d'Android personnalisée afin d'identifier les informations auxquelles l'application accédait ainsi qu'un service de VPN personnalisé dans le but de vérifier à qui ces informations étaient transmises. Cette étude montre notamment que, sur 5855 applications analysées automatiquement, 7% entraient potentiellement en violation avec la COPPA et que 19% des applications mobiles pour enfants collectaient des informations privées par le biais de kits de développement dont les conditions d'utilisation interdisent leur utilisation pour de telles applications.

Les commentaires sont aussi utilisés pour la détection de problématiques au niveau du traitement des informations privées. Cen et al. (2014), par exemple, utilisent une méthode d'apprentissage multilabel pour détecter la présence de commentaires évoquant des inquiétudes portant sur la sécurité ou le respect des informations privées de l'application.

2.2 Publicités

Les publicités sont aussi un aspect populaire de recherche. Les sujets sont variés et peuvent porter sur les effets possiblement néfastes des publicités sur les enfants, la détection de fraude ou certains enjeux au niveau de la confidentialité.

Grace et al. (2012), par exemple, s'intéressent aux risques posés par les publicités au niveau de la confidentialité ainsi qu'au niveau de la sécurité dans les applications Android et propose un outil visant à identifier de tels risques. Cette identification est effectuée en quatre étapes après une identification des appels API présentant un risque potentiel. La première a consisté à dissocier le code de la bibliothèque de publicité du reste du code de l'application. Une analyse statique est ensuite effectuée dans le but d'identifier les risques potentiels, allant de la transmission d'informations personnelles à l'exécution de code provenant de sources externes. Dans un second temps, pour chaque API identifiée, deux cas de figure se présentent: les APIs qui sont intrinsèquement dangereuses (et donc identifiées comme telles) et celles qui ne peuvent représenter un risque qu'en présence d'un point de sortie. Dans ce dernier cas, l'outil effectue une analyse rétrospective dans le code source de la bibliothèque de publicité pour rechercher des points de sorties potentiels. Enfin, un CFG est utilisé pour vérifier si les API identifiées

précédemment peuvent être atteintes dans le code. L'analyse des bibliothèques de publicité les plus populaires effectuée par l'étude révèle que ces bibliothèques peuvent poser un danger significatif envers les informations privées de l'utilisateur: notamment, 18% des bibliothèques analysées utilisent des API visant à collecter le numéro de téléphone de l'appareil et près de la moitié collectent la position de l'appareil.

Stevens et al. (2012) se penchent sur les politiques de confidentialité des réseaux de publicités populaires pour connaitre les permissions demandées par ceux-ci, les informations privées qu'ils peuvent collecter auprès de l'appareil ainsi que la manière dont ceux-ci peuvent suivre l'utilisateur à travers de multiples applications différentes. Pour réaliser cela, les auteurs se sont inscrits comme développeurs auprès de chaque réseau choisi pour obtenir les bibliothèques les plus récentes au moment de l'étude (soit durant le mois de février 2012) et ensuite instrumenter une application effectuant une demande de publicité tout en capturant le réseau. L'étude montre plusieurs risques importants au niveau de la collecte des informations sensibles de l'utilisateur. En effet, plusieurs utilisaient des permissions non documentées par ledit réseau lorsque l'application possédait ces permissions – profitant du fait qu'une bibliothèque présente dans une application Android hérite des permissions de l'application. Bien que certaines bibliothèques, telles que Admob, n'essayaient pas de récolter des informations sensibles, laissant cette responsabilité au développeur de l'application, d'autres, telles que Airpush, incluaient automatiquement des données telles que les coordonnées GPS dans les demandes de publicité lorsque les permissions requises étaient accordées à l'application. Finalement, toutes les bibliothèques examinées utilisaient ou bien l'ANDROID_ID (une chaîne hexadécimale générée aléatoirement lors du premier démarrage de l'appareil) ou bien le DEVICE_ID (correspondant au numéro de série de l'appareil) pour suivre l'utilisateur à travers plusieurs applications distinctes.

Certaines études portent aussi sur la fraude effectuée par certaines applications envers les réseaux de publicité. La rémunération d'une application par les réseaux de publicité étant calculée selon le nombre de visionnements des publicités ainsi que le nombre de "clics" par l'utilisateur, les développeurs sont incités à maximiser autant la quantité de publicités dans une application que la quantité d'utilisateurs cliquant sur ces publicités. Une quantité trop importante de publicités dans une application pouvant éloigner des utilisateurs potentiels, certaines applications cherchent à créer de "faux" visionnements ainsi que de faux "clics" pour pouvoir augmenter le nombre de visionnements ou de clics sans éloigner

des consommateurs potentiels de l'application.

Les travaux de Crussell et al. (2014), par exemple, s'intéressent à deux types de comportements frauduleux: les requêtes de publicités auprès du réseau de publicité alors que l'application est en arrière-plan (et, conséquemment, qui n'est pas affichée à l'utilisateur) ainsi que la génération des clics sans que l'utilisateur n'interagisse avec l'application. Dans le but de détecter ces comportements, l'étude utilise une analyse dynamique des applications dans un émulateur afin de capturer les visionnements et les clics. Effectuée sur un échantillon composé de 130 339 applications disponibles sur diverses boutiques d'application et 35 087 applications contenant potentiellement du code malicieux, l'analyse montre qu'environ 30% des 40 409 applications étudiées font la demande d'une publicité alors que ladite application est en arrière-plan et que 27 applications ont falsifié des clics. Notons que la demande des publicités quand l'application est en arrière-plan n'est pas un indicateur parfait d'un comportement frauduleux de la part de l'application. En effet, il est possible qu'une demande soit complétée en arrière-plan sans que ce soit dans un contexte de fraude. Par exemple, il est possible que la demande ait été initiée alors que l'application était en premier plan et n'a pas pu être complétée avant que l'application n'ait été déplacée en arrière-plan. Dans le but de diminuer le risque d'erreur provenant de tels cas de figure, l'étude a mis en place une période de grâce de cinq secondes, durant laquelle les requêtes de publicités détectées n'étaient pas considérées comme étant des requêtes frauduleuses.

Toujours dans la détection de fraude, les travaux de Dong et al. (2018), créent une taxonomie de comportement frauduleux déjà existant en plus de créer une nouvelle catégorie de fraude: les fraudes aux interactions dynamiques. Celles-ci se distinguent des autres types de fraudes du fait qu'elles impliquent plusieurs états de l'interface utilisateur. L'étude présente aussi un outil, FrauDoid, visant à détecter automatiquement les comportements frauduleux identifiés. Cet outil utilise une analyse dynamique des applications et analyse la disposition de l'interface ainsi que le trafic Internet pour ensuite se baser sur un ensemble de règles heuristiques afin de détecter les comportements frauduleux. Par exemple, la fraude de publicités cachées, qui consiste à "cacher" une publicité et ainsi donner l'impression à l'utilisateur que l'application ne contient pas de publicité, est détectée si l'une des bordures de la publicité intersecte avec un autre élément de l'interface en plus d'avoir une coordonnée "z" inférieure audit élément. L'outil proposé par l'étude offre une précision de 93.88% et un rappel de 92%. L'outil a aussi été utilisé sur 12 000 applications Android contenant de la publicité pour découvrir 335 cas de fraude.

Finalement, certaines recherches ont étudié le niveau de maturité des publicités affiché dans les applications mobiles. On retrouve notamment Chen et al. (2013) qui analyse manuellement 405 applications mobiles disponibles sur le Google Play Store et sur l'App Store pour recueillir un total de 3921 publicités. Après que les niveaux de maturité de ces publicités furent déterminés et comparés avec les classifications d'âge des applications, il est apparu que 35.9% des publicités sur les applications Android et 38.9% des publicités sur les applications iOS avaient un niveau de maturité trop élevé pour la classification d'âge de l'application.

On retrouve aussi Zhao et al. (2023) qui utilise une analyse dynamique des applications Android pour vérifier que les publicités dans les applications mobiles destinées à un groupe d'âge incluant les enfants sont appropriées pour les enfants. Google n'autorisant qu'un nombre restreint de bibliothèques de publicités pour les applications dans un tel groupe d'âge, ladite analyse permet aussi de vérifier que l'application utilise uniquement ces bibliothèques. Utilisée sur un échantillon composé de 25 000 applications dont le groupe d'âge du Google Play Store incluait les enfants, l'étude a pu recueillir des publicités de 8 971 applications. Parmi ces 8 971 applications, il est apparu que 775 d'entre elles ont affiché des publicités inappropriées pour les enfants.
La différence importante entre l'échantillon et le nombre d'applications dont des publicités ont pu être recueillies provient principalement du manque de stabilité de certaines applications ainsi que l'incapacité de l'outil de simuler des interactions complexes parfois nécessaires pour l'affichage de la publicité.

2.3 Microtransactions

Les sujets des études portant sur les microtransactions sont variés. Ces études s'intéressent entre autres à l'aspect purement commercial des microtransactions, aux dangers associés à certains types de microtransactions, tels que les lootboxes, ainsi qu'à la fraude.

D'un point de vue commercial, Salehudin & Alpert (2021) s'intéressent aux raisons pour lesquelles les utilisateurs des applications ne sont pas enclins à procéder à des microtransactions. Cette étude qualitative a examiné le contenu présent sur des sites de critiques de jeux et des sites sur lesquels le sujet des microtransactions était discuté, ainsi qu'une série d'entrevues avec des utilisateurs actifs de jeux mobiles. Cette étude propose que l'un des concepts les plus importants pour la réticence à l'achat soit la

perception d'injustice de l'utilisateur par rapport aux microtransactions présentes dans l'application. Ce concept amène au thème principal de l'étude voulant que la perception de l'agressivité de la monétisation soit l'un des critères les plus importants amenant la réticence à l'achat: plus l'utilisateur perçoit que le développeur "exagère" au niveau de la monétisation présente dans l'application, plus l'utilisateur sera réticent à procéder à des microtransactions dans l'application. La perception de l'agressitivé de la monétisation amenée par l'étude est composée de cinq dimensions, divisées en deux catégories. Au niveau de la perception de liberté, on retrouve la perception qu'a l'utilisateur quant au caractère manipulateur, addictif et intrusif, alors qu'au niveau de la perception de l'égalité, on retrouve la perception qu'a l'utilisateur face au surcoût et au risque.

À l'opposé, Hamari et al. (2017) s'intéressent aux raisons pour lesquelles les utilisateurs d'applications gratuites effectuent des microtransactions. L'étude a premièrement identifié un ensemble de raisons obtenues par une analyse des 100 applications gratuites les plus populaires (les jeux de casinos étant exclus), triangulée avec la littérature sur le modèle économique des jeux vidéos et sur les motivations de jeu, et appuyée par des discussions menées avec des développeurs de jeux au cours des huit dernières années. Ces raisons ont ensuite été approfondies, évaluées et affinées en collaboration avec des spécialistes de l'industrie. L'étude a ensuite développé un questionnaire visant à déterminer lesquelles de ces raisons sont les raisons principales poussant les utilisateurs à faire des achats. Le questionnaire a révélé que la principale motivation d'effectuer une microtransaction était de déverrouiller du contenu dans l'application, suivi par une envie d'encourager les développeurs de l'application, un prix raisonnable des produits offerts, les offres spéciales et le désir d'investir dans un passe-temps.

Les risques possibles des microtransactions sont aussi un sujet de recherche important, notamment sur les lootboxes. Ce type de microtransaction se distingue des autres car, contrairement aux autres types de microtransactions, ce qui est reçu lors de l'achat n'est pas connu avant l'achat: les lootboxes, lors de son utilisation, vont plutôt donner un produit aléatoire au joueur (produit pouvant, selon les applications, donner un avantage au joueur ou encore être purement esthétique). Ce type de microtransaction est souvent comparé à des jeux d'argent à cause de l'aspect aléatoire de la "récompense" et de l'utilisation de sons et de lumières excitantes.
On retrouve notamment les travaux de Brooks & Clark (2019) démontrant une corrélation entre l'utilisation des lootboxes et les comportements de jeu d'argent ainsi que les travaux de

Zendle & Cairns (2018) qui montrent un lien entre l'utilisation de ce type de microtransaction et le jeu problématique; Zendle & Cairns (2019) démontrent une corrélation positive entre le montant des dépenses dédiées aux lootboxes et la sévérité du problème de jeu de l'utilisateur.

Finalement, les problèmes de fraude et la protection contre celle-ci sont aussi un aspect étudié au niveau des microtransactions. La fraude est étudiée autant du point de vue de l'utilisateur des applications que des développeurs ou des boutiques d'applications prenant en charge la transaction. Notamment, Zhou et al. (2021) proposent un système – nommé Payment-Guard - visant à détecter les achats frauduleux sur les systèmes iOS. Afin de détecter un achat frauduleux, Payment-Guard caractérise chaque achat selon quatre dimensions: le comportement du compte de l'utilisateur, le comportement de l'appareil, le comportement de l'IP ainsi qu'une dimension combinant ces trois aspects. Il procède ensuite à la détection en se basant sur ces dimensions. Cette analyse permet de détecter 92.2% des transactions frauduleuses avec 2% de faux positifs.

Au niveau de la protection du consommateur, Yue et al. (2024) proposent la détection d'un type spécifique de fraude, soit les applications visant à escroquer l'utilisateur avec des abonnements payants, notamment en rendant difficile l'annulation de l'abonnement ou en dissimulant des frais à l'utilisateur. Une des grandes difficultés qui rend difficile la détection de ce type de fraude est que non seulement les applications avec ce type de fraude ne contiennent pas de code malicieux, mais utilisent aussi des pratiques proches des applications légitimes. Pour détecter ce type de fraude, les auteurs de l'étude proposent une méthode de détection automatisée qui, afin de contourner ces difficultés, ne se base pas sur le code de l'application. Elle se base plutôt sur l'interface de l'application concernant les abonnements, par exemple le contenu du texte présent sur l'écran. Ces informations sont recueillies avec des captures d'écran ainsi que les fichiers de mise en page de l'application et sont ensuite utilisées par un modèle de classificateur à arbre de décision peu profond pour déterminer si l'application analysée contient un tel type de fraude.

2.4 Détection de "smells"

On retrouve aussi des études visant à détecter certains "smells" d'une application à l'aide d'une détection de motifs dans le code de l'application. Par exemple, Palomba et al. (2017) proposent l'outil aDoctor

visant à détecter 15 types de “smells”, soit des symptômes d’un mauvais choix d’implémentation ou de conception durant le développement d’une application Android. Parmi ces “smells” on retrouve la transmission de fichier sans compression; l’utilisation d’un *wakelock*, mécanisme permettant de garder l’appareil allumé afin de compléter certaines taches, qui n’est pas désactivé; ou encore l’utilisation de la classe *alarmManager*, classe permettant l’exécution d’une opération à un moment spécifique, sans que la méthode “setInexactRepeating”, qui permet au système Android de grouper plusieurs exécutions pour maximiser le temps que l’appareil reste en veille, soit utilisée. L’approche utilisée pour la détection de ces “smells” se base sur des règles basées sur la recherche de motifs à des endroits spécifiques dans le code source de l’application. La détection de la transmission de fichiers sans compression, par exemple, est faite en vérifiant qu’une classe dont le corps contient “File “ contient aussi “zip”.
À l’aide d’une telle approche, l’outil parvient à obtenir, sur l’ensemble des 15 “smells”, une précision et un rappel de 0.98. Notons que cet outil vise à être opéré sur le code d’une application durant le développement de celle-ci et n’a pas comme objectif d’être utilisé sur du code décompilé.

L’étude de Hecht (2015) vise à détecter 8 aspects tels que les classes “Dieu” (soit les classes avec un nombre élevé d’attributs ou d’opérations et prenant en charge un nombre important de responsabilités comparé aux autres classes du programme) ou des méthodes excessivement longues comparées aux autres méthodes. Contrairement à aDoctor, Paprika opère sur le bytecode des applications Android. Afin de procéder à l’analyse de l’application, l’outil utilise le cadre Soot ainsi que le module Dexpler pour convertir le bytecode Dalvik de l’application en une représentation proche du Java. Cette représentation est ensuite utilisée par Paprika pour extraire les informations importantes pour l’analyse, telles que le nombre de paramètres de chaque méthode, permettant ainsi la détection des “smells”.
Lorsqu’appliqué sur des applications ayant été optimisées et obfusqué avec Proguard, Paprika permettait d’obtenir un rappel de 0.9032 et une précision de 1.

CHAPITRE 3

ANDROID

Android est un système d'exploitation ouvert basé sur le noyau Linux et est principalement conçu pour être utilisé par des appareils mobiles tels que les téléphones intelligents ou les tablettes. Originellement développé par l'entreprise Android Inc, le développement du système d'exploitation est ensuite pris en charge par un consortium dirigé par Google après l'achat d'Android Inc par Google. Le système Android est ensuite publié en 2007 sous la licence Apache 2.0, licence permettant entre autres l'utilisation commerciale du logiciel, la modification et distribution du code source ainsi que la distribution d'une version modifiée sous une nouvelle licence, que celle-ci soit plus restrictive ou plus libre que la licence originale (Apache Software Foundation, 2004).

Bien que le coeur du système d'exploitation Android, appelée *Android Open Source Project* (Projet Open Source Android) est, comme son nom l'indique, ouvert, la majorité des appareils disponibles sur le marché utilisent une version modifiée de l'AOSP. En effet, même si l'AOSP est techniquement suffisant pour être utilisé sur un téléphone mobile, la majorité des fonctionnalités typiquement disponibles et attendues par les consommateurs sur un appareil intelligent ne sont pas utilisables sans l'intégration de pilotes nécessaires pour permettre l'utilisation des composantes matérielles du téléphone. C'est le cas par exemple pour les ressources matérielles requises pour les connexions Bluetooth, ainsi que pour certaines composantes spécifiques à la puce utilisée par l'appareil. Le code distribué sous l'AOSP ne vise pas tant à être utilisé sur les appareils mobiles sans modifications, mais plutôt à être utilisé comme base pour les besoins du manufacturier. La portée du système Android n'est pas uniquement sur les appareils mobiles. En effet, Android peut aussi être utilisé, avec des modifications plus ou moins importantes, pour les systèmes d'infodivertissement des voitures (Android Open Source Project, 2025c) ou les télévisions intelligentes (Android Open Source Project, 2025a).

Ce chapitre commence par introduire la pile logicielle du système Android pour permettre une meilleure compréhension du mécanisme d'isolation des applications, présenté à la section 4.5. Le chapitre présente ensuite les fichiers utilisés par les applications dans un système Android afin d'appronfondir l'approche utilisée pour répondre aux questions de recherche.

3.1 Couches de l'architecture Android

La pile logicielle d'Android est composée de six modules, répartis sur cinq couches: les applications, le cadre de l'API Java, les bibliothèques C/C++, l'environnement d'exécution, la couche d'abstraction matérielle et le noyau Linux. Cette architecture suit le modèle représenté à la figure 3.1.

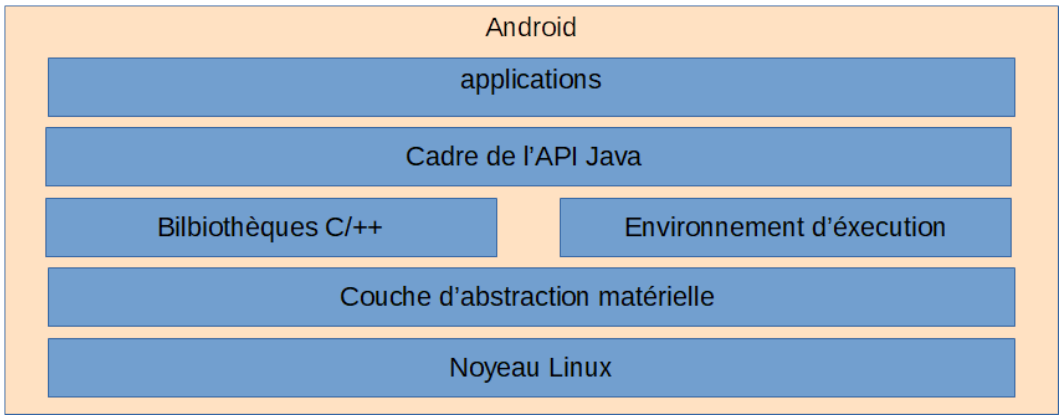

Figure 3.1: Schéma de l'architecture Android

3.1.1 Applications

Les applications sont la couche supérieure de l'architecture Android. Autant les applications préinstallées sur l'appareil, par exemple l'application de téléphonie ou la caméra, que les applications installées par l'utilisateur, que ce soit par le biais d'une boutique d'application ou installée manuellement, sont installées sur cette couche.

À quelques exceptions près, notamment l'application permettant de changer les paramètres de l'appareil, les applications préinstallées sur l'appareil n'ont pas de statut spécial comparé aux applications installées par l'utilisateur: comme toutes les autres applications installées par l'utilisateur, les applications préinstallées sur l'appareil peuvent être utilisées par les applications externes pour accomplir des tâches à la demande d'autres applications. En prenant pour exemple une application fictive appelée ToyFinder, qui vise à permettre d'identifier les commerces offrant un jouet pris en photo, il est possible pour cette application d'utiliser l'application de photographie de l'appareil, évitant ainsi au développeur de l'application en question de devoir implémenter nativement la prise de photographie.

3.1.2 Cadre de l'API Java

Cette couche, illustrée par la figure 3.2, représente l'ensemble des fonctionnalités du système d'exploitation. Ses fonctionnalités peuvent être exécutées par toutes les applications par le biais d'une API et incluent les services système et les gestionnaires du système Android.

On y retrouve, entre autres, les fournisseurs de contenu, le système de vues (utilisé par les applications Android pour la gestion des composants graphiques de l'interface des activités telles que les boutons ou les champs de texte), le gestionnaire d'activité (qui gère le cycle de vie de toutes les activités dans une application), le gestionnaire de notification (qui permet aux applications d'envoyer des notifications poussées) ainsi que le gestionnaire de fenêtre (qui prend en charge le placement ainsi que l'apparence des fenêtres des applications).

Notons que depuis l'introduction de Jetpack Compose en 2021, le système de vues n'est pas le seul système – ni celui recommandé par Android – pour la conception de l'interface des applications Android. En effet, Jetpack Compose a rendu possible l'utilisation d'une approche déclarative pour la gestion de l'interface utilisateur, ce qui n'était pas le cas avec l'ancienne méthode qui était basée sur des fichiers XML et une logique impérative. De plus, cette nouvelle approche de la gestion de l'interface utilisateur permet aussi de réduire le code générique nécessaire tout en facilitant la lecture et la maintenance du code de l'application.

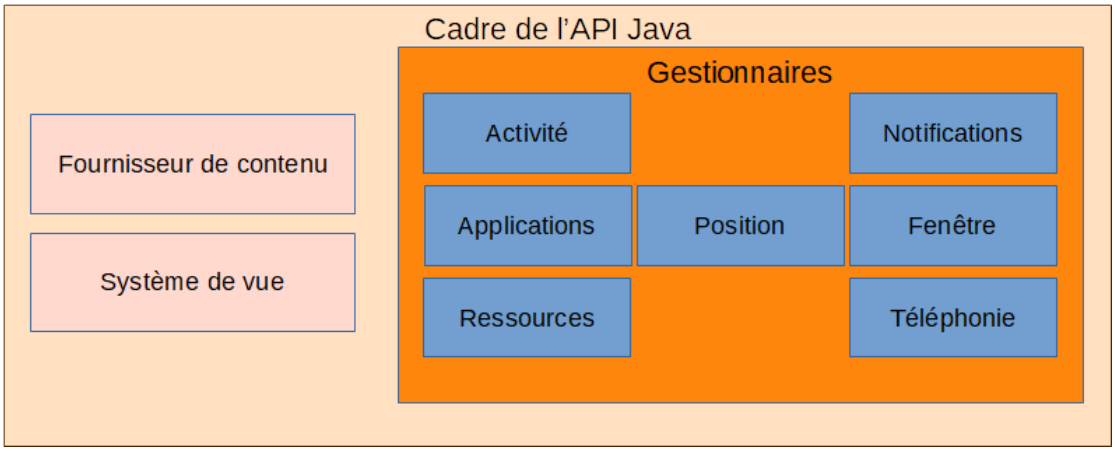

Figure 3.2: Schéma du cadre de l'API Java

Pour reprendre l'exemple de ToyFinder, cette couche sera utilisée pour, entre autre, envoyer une notification à l'utilisateur lorsque l'analyse d'un jouet est complétée et que des commerces offrant le

jouet ont été trouvés et pour implémenter les diverses activités de l'application telles que l'affichage des résultats ou l'écran d'accueil de l'application.

3.1.3 Environnement d'exécution et bibliothèques natives

Cette couche, illustrée par la figure 3.3, comporte deux composants principaux: l'Android Runtime (ART) et les bibliothèques C et C++.

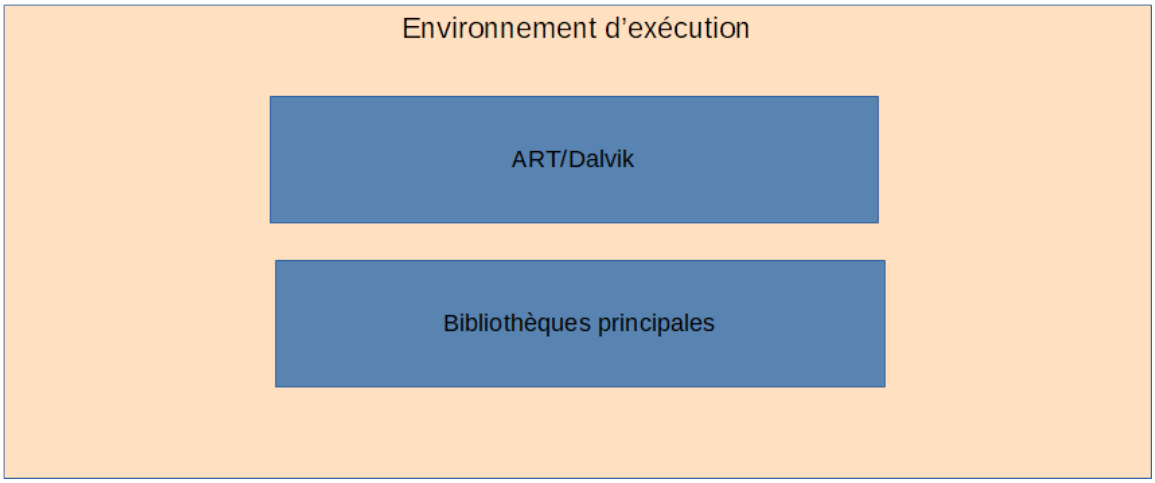

Figure 3.3: Environnement d'exécution Android

3.1.3.1 Android Runtime et Dalvik

L'ART (de l'anglais "Android Runtime", ou "environnement d'exécution Android") et Dalvik sont tous deux des environnements d'exécution d'Android. L'ART est l'environnement utilisé dans la version actuelle d'Android (soit la version 16 au moment de l'écriture de ce mémoire) dans l'architecture Android et remplace, depuis la version 5.0 d'Android, Dalvik. La distinction et l'historique des deux environnements peuvent être trouvés en annexe A.

3.1.4 Bibliothèques C/C++

Ces bibliothèques sont des bibliothèques précompilées en C ou en C++ visant à offrir un support pour le développement des applications Android. On retrouve, parmi celles-ci, des bibliothèques comme

OpenGL (utilisé pour les graphismes 2D et 3D), SQLite (qui apporte un support pour les bases de données) et WebKit (qui est un moteur de navigateur Internet ouvert apportant les fonctionnalités requises pour l'affichage de contenu web).

Dans l'exemple de ToyFinder, la bibliothèque WebKit sera utilisée pour afficher la page web des commerces présentant le produit recherché une fois que les commerces offrant le jouet pris en photos ont été identifiés.

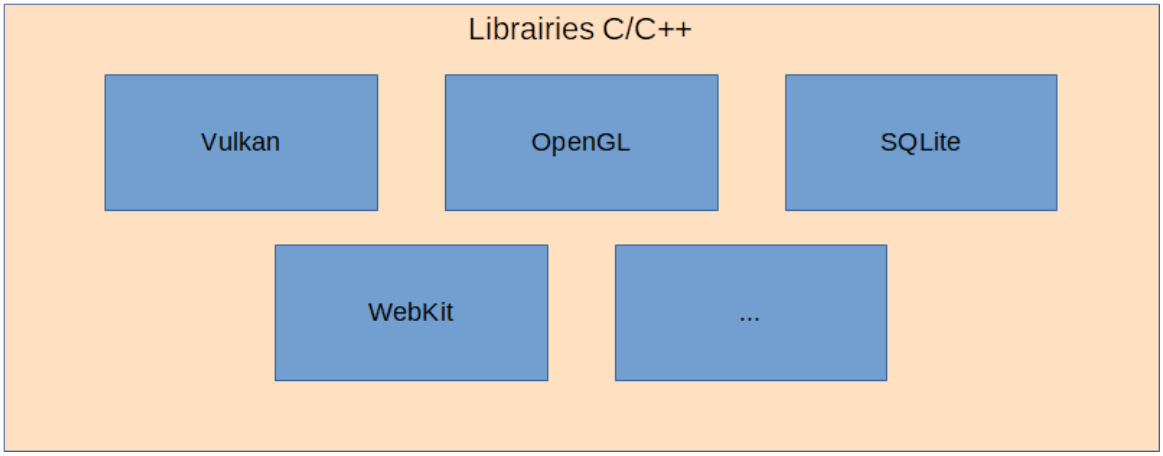

Figure 3.4: Bibliothèques C/C++

3.1.5 Couche d'abstraction matérielle

La couche d'abstraction, illustrée à la figure 3.5, sert de pont entre le cadre Android, écrit en Java et en Kotlin, et les divers pilotes des composantes matérielles de l'appareil. Cette couche établit une interface standardisée devant être implémentée par les fabricants des composantes matérielles de l'appareil et permet au cadre Android d'interagir, par le biais du HAL (de l'anglais *Hardware Abstraction Layer*), avec les composantes matérielles de l'appareil en utilisant un ensemble de fonctions préétablies. Cette couche est conçue pour être modulaire. Chaque composante matérielle possède leurs propres modules pouvant être développés indépendamment, remplacés ou mis à jour sans que le reste du système soit affecté. Cela rend ainsi le système Android indépendant des composantes matérielles de l'appareil, permettant à ce système d'être utilisé sur une vaste variété d'appareils avec une vaste variété de configurations matérielles sans nécessiter une modification du système d'Android spécifique à la configuration matérielle de l'appareil en question. (Android Open Source Project, 2025b)

Toujours dans le cas de ToyFinder, cette couche sera utilisée pour accéder au réseau WIFI lors de la recherche des commerces, ainsi que pour localiser l'utilisateur dans le but de prioriser les commerces à proximité de l'utilisateur. Un court historique de l'évolution du HAL peut être trouvé en annexe B.

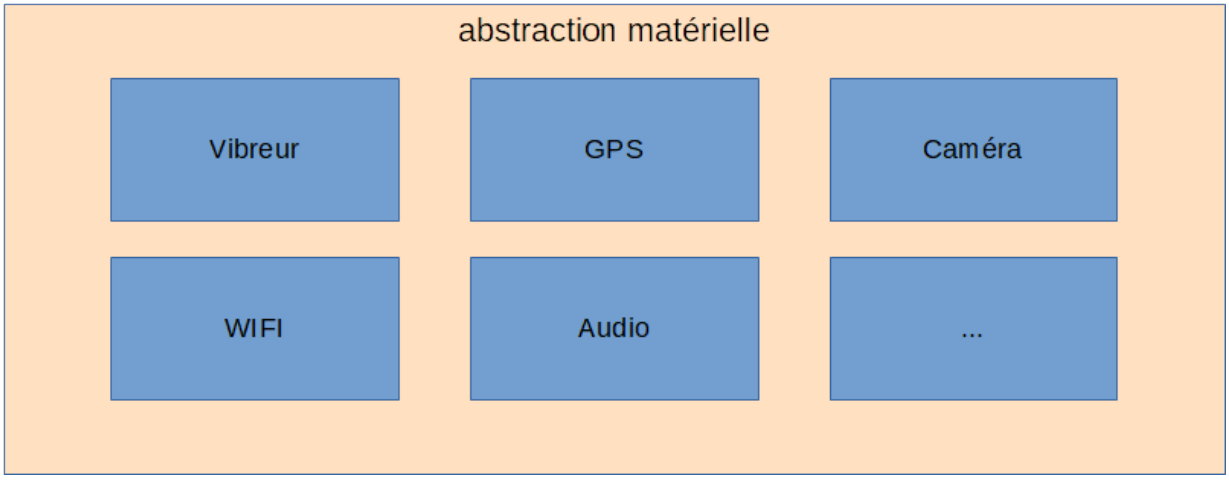

Figure 3.5: Schéma de la couche d'abstraction matérielle

3.1.6 Linux

Finalement, la dernière couche de l'architecture Android est le noyau Linux et est responsable de la gestion des ressources du système, que ce soit le processeur de l'appareil, sa mémoire ou l'implémentation des périphériques d'entrées, la gestion des processus ainsi que les mécanismes de sécurité bas niveau. Le noyau Linux utilisé est basé sur une version LTS (Long Term Supported) et est combiné avec des modifications spécifiques à Android nommé "Android Common Kernels" pour répondre aux besoins spécifiques des appareils mobiles.

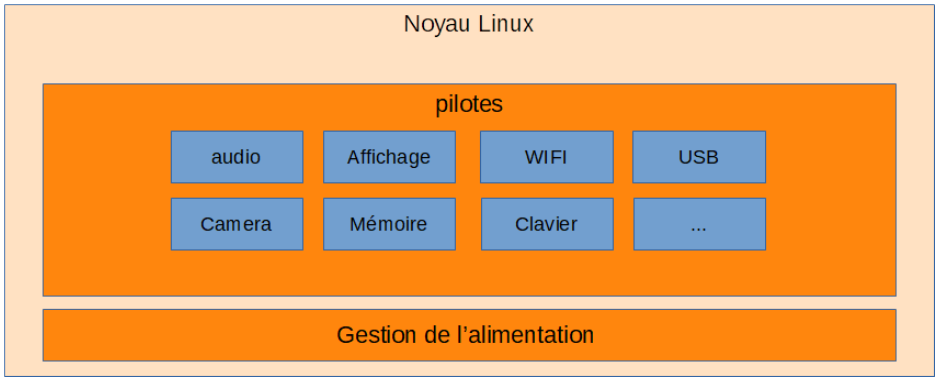

Figure 3.6: Schéma du noyau Linux

Dans le cas de ToyFinder, cette couche sera utilisée, indirectement, par les divers pilotes disponibles, notamment le pilote responsable du WIFI, de l'affichage et de la caméra.

3.2 Fichier APK

Les fichiers utilisés par une application Android, que ce soit le code source, les images, les vidéos ou les animations, sont regroupés en un fichier compressé de format APK (tenant de l'anglais "Android Package Kit"). Ce format de fichier est très similaire au format ZIP et utilise les mêmes algorithmes de compression que le format ZIP.

La principale différence entre le format ZIP et le format APK est la structure interne des fichiers contenus dans l'archive. En effet, contrairement aux archives de format ZIP, les archives de format APK suivent une structure prédéfinie. Plus spécifiquement, les archives de ce format contiennent minimalement un fichier "AndroidManifest.xml", un ou des fichiers "classes.dex", un fichier resources.arsc et des dossiers "META-INF", "assets", "lib" et "res". Si l'application a été développée avec le langage Kotlin, l'archive contiendra aussi le dossier "kotlin".

3.2.1 Structure des fichiers APK

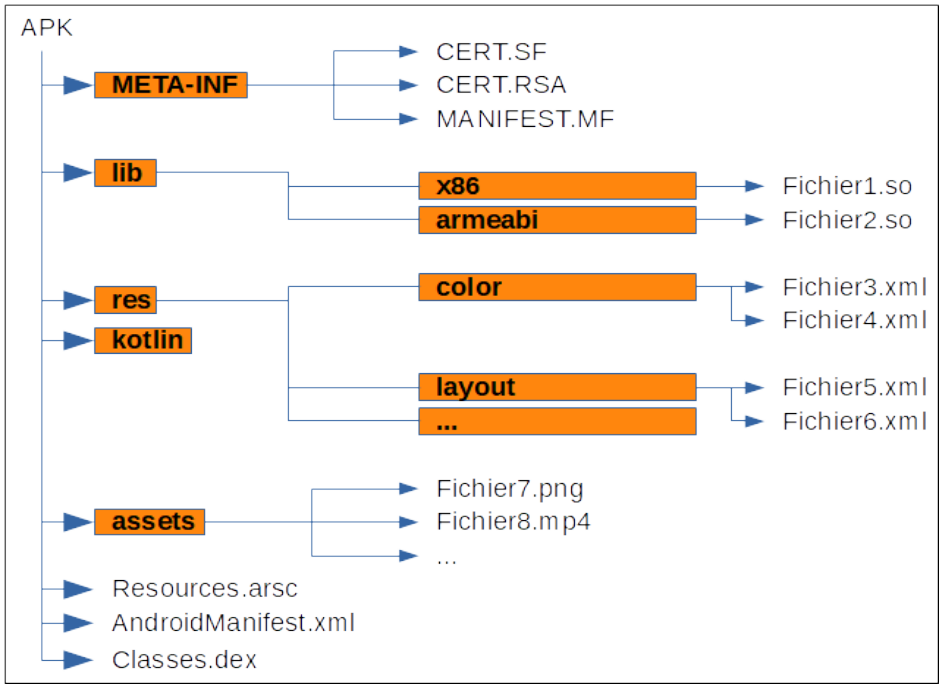

Figure 3.7: Structure d'un fichier APK

3.2.1.1 AndroidManifest.xml

Fichier en format XML indiquant des métadonnées essentielles au sujet de l'application. Parmi ces métadonnées, on retrouve entre autres les permissions demandées par l'application (section 4.4), les versions d'Android compatibles et le nom du paquet (*package*). Ce fichier est converti en un XML binarisé lors de la fabrication du paquet.

Ce fichier est important pour la détection des notifications poussées selon l'approche utilisée dans ce projet: les notifications requièrent une permission spécifique pour être utilisées par l'application. Puisque les permissions sont définies dans le fichier AndroidManifest.xml, il est possible de réduire les faux positifs en vérifiant que l'application possède la permission requise pour afficher une notification poussée.

3.2.1.2 Classes.dex

Extension tenant pour Dalvik EXecutable, les fichiers du format DEX contiennent le bytecode Dalvik de l'application et seront compilés par l'Android Runtime (ou, pour les versions 5.0 et antérieur, par Dalvik) de l'appareil. Il est possible pour une application de posséder plusieurs fichiers de ce format selon la taille du code de l'application.

Un fichier DEX est structuré comme un fichier binaire possédant plusieurs sections, chacune référencée par des décalages à partir d'un en-tête commun, tel que les identifiants des chaînes de caractères (listant toutes les chaînes utilisées dans l'application telles que les noms de classes ou le nom des champs), les identifiants des méthodes référencées par le fichier (que la méthode soit définie dans le fichier en question ou non) ou encore les définitions des classes. Contrairement au bytecode Java, qui empile/dépile les valeurs sur une pile, DEX utilise un bytecode basé sur les registres: chaque méthode déclare un certain nombre de registres virtuels et effectue les opérations sur ceux-ci. Le langage assembleur lisible par l'humain correspondant au bytecode Dalvik est connu sous le nom de Smali. Ces fichiers sont clés pour la détection des anti-fonctionnalités dans une application: ceux-ci seront utilisés pour récupérer le code source de l'application et ainsi permettre l'analyse de l'application en question.

3.2.1.3 Resources.arsc

Fichier binaire contenant les valeurs des ressources définies dans les fichiers “strings.xml”, “colors.xml” et “styles.xml” en plus des références aux ressources plus complexes telles que les images ou la mise en page des interfaces. Le fichier Resources.arsc contient aussi les identifiants de ces ressources, assignés lors de la compilation de l’application. Il permet à Android de faire le lien entre les identifiants de ces ressources et leurs valeurs, ce qui permet ainsi au développeur de séparer les valeurs des ressources du code de l’application et, par exemple, faciliter le support de plusieurs langues dans l’interface utilisateur de l’application.

3.2.1.4 META-INF

Dossier contenant le fichier MANIFEST.MF, indiquant une somme de contrôle pour chaque fichier dans l’APK; le fichier CERT.SF, contenant le hash de chaque entrée du fichier MANIFEST.MF; et le fichier CERT.RSA qui contient le certificat du développeur ainsi que la signature du fichier CERT.SF.

3.2.1.5 Res

Dossier contenant diverses ressources statiques de l’application. Le type de ces ressources est prédéterminé et ne peut être changé durant l’exécution de l’application. Ces ressources peuvent inclure, entre autres, les chaînes de caractères utilisés dans l’interface utilisateur (ainsi que, si une telle fonctionnalité est implémentée dans l’application, les traductions pour la localisation de ladite interface) ou l’icône de l’application.

3.2.1.6 Assets

Dossier contenant certaines ressources de l’application. Bien que son rôle soit similaire au dossier "res", le dossier "Assets" est surtout dédié au stockage des fichiers bruts. Contrairement au dossier "res", le dossier "assets" n’a aucun requis quant à la structure interne et les fichiers présents dans ce dossier ne sont pas compressés lors de la création du fichier APK de l’application.

3.2.1.7 Lib

Dossier contenant le code compilé en fonction de l'architecture de l'appareil. Ledit code est organisé en sous-dossiers dont le nom indique l'architecture visée. Au moment de la rédaction de ce mémoire, les architectures supportées sont les architectures ARMv7 (ainsi que les versions supérieures), ARM64v8 (ainsi que les versions supérieures), x86-64 et x86.

CHAPITRE 4

APPLICATIONS ANDROID

4.1 Introduction

Il est important d'appronfondir plusieurs aspects des applications Android afin d'expliquer certaines difficultés rencontrées et certains choix méthodologiques effectués. Notamment, le processus de la construction d'une application explique une difficulté majeure rencontrée lors de la décompilation d'une application Android et le système de permission est, en partie, utilisé lors de la détection des notifications poussées. Ce chapitre commence par expliquer le processus de construction d'une application Android et décrit certaines difficultés entourant la décompilation de ces applications. Il énumère ensuite les composantes importantes des applications Android pour montrer les points distinctifs de ces applications. Il présente ensuite le système de permissions d'Android et son implémentation dans les applications Android. Finalement, le chapitre présente la manière avec laquelle les applications installées sont isolées des autres applications.

4.2 Construction d'une application

Les applications Androids sont typiquement construites avec Gradle, qui est un outil d'automatisation de compilation pour les projets Java et Android et se déroule en plusieurs étapes: l'initialisation, la configuration et l'exécution.

4.2.1 Fichiers de construction d'application Android

Il existe trois fichiers clés pour la compilation d'une application Android avec Gradle: le fichier "settings.gradle", le fichier "gradle.properties" et le fichier "build.gradle". Ces fichiers étant uniquement utilisés pour la construction du fichier APK de l'application, ils ne sont pas conservés durant cette opération.

```
pluginManagement {
    repositories {
        google()
        mavenCentral()
        gradlePluginPortal()
    }
}
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
    }
}

rootProject.name = "ToyFinder"
include(":app")
```

Figure 4.1: exemple de contenu d'un fichier settings.gradle pour un projet nommé "ToyFinder" utilisant les dépôts Google et Maven

Le fichier "settings.gradle" (figure 4.1) n'est obligatoire que si l'application Android possède des sous-projets devant être inclus durant la compilation de celle-ci. Il définit la structure de la compilation et contient les informations globales pour l'initialisation de Gradle et la configuration du projet, telles que son nom ou les spécifications des dépôts pour trouver les plugins et les dépendances.

```
org.gradle.jvmargs=-Xmx2048m -Dfile.encoding=UTF-8
#Arguments utilisés pour la machine virtuelle de Java

android.useAndroidX=true
#Drapeau indiquant si l'application utilise la librairie
AndroidX

kotlin.code.style=official
#Drapeau indiquant le style utilisé pour le code Kotlin,
entre "officiel" et "obsolete".
```

Figure 4.2: Exemple de contenu d'un fichier gradle.properties

Le fichier "gradle.properties" (figure 4.2) est utilisé pour définir certaines propriétés de configuration globale, telle que des options de performances pour Gradle, des constantes utilisées par des scripts de compilation ou les plugins ou encore des drapeaux pour le plugin Android.

Le fichier "build.gradle" (figure 4.3) est le coeur de la configuration de compilation d'un projet Android. Ce fichier contient la logique de compilation, de l'assemblage et de la configuration spécifique à chaque module. Il y a, dans un projet Android, plusieurs fichiers de ce type. Un se trouvant dans le répertoire principal, qui sera utilisé pour la configuration commune à tous les modules, ainsi qu'un autre dans chaque répertoire de chaque module de l'application indiquant chacun les configurations de compilation du module concerné.

```
defaultConfig {
    applicationId = "com.example.ToyFinder"
    minSdk = 33
    targetSdk = 34
    versionCode = 1
    versionName = "1.0"
    testInstrumentationRunner =
"androidx.test.runner.AndroidJUnitRunner"
    vectorDrawables {
        useSupportLibrary = true
    }
}
buildTypes {
    release {
        isMinifyEnabled = false
        proguardFiles(
            getDefaultProguardFile("proguard-android-optimize.txt"),
            "proguard-rules.pro"
        )
    }
}
```

Figure 4.3: Extrait d'un fichier build.gradle dans le répertoire d'un module pour une application nommée "ToyFinder"

4.2.2 Processus de construction

Durant l'initialisation, Gradle identifie le ou les projets qui feront partie de l'application construite à l'aide du fichier "settings.gradle" et crée, pour chaque projet détecté, une instance Project en plus d'initialiser les propriétés provenant ou bien de la ligne de commande ou du fichier "gradle.properties". Ensuite, Gradle exécute les fichiers "build.gradle" de chaque projet pour obtenir un graphe orienté acyclique de toutes les tâches qui seront potentiellement exécutées. Les tâches représentées dans ce graphe seront ordonnées selon leurs dépendances respectives aux autres tâches devant être réalisées. L'ordre des tâches est automatiquement déterminé: le fichier "build.gradle" précise uniquement les tâches devant être réalisées (par exemple, déclarer une tâche assembleDebug) sans préciser comment cette tâche doit

être réalisée ni quand elle doit l'être. Durant cette étape, Gradle analyse la configuration Android (tel que le type d'assemblage, les options de compilation de Java ou Kotlin ou encore les variantes d'assemblages). Pour chaque combinaison type/variante, un ensemble de tâches spécifiques sont créées, chacune d'entre elles visant à aboutir à une version différente. Finalement, la troisième et dernière étape consiste en l'exécution des tâches sélectionnées selon l'ordre défini lors de la seconde étape. Les tâches dont l'entrée est restée inchangée ne sont pas exécutées. Les seules tâches exécutées sont les tâches n'ayant jamais été exécutées ainsi que les tâches dont l'entrée a été modifiée.La compilation de l'application en tant que telle se fait, elle aussi, en plusieurs étapes. La première étape consiste en la compilation des ressources de l'application. Pour cette étape, l'Android Asset Packaging Tool est utilisé pour compiler les ressources dans le dossier "res" de l'application – tel que les mises en page ou les chaînes de caractères – et le fichier AndroidManifest dans un fichier de format binaire. Le fichier "R.java", qui contient les ID des ressources de l'application, est créé durant cette étape.
La seconde étape consiste en la compilation du code source de l'application. Étant donné que la DVM (Dalvik Virtual Machine) et l'ART (Android RunTime) n'utilisent pas le bytecode Java, mais le bytecode Dalvik, la compilation du code source d'une application Android développée en Java ou en Kotlin se déroule en trois étapes.

La première étape consiste en la génération du bytecode Java. Le compilateur utilisé dépendra du langage utilisé pour le développement de l'application. Le compilateur javac sera utilisé pour les applications développées en Java et le compilateur Kotlinc sera utilisé pour les applications développées en Kotlin. Si cela est désiré par le développeur, le bytecode ainsi obtenu est traité par Proguard ou R8 (voir section suivante) pour des buts de compression de taille, d'optimisation et d'obfuscation.
Le bytecode ainsi généré est ensuite converti en bytecode Dalvik à l'aide du compilateur R8. Durant cette étape, les bibliothèques tierces – ainsi que tout autre fichier ".class" inclus par le développeur – sont aussi converties et feront partie intégrante de l'application sans distinctions, signifiant ainsi que lesdits fichiers et bibliothèques posséderont les mêmes permissions que l'application.

Le moment où ce bytecode sera compilé dépendra de la version de l'appareil sur lequel l'application est utilisée ou, plus spécifiquement, de l'environnement d'exécution utilisé. En effet, comme mentionné précédemment, la DVM et l'ART ne compilent pas le bytecode Dalvik au même moment. Si l'appareil utilise la version 4.4 ou une version antérieure d'Android (et, conséquemment, exécute les applications

sur la DVM), ladite compilation aura lieu durant l'exécution de l'application par l'utilisateur. Si l'appareil utilise une version d'Android entre 5 et 6 (et exécute donc les applications sur l'ART), alors cette compilation aura lieu au moment de l'installation de l'application sur l'appareil. Finalement, si l'appareil utilise une version plus moderne d'Android, la compilation aura lieu en partie lors de l'installation de l'application et en partie durant l'exécution de l'application – avec, possiblement, des compilations partielles lorsque l'application n'est pas en cours d'utilisation.

La dernière étape de la compilation d'un projet Android est l'empaquetage. Durant cette étape, les fichiers DEX créés lors de l'étape précédente ainsi que les ressources de l'application sont assemblés en un fichier APK qui sera ensuite signé et optimisé pour les spécificités des systèmes Android.

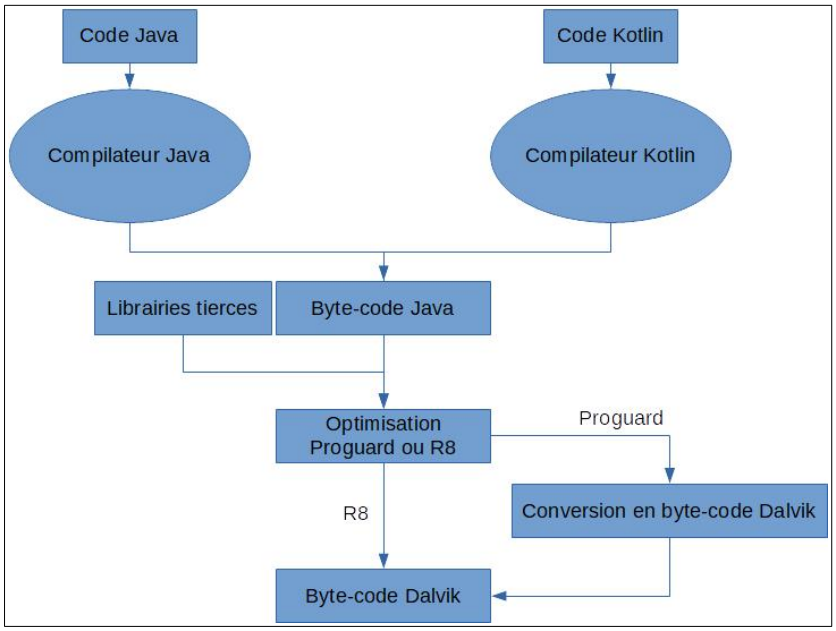

Figure 4.4: Étapes de la compilation du code source d'une application Android

4.2.3 Proguard et R8

Proguard est un outil de réduction, d'optimisation, d'obfuscation et de prévérification publié en 2002 et conçu pour le bytecode Java. Cet outil est utilisé, entre autres, pour le développement des applications Android afin de réduire la taille des applications, les optimiser et rendre plus difficile l'ingénierie inverse des applications une fois publiée. Cet outil est devenu l'outil "officiel" pour la compilation des applications Android en 2010 après avoir été ajouté dans le kit de développement Android lors du

déploiement de la version 2.3 d'Android. Proguard était utilisé durant la compilation de l'application après la création du bytecode Java et avant que celui-ci soit converti en bytecode Dalvik.

L'obfuscation réalisée par Proguard et R8 est relativement simple. Elle consiste principalement à changer le nom des classes, des champs et des méthodes pour des noms plus courts et sans signification – par exemple, une classe nommée "OpponentDriver" pourrait être renommée en "a". Cette obfuscation, bien que simple, permet aussi de réduire la taille de l'application; en plus de la réduction obtenue par l'obfuscation, Proguard et R8 réduisent aussi la taille de l'application via l'élimination du code et des ressources qui ne seront jamais utilisées par l'application.
Notons que ladite obfuscation n'est pas appliquée à toutes les méthodes et classes d'une application Android. Par exemple, les méthodes référencées par le biais d'une réflexion ou celles déclarées dans une interface sont exemptées de l'obfuscation.

La principale différence entre R8 et Proguard est que, contrairement à Proguard, R8 est spécifiquement conçu pour être utilisé pour la compilation des applications Android. Ainsi, alors que Proguard fournit du bytecode Java et travaille uniquement au niveau du bytecode Java, R8 fournit du bytecode Dalvik. En plus de réduire le temps de compilation – puisque, contrairement à Proguard, l'optimisation du bytecode Java et sa traduction en bytecode Dalvik ont lieu lors de la même étape – il permet une meilleure optimisation en étant conscient des contraintes du format DEX.

4.3 Structure d'une application

Les applications Android peuvent être développées avec deux langages de programmation, soit le langage Kotlin et le langage Java; le langage C++ peut aussi être utilisé pour la création de bibliothèques natives. Originellement, le langage recommandé pour le développement des applications Android était le langage Java, avant d'être remplacé en 2019 par le langage Kotlin – une des raisons étant que ce langage est en partie conçu pour le développement des applications mobiles Android. Les applications Android possèdent quatre composantes principales: les activités, les services, les récepteurs de diffusion et les fournisseurs de contenu.

4.3.1 Activités

Les activités sont une composante essentielle aux applications Android et représentent un écran dans l'application. Contrairement à de nombreux langages de programmation, les activités ne sont pas lancées à partir d'un point d'entrée spécifique, mais possèdent plusieurs points d'entrée possibles, chacun correspondant à une étape spécifique du cycle de vie de l'activité et étant pris en charge par une méthode distincte des autres points d'entrées.

```
<activity
    android:name=".MainActivity"
    android:exported="true"
    android:label="@string/app_name"
    android:theme="@style/Theme.HelloWorld">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Figure 4.5: Exemple de déclaration d'une activité dans le fichier Manifest d'une application.

Cette fonctionnalité est présente, car contrairement à un programme classique, le point de départ d'une application mobile n'est pas unique. Une application mobile nécessite donc de prendre en charge une multitude de points d'entrées, chacun pouvant avoir une interface ou un objectif différent des autres points d'entrées – par exemple, l'application ToyFinder pourrait débuter par la page de connexion, lorsque lancée à partir du menu principal de l'appareil, mais afficher directement une page spécifique si lancée lorsque l'utilisateur clique sur un lien Internet dans une application tierce.

Chaque activité doit être déclarée dans le fichier Manifest de l'application. Lors de cette déclaration, plusieurs attributs peuvent être déclarés selon les besoins de l'activité ainsi que l'objectif de celle-ci. L'attribut le plus important, ainsi que le seul attribut devant obligatoirement être déclaré pour chaque activité, est le nom de l'activité. Ce nom n'est pas seulement utilisé par l'application en elle-même, mais peut aussi être utilisé par des applications externes ou par le système Android. Il doit donc, une fois l'application déployée, rester inchangé au risque de compromettre certaines fonctionnalités tel que les raccourcis créés par l'utilisateur. On retrouve aussi des attributs tels que "canDisplayOnRemoteDevices", qui indiquent si l'activité concernée peut être affichée sur un appareil à distance (appareil qui n'utilise

pas obligatoirement le système Android); "screenOrientation", qui indique l'orientation de l'écran demandé par l'activité ou encore "minAspectRatio", qui indique le rapport d'aspect minimal supporté. Notamment, on retrouve aussi les attributs permettant l'utilisation d'*intents* et les permissions d'utilisation, les deux attributs étant étroitement liés.

Pour reprendre l'exemple de ToyFinder, les activités de l'application seront utilisées pour chaque fenêtre de l'application, que ce soit pour afficher les résultats de la dernière analyse, pour afficher la politique de confidentialité de l'application, pour lister les liens de contact du studio de développement de l'application ou encore pour afficher les résultats des recherches précédentes.

4.3.1.1 Intents

Les *intents* sont des messages permettant à différentes composantes du système, tel qu'une activité, un service ou un récepteur de diffusion, de demander des actions entre eux. Les *intents* communiquent au système Android l'action devant être réalisée ainsi que la composante devant réaliser l'action en question. Ils peuvent être utilisés par l'application les déclarant pour accéder à une autre activité tout comme ils peuvent être utilisés par une application externe. Les intents sont, pour une application utilisant les fichiers XML pour construire les interfaces, un des principaux moyens de changer d'activité durant l'exécution. Il y a deux principaux types d'*intent*: les *intents* explicites et les *intents* implicites.

Les *intents* explicites sont des *intents* dans lesquels la composante devant traiter la requête est identifiée, par exemple lorsqu'ils sont utilisés dans le but de lancer une activité spécifique de l'application. Les *intents* implicites, au contraire, ne spécifient pas la composante devant traiter la requête. Plutôt, ils déclarent une action générale et le système Android détermine quelle composante est capable de la gérer, comme ouvrir une page web ou prendre une photo.

Dans le cas de ToyFinder, les *intents* explicites seront utilisés pour naviguer entre les diverses activités et interfaces de l'application. Les *intents* implicites seront utilisés pour accéder à l'application de photographie de l'utilisateur lorsque celui-ci désire prendre une photo du jouet devant être recherché par l'application.

4.3.2 Services

Les services sont des points d'entrées qui permettent à une application d'être exécutée en arrière-plan – un exemple serait une application permettant à l'utilisateur d'écouter de la musique lorsque l'appareil est verrouillé ou encore une application de navigation permettant de suivre la position de l'utilisateur en continu. Contrairement aux activités, un service ne possède pas d'interface utilisateur. Il y a deux types de services: le service démarré et le service lié.

Les services démarrés désignent les services qui, une fois lancés, continueront à fonctionner jusqu'à ce que leur tâche soit accomplie ou qu'ils soient explicitement arrêtés par l'application. Ce type de service est aussi composé de deux types de services, selon l'usage prévu par ce type de service: les services en premier plan et les services en arrière-plan.
Les services en premier plan sont des services qui visent à être utilisés pour des opérations dont l'utilisateur est conscient de l'exécution du service, comme un appel ou de la lecture de musique. Une fin prématurée d'un tel service, étant directement utilisée par l'utilisateur, serait remarquée immédiatement et causerait une dégradation significative de l'expérience de l'utilisateur et, conséquemment, son exécution est priorisée par le système afin de garantir un maintien de l'exécution. Ces services doivent afficher une notification annonçant l'exécution du service durant leurs exécutions et ne peuvent retirer cette notification tant que le service est en cours d'exécution. Ce type de service démarré pourrait, par exemple, être utilisé par une application d'appel afin de garantir que le système Android priorise le maintien de l'appel et ainsi éviter une coupure de l'appel si les ressources de l'appareil ne sont pas suffisantes pour l'exécution d'autres tâches moins importantes ou pouvant être mises en pause temporairement.

Les services en arrière-plan sont des services qui visent à être utilisés pour des opérations dont l'usager n'a pas directement conscience de l'exécution. Une fin prématurée des services liés, n'étant pas directement utilisés par l'utilisateur, ne causerait pas une dégradation significative de l'expérience de l'utilisateur. Du fait de cette importance moindre, le système a une plus grande liberté quant au maintien de l'exécution de ce type de service et peut, par exemple, mettre fin à cette exécution pour la continuer plus tard si le service en question demande trop de ressources pour être maintenu en même temps qu'un service en premier plan.

Le second type de service est le service lié. Contrairement au service démarré, ce type de service ne demande pas au système de continuer son exécution, mais est exécuté lorsqu'une application ou le système Android demande l'utilisation du service en question. Durant l'utilisation du service lié, qui est réalisé par le biais d'une API fournie par le service, le lien entre l'utilisateur du service et le service est conservé. L'appareil est en mesure de savoir pour quel composante le service doit être maintenu en plus de l'importance du maintien du service: si l'utilisateur du service est important, le maintien du service sera lui aussi important.

```
<service
    android:name=".MyService"
    android:enabled="true"
    android:exported="false" />
```

Figure 4.6: Exemple de déclaration d'un service démarré dans un fichier Manifest d'une application Android

Les services démarrés seront utilisés par ToyFinder pour la recherche des commerces après que l'application ait analysé une photo prise par l'utilisateur et pour suivre la position de l'utilisateur durant l'utilisation de l'application. Ce service sera un service en arrière-plan, cette opération pouvant être mise en pause sans causer un désagrément significatif à l'utilisateur. Les services liés seront utilisés dans le but de suivre la position de l'utilisateur lors de la recherche des commerces offrant le jouet identifié et ainsi permettre à l'application de prioriser les commerces se trouvant à proximité de l'utilisateur.

4.3.3 Récepteur de diffusions ("*broadcast receivers*")

Les récepteurs de diffusions font partie du mécanisme publish-subscribe d'Android. Ces diffusions peuvent être émises autant par le système que par une application et de ce fait, les applications peuvent déclarer quelle catégorie de diffusion elles désirent écouter. Lorsqu'une diffusion est émise, que ce soit par le système ou par une application, le système envoie ladite diffusion aux applications ayant déclaré vouloir recevoir ces diffusions.

Les récepteurs de diffusions sont la composante d'une application permettant à l'application de recevoir une diffusion dont elle est à l'écoute. Ces récepteurs de diffusions étant des points d'entrée de l'application, les applications n'ont pas besoin d'être en cours d'exécution pour recevoir une diffusion et peuvent ainsi être lancées automatiquement lorsque certains événements se produisent. Par exemple, il

est possible pour une application de messagerie de produire automatiquement une notification lorsqu'un SMS est reçu par le téléphone ou encore qu'une application jouant de la musique mette automatiquement la lecture de la musique en pause si des écouteurs sont débranchés de l'appareil.

Les récepteurs de diffusions seront utilisés par ToyFinder afin d'être avertis par le système si la charge de la batterie est faible et ainsi ajuster le comportement de l'application en conséquence, par exemple en proposant à l'utilisateur de conserver la photo de l'objet en mémoire pour procéder à l'analyse, opération couteuse en énergie, lorsque la batterie sera à un niveau plus élevé.

4.3.4 Fournisseurs de contenu ("*content providers*")

Les fournisseurs de contenus sont des composantes visant à gérer des ensembles de données partagées pouvant être conservés dans la mémoire de l'appareil. Plus spécifiquement, les fournisseurs de contenus visent à gérer l'accès d'une application à la couche de données de celle-ci et, entre autres, permettre un partage de données entre plusieurs applications ainsi que l'envoi de données vers les composantes graphiques de l'application. Les fournisseurs de contenu utilisent, pour représenter les données, des tables similaires à celles utilisées dans une base de données relationnelles, les colonnes indiquant chaque élément de données collectées et les lignes représentant chaque instance des données collectées.

NAME	calendar_display_name	visible	sync_events
calendrier_1	Calendrier	1	0
calendrier_2	Autre_calendrier	0	1
calendrier_3	Un_autre_calendrier	1	1
calendrier_4	un_dernier_calendrier	0	0

Figure 4.7: Exemple d'une représentation de données de la table CalendarContract.Calendars

Dans la figure 4.7, les rangées représentent un calendrier dans l'appareil. Les quatre colonnes ("NAME", "calendar_display_name", "visible" et "sync_events") représentent respectivement le nom du calendrier, le nom devant être affiché à l'utilisateur, une booléenne indiquant si les événements associés au calendrier doivent être affichés à l'utilisateur et une booléenne indiquant si le calendrier doit être

synchronisé et conserver ces événements localement sur l'appareil ou si ceux-ci doivent uniquement être conservés sur un serveur externe. Par exemple, dans la même figure, la première rangée représente un calendrier nommé "calendrier_1", dont le nom affiché à l'utilisateur est "Calendrier" et dont les événements associés à ce calendrier seront affichés, mais pas synchronisés.
L'application ToyFinder ferait usage des fournisseurs de contenus dans le but d'accéder aux fichiers de l'appareil, permettant ainsi à l'utilisateur de rechercher des jouets ayant été pris en photo dans le passé ou dont l'image a été retrouvée sur Internet.

4.4 Permissions

Un point important pour la sécurité de l'architecture Android est que toute application soit, par défaut, empêchée de faire des opérations pouvant avoir un impact néfaste sur l'appareil, l'usager ou d'autres applications installées sur l'appareil – par exemple en manipulant des fichiers d'une autre application, en accédant à des données jugées sensibles (tel que les contacts de l'utilisateur) ou encore en utilisant les capteurs présents sur l'appareil. Du fait de l'architecture du système Android, une application est isolée autant du système que des autres applications présentes sur l'appareil.

Étant donné les limites importantes qu'une isolation totale d'une application causerait au niveau des fonctionnalités possibles – une application serait, par exemple, incapable de faire vibrer l'appareil - Android implémente un système de permissions. Ce système de permissions permet autant aux applications qu'au système lui-même d'exposer certaines fonctionnalités et données aux autres applications.

```
<uses-permission android:name="android.hardware.sensor.proximity"/>
<uses-permission android:name="android.hardware.sensor.gyroscope"/>
<uses-permission
android:name="android.permission.HIGH_SAMPLING_RATE_SENSORS"/>
```

Figure 4.8: Exemple de déclaration des permissions autorisant l'utilisation du capteur de proximité, du gyroscope et de l'échantillonnage à haute fréquence

Avec ce système, une application ayant besoin de faire une opération normalement interdite doit déclarer ce besoin dans le fichier Manifest de l'application. Le fichier Manifest étant statique et ne pouvant être modifié après la compilation de l'application, une application ne peut pas demander de nouvelles permissions au système durant l'exécution de l'application. De plus, étant donné que le fichier

Manifest est distinct du code exécutable de l'application et déclaratif, l'implémentation de ce système de permissions permet au système Android de connaitre les permissions demandées par une application sans avoir besoin d'exécuter ladite application.

Il est possible pour une application de définir des permissions personnalisées. Une telle définition est composée de deux parties: le nom de la permission et le niveau de protection de celle-ci. Le niveau de protection est lui-même composé de deux aspects: le type de protection et des modificateurs optionnels.

```
<permission
android:name="com.example.ToyFinder.permission.MA_PERMISSION"
      android:label="@string/labelPermission_maPermission"
      android:description="@string/permissionDesc_maPermission"
      android:protectionLevel="dangerous"/>
```

Figure 4.9: Exemple de déclaration d'une permission personnalisée

Il y a, à la version 16 d'Android, trois principaux types de permissions: les permissions à l'installation (*install-time*), les permissions à l'exécution (*runtime*) et les permissions spéciales. Chaque type indique le niveau d'accès/actions que l'application obtient lorsque la permission est octroyée. L'appartenance à une catégorie est basée sur le risque posé de chaque permission. Ces permissions, peu importe le type de celles-ci, doivent être déclarées dans le fichier Manifest de l'application .

4.4.1 Permission à l'installation ("*install-time*")

Le premier type de permission, soit les permissions *install-time*, désigne des permissions permettant un accès limité aux données sensibles ainsi que des actions dont l'impact sur l'appareil de l'utilisateur est limité. Ces permissions sont, contrairement aux permissions des autres types, accordées automatiquement à l'application lors de son installation et doivent, si l'utilisateur ne désire pas que l'application possède ces permissions, être manuellement révoquées via les réglages de l'appareil. Les permissions de ce type sont aussi subdivisées en deux sous-types de permissions: les permissions normales et les permissions dites "signatures". Les permissions "normales", comme leur nom l'indique, n'ont pas de réelles particularités.

Plusieurs permissions utilisées par ToyFinder appartiennent à cette catégorie de permission, notamment la permission d'accéder à Internet (nécessaire pour rechercher les commerces à proximité de l'utilisateur) ainsi que la permission de faire vibrer l'appareil (utilisé pour avertir l'utilisateur que la recherche est complétée).

4.4.1.1 Permissions "signatures"

Les permissions "signatures" diffèrent des permissions normales. En effet, alors qu'une permission normale est octroyée dès lors qu'elle est déclarée dans le fichier Manifest de l'application, une permission "signature" est uniquement octroyée par le système Android si l'application requérante est signée avec la même clé que celle utilisée pour signer l'application ou le composant du système ayant défini ladite permission. Ce type de permission permet ainsi au système Android ou à une application de définir une permission exposant certaines fonctionnalités ou données tout en garantissant que seules les applications autorisées par le développeur puissent utiliser ladite permission.
Ce type de permission est particulièrement utilisé par Android pour des permissions qui ne peuvent être utilisées que par les applications système. Un exemple notable est la permission "BIND_ACCESSIBILITY_SERVICE". Cette permission, visant à offrir des fonctions d'accessibilité, est hautement sensible, car elle donne la capacité, entre autres, de lire les éléments de l'interface utilisateur affichés à l'écran ainsi que d'exécuter des gestes à la place de l'utilisateur.

4.4.2 Permissions à l'exécution (*runtime*)

Ce type de permission a été implémenté à la version 6 (API 23) d'Android. Elle vise à offrir un niveau de protection plus élevé à l'utilisateur ainsi qu'un contrôle plus fin des permissions des applications. Ce type de permission regroupe des permissions jugées plus sensibles et nécessitant un consentement explicite et éclairé de l'utilisateur, tel que la permission permettant la localisation précise ou la prise de photographie.

Les permissions *runtime* sont, tout comme les permissions *install-time*, déclarées dans le fichier Manifest de l'application. Cependant, alors que les permissions *install-time* sont automatiquement accordées à l'application les demandant lors de l'installation, les permissions *runtime* sont, par défaut, refusées à

l'application. Pour qu'une application reçoive une permission de ce type, celle-ci doit demander à l'utilisateur de confirmer l'accord de la permission désirée par l'application. Si l'utilisateur accepte la demande, la permission est accordée à l'application en question.

Ce type de permission peut être accordé à une application uniquement pour la session en cours (i.e.: la permission sera révoquée lors de la fermeture de l'application) ou perpétuellement. Il faut noter que la fonctionnalité permettant d'accorder une permission *runtime* à une application uniquement jusqu'à la fermeture de celle-ci a été ajoutée à la version 11 d'Android et que, depuis cette même version, les permissions *runtime* ne sont pas véritablement accordées perpétuellement. En effet, les permissions *runtime* d'une application sont automatiquement révoquées si l'application en question n'a pas été utilisée pour plusieurs mois, à l'exception des applications possédant un rôle qui, contrairement aux autres, ne perdent leurs permissions que si elles sont manuellement retirées par l'utilisateur ou si le rôle est repris par une autre application.

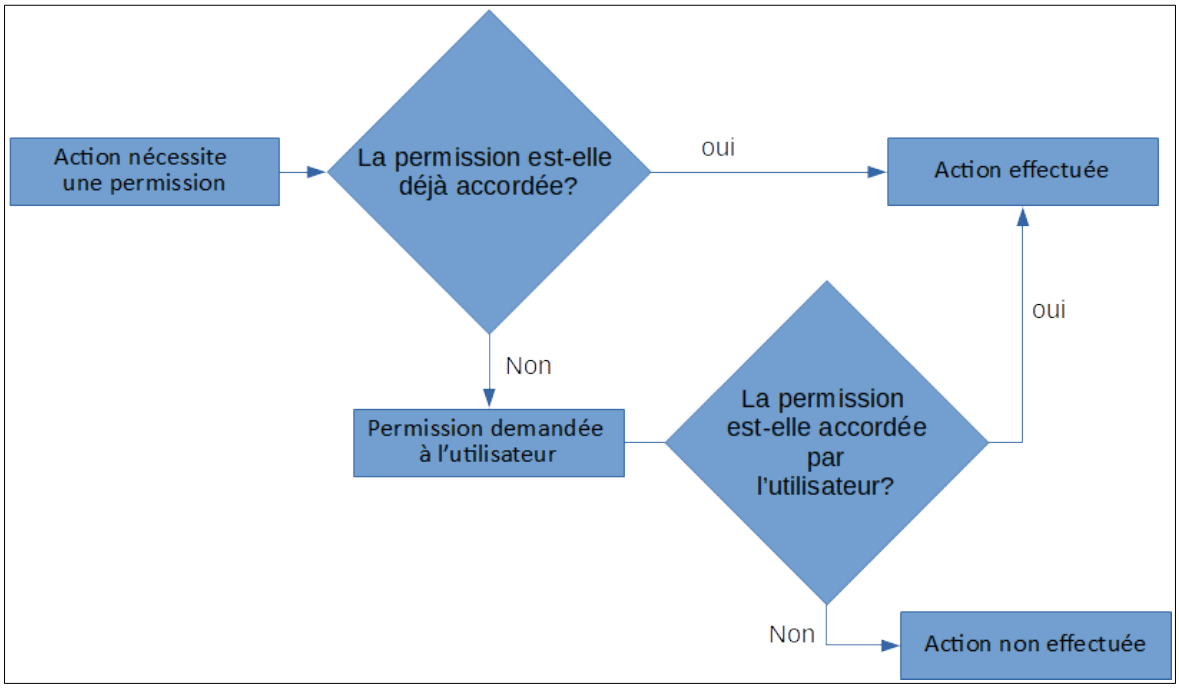

Figure 4.10: Étapes d'une demande d'une permission à l'exécution

Plusieurs permissions utilisées par ToyFinder font partie de cette catégorie de permission, notamment la permission d'accéder à la position de l'appareil et la permission d'accéder aux fichiers présents dans l'appareil.

4.4.3 Permissions spéciales

Les permissions spéciales, contrairement aux permissions *install-time* et aux permissions *runtime*, ne peuvent pas être définies par une application tierce, mais uniquement par la plateforme et les manufacturiers d'appareil Android. Les permissions de ce type sont considérées comme plus sensibles que les permissions *runtime* et, conséquemment, le processus utilisé par les applications pour recevoir les permissions de ce type diffère significativement de celui utilisé pour les permissions *runtime*. En effet, contrairement aux permissions *runtime*, une application nécessitant une permission spéciale ne peut pas recevoir ladite permission en demandant une confirmation auprès de l'utilisateur. Plutôt, celle-ci doit informer l'utilisateur du besoin et l'inviter à accéder aux réglages de l'appareil pour accorder la permission requise.

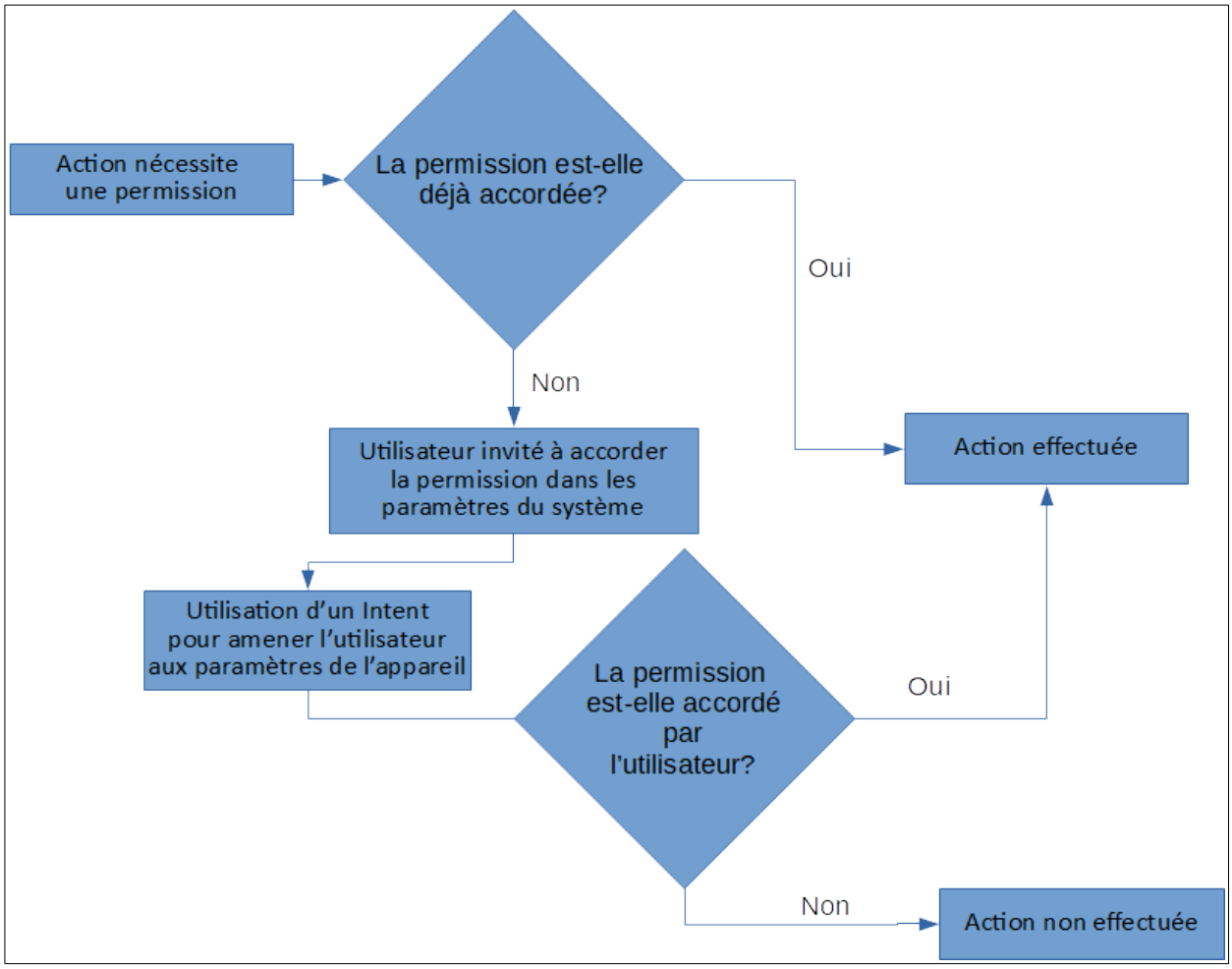

Figure 4.11: Étapes d'une demande d'une permission spéciale

4.4.4 Rôles

Un autre aspect important des permissions dans les systèmes Android est un concept implémenté à la version 10 d'Android : les rôles. Ce système a pour but de simplifier et faciliter la gestion de certaines permissions associée à des rôles spécifiques — par exemple, l'application de messagerie ou le navigateur Internet. Avec ce système, chaque rôle possède, par défaut, des permissions appropriées pour accomplir sa mission. Chaque rôle consiste en un nom unique dans le système Android et est associé à des permissions et privilèges spécifiques à ce rôle. Une application étant désignée comme l'application possédant un rôle devient l'application utilisée par défaut par l'appareil pour les fonctionnalités associées au rôle en question.

Les permissions associées aux rôles ne diffèrent pas des permissions décrites précédemment. La différence ne porte pas sur des permissions en elle-même, mais bien au processus de l'octroi de ces permissions. En effet, alors qu'une application « normale » doit demander toutes les permissions requises à l'utilisateur, les rôles permettent à une application d'obtenir un ensemble de permissions en demandant le rôle à l'utilisateur, par exemple en demandant de devenir l'application par défaut pour la fonctionnalité associée au rôle. Par exemple, lorsqu'un utilisateur accepte qu'une application devienne l'application de messagerie par défaut de l'appareil, les permissions associées à cette fonctionnalité (tel que l'envoi de SMS) sont automatiquement accordées à cette application et le rôle est retiré à l'ancienne application de messagerie.

De plus, comme mentionné précédemment, les permissions associées à un rôle ne sont pas révoquées après plusieurs mois d'inutilisation. Cette particularité provient de l'objectif du système de rôle : désigner une application "officielle" dans l'appareil pour offrir une fonctionnalité donnée. Pour reprendre l'exemple d'une application de messagerie, si les permissions étaient retirées comme les autres applications, cela signifierait que les permissions nécessaires pour faire apparaître des notifications ou lire les SMS reçus seraient retirées si l'utilisateur n'utilise pas l'application pour plusieurs mois. Sans ces permissions, l'application ne serait plus en mesure ni de recevoir des SMS ni de produire une notification lors de la réception d'un SMS, ce qui, pour des raisons évidentes, poserait de nombreux désavantages pour une telle application.

4.5 Isolation des applications

Pour garantir que chaque application est isolée des autres applications installées dans l'appareil, Android se base sur le modèle de sécurité de Linux ainsi que le module SELinux[1]. Plus précisément, la capacité d'isoler les applications et fichiers dans leur bac à sable du système Android repose sur une adaptation du mécanisme multi-utilisateur déjà implémenté dans le système Linux.

Lorsqu'une nouvelle application est installée dans un appareil Android, le système lui assigne un identifiant utilisateur unique (UID). Cet identifiant correspond au mécanisme multi-utilisateur de Linux et est utilisé par celui-ci pour représenter les différents utilisateurs. Conséquemment, chaque application est, pour le noyau Linux, considérée comme un utilisateur distinct et ses processus s'exécutent sous cet identifiant. De ce fait, sauf en cas d'attaque ou d'exploitation d'une vulnérabilité du système Linux, chaque application n'a accès qu'aux ressources pour lesquelles son identifiant dispose des permissions nécessaires et est dans l'impossibilité de lire ou modifier les fichiers associés à d'autres UIDs. Chaque application, lors de son installation, reçoit un répertoire privé qui appartient au UID de l'application, garantissant ainsi que seule cette application pourra y accéder.

Un des grands avantages de ce système réside dans le fait que ce bac à sable est implémenté au niveau du système d'exploitation. Ainsi, l'ensemble de la pile logicielle au-dessus du noyau est contrainte par les règles d'isolation propres à chaque application, y compris le code natif ou les applications système. Les bibliothèques du système d'exploitation, étant chargées dans le processus de l'application, sont aussi dans l'impossibilité d'accéder aux ressources d'une autre application à moins que l'application appelant ces bibliothèques dispose des permissions requises.

Depuis Android 4.4, le système Android intègre SELinux pour renforcer la sécurité globale de la plateforme. SELinux est un module pour les systèmes Linux visant à apporter une meilleure sécurité au système en ajoutant un mécanisme de contrôle d'accès obligatoire (Mandatory Access Control, ou MAC). Ce mécanisme est, dans les systèmes Android, appliqué à tous les processus, y compris les processus exécutés avec les privilèges *root*.

Traditionnellement, Linux repose sur un modèle de contrôle d'accès discrétionnaire (de l'anglais

1 Security-Enhanced Linux

Discretionary Access Control, ou DAC). Dans ce modèle, le propriétaire d'un fichier ou d'une ressource décide qui peut y accéder et avec quels droits parmi la lecture, l'écriture ou l'exécution. L'utilisateur *root* a un accès total sur tous les fichiers et ressources du système.

L'approche introduite par SELinux repose plutôt sur un modèle de contrôle d'accès obligatoire qui vient s'ajouter au contrôle d'accès discrétionnaire et applique en permanence des politiques de sécurité. Ces politiques spécifient précisément ce que chaque processus est autorisé à faire, les appels système qu'ils peuvent invoquer ainsi que la manière dont ils peuvent interagir avec les autres composants du système. Contrairement au contrôle d'accès discrétionnaire, le contrôle d'accès obligatoire utilisé par SELinux permet ainsi d'imposer des restrictions au processus *root*, limitant de manière stricte son champ d'action. Il est ainsi possible, par exemple, de garantir qu'une application malicieuse ayant exploité une vulnérabilité pour recevoir les privilèges *root* ne soit tout de même pas en mesure d'utiliser ces privilèges pour accéder à des fichiers ou effectuer des actions qui lui sont normalement interdites.

4.6 Initiative *Privacy Sandox*

Comme mentionné précédemment, les permissions accordées à l'application sont aussi accordées aux bibliothèques intégrées dans ladite application. Ces bibliothèques sont aussi exécutées dans le bac à sable de l'application. Cette caractéristique du système Android, bien qu'elle permette l'intégration des bibliothèques plus facilement, cause un risque important au niveau de la confidentialité et ouvre la porte à des abus de la part des réseaux de publicités. En effet, les bibliothèques fournies par un réseau de publicité, puisqu'elles possèdent les mêmes permissions que l'application, possèdent un grand contrôle sur l'application (Su et al., 2018) et peuvent faire usage de ces mêmes permissions pour collecter les mêmes informations que l'application en elle-même peut collecter. Par exemple, une bibliothèque de publicité installée sur une application de messagerie possèdera les permissions donnant accès à la liste des contacts de l'utilisateur, permettant ainsi au réseau de publicité de récupérer les nom et numéro de téléphone des contacts de l'utilisateur.

Cette problématique cause aussi un problème au niveau du consentement de l'utilisateur par rapport aux permissions accordées à l'application. En effet, pour reprendre l'exemple d'une bibliothèque de publicité, bien que celle-ci ne soit pas en mesure d'accéder à ces permissions que si l'utilisateur donne lesdites permissions à l'application, ce consentement n'est pas éclairé. Lorsqu'une permission est demandée à

l'utilisateur, le système Android ne spécifie pas si ladite permission sera utilisée par l'application en elle-même ou par la bibliothèque de publicité incluse dans l'application.

Pour cette raison, depuis la version 14 d'Android, des mesures sont prises pour renforcer la protection des données privées des utilisateurs tout en continuant à supporter les services de publicité et d'analyse dans le cadre de l'initiative *Privacy Sandbox*. Pour atteindre cet objectif sur les plateformes Android, *Privacy Sandbox* propose deux solutions clés: un environnement d'exécution des kits de développement logiciel et un ensemble d'APIs offrant une meilleure protection des informations privées.

4.6.1 Environnement d'exécution des kits de développement logiciel (*SDK runtime*)

Introduit avec la version 14 d'Android en 2023, l'environnement d'exécution des kits de développement logiciels vise principalement à mitiger le risque posé par l'héritage des permissions par les bibliothèques des réseaux de publicités. Bien qu'étant déjà implémenté, le développement de cette fonctionnalité est encore en cours et est considéré en phase de test et de transition lors de la version d'Android la plus récente (Android 16).[2]

Au coeur de cette solution se trouve la possibilité d'exécuter les kits de développement et l'application en elle-même dans deux processus distincts. À ce jour, cette isolation n'est pas obligatoire et fonctionne uniquement avec les kits compatibles; dans les cas où le kit n'utilise pas cet environnement d'exécution, ou n'est pas compatible, l'exécution dudit kit aura lieu dans le même processus que l'application. Cette séparation implique que les permissions possédées par les kits et par l'application ne seront pas obligatoirement les mêmes: les bibliothèques disposeront des permissions propres et définies de manière indépendante.

Les bibliothèques exécutées dans l'environnement d'exécution des kits de développement logiciel ont par défaut plusieurs permissions: "INTERNET" (soit l'accès à Internet, qui est nécessaire pour communiquer avec les réseaux de publicités), "ACCESS_NETWORK_STATE" (qui permet d'accéder à des informations basiques au sujet des réseaux Internet de l'appareil tel que l'état de la connexion), "READ_BASIC_PHONE_STATE" (qui donne accès à des informations de base au sujet de la téléphonie telle

2 Le projet a, durant l'écriture du mémoire, été abandonné

que le type de réseau téléphonique utilisé par l'appareil), "AD_ID" (qui octroie la possibilité d'accéder à l'identifiant publicitaire du téléphone) et les permissions d'accès aux APIs développées dans le but d'offrir une meilleure protection des informations privée de l'utilisateur.

En parallèle, un nouveau modèle de distribution reposant sur une séparation claire entre l'application et les kits de développement logiciel est introduit. Concrètement, les kits ne sont plus liés statiquement et assemblés avec les applications les utilisant, mais sont distribués sous forme d'APKs indépendant du code de l'application. Dans ce modèle, les développeurs de kits publient directement leurs kits versionnés sur les boutiques d'applications alors que les développeurs des applications, au lieu d'assembler les trousses avec l'application, déclarent leurs dépendances vis-à-vis de ces trousses dans l'application avant de distribuer l'application sur la boutique d'application offrant les kits. Lors du téléchargement de l'application par l'utilisateur, la boutique se charge automatiquement de récupérer et d'installer les kits nécessaires.

4.6.2 API de protection de la confidentialité

Un autre aspect important du projet consiste en des APIs permettant une meilleure protection de la confidentialité. Actuellement, le profilage des utilisateurs est fait avec l'aide d'identifiants interapplications – notamment avec l'a*dvertising ID* (AdID) de Google. Cet identifiant, assigné à un appareil spécifique, peut être accédé par les réseaux de publicités et les développeurs des applications pour suivre le comportement de l'utilisateur. Cet identifiant étant unique et pouvant être utilisé simultanément par plusieurs applications, il peut être utilisé pour observer le comportement d'un utilisateur à travers toutes les applications utilisant un tel identifiant afin de construire un profil centralisé de l'utilisateur sans nécessiter une collaboration volontaire entre ces applications. Ainsi, un utilisateur jouant à des jeux de combat et à des simulateurs de vol pourrait se faire cibler pour des publicités de jeu de combat aérien sans que les applications ou les développeurs connaissent l'existence de l'autre application, pourvu que les applications utilisent le même kit de développement pour les publicités.

Cet agrégat de données représente un problème pour la protection des informations privées de l'utilisateur et représente donc un aspect important pour le projet, dont l'objectif est d'offrir une meilleure protection de la confidentialité des utilisateurs. Afin de résoudre ce problème tout en

permettant le ciblage publicitaire, le projet *Privacy Sandbox* amène une nouvelle approche du suivi publicitaire: les informations personnelles ne sont plus conservées sur les serveurs des réseaux de publicités, mais dans la mémoire de l'appareil de l'utilisateur. De plus, afin de limiter les problèmes liés aux identifiants personnels, cette approche opère sans utiliser un identifiant individuel. Pour atteindre ce but, deux concepts sont utilisés: les *thèmes* et l'API *Protected Audience*.

Les *thèmes* visent à préserver la confidentialité tout en permettant un ciblage publicitaire basé sur les centres d'intérêt de l'utilisateur. Ces centres d'intérêt sont déduits de l'activité de l'utilisateur, tels que les applications utilisées ou les sites Internet visités. Ils consistent en un ensemble de "thèmes" qui suivent une taxonomie structurée – tel que "/Arts & Entertainment/Music & Audio/Soul & R&B" – qui est utilisée pour refléter les centres d'intérêt de l'utilisateur en question. L'API *Topics* est utilisé pour observer et accéder aux centres d'intérêt correspondant aux intérêts de l'utilisateur. Une observation se produit lorsqu'un appelant - généralement un kit de développement logiciel intégré dans l'application - invoque l'API pour observer l'activité de l'utilisateur. Par exemple, un kit présent dans une application pourrait faire appel à cet API pour indiquer que le *thème* "/Autos & Vehicles/Motor Vehicles (By Type)/Hatchbacks" a été observé par le kit en question lorsque ledit utilisateur utilise une application de vente de voiture pour s'informer des prix de certains modèles de voitures. Une fois collectées, ces données sont utilisées pour déduire les principaux centres d'intérêt de l'utilisateur. Lorsqu'une application désire afficher une publicité ciblée à l'utilisateur, le kit utilise l'API pour obtenir jusqu'à trois *thèmes* représentant les centres d'intérêt de l'utilisateur. Ces *thèmes* sont ensuite transmis au réseau de publicité qui les utilisera pour choisir une publicité ciblée aux intérêts de l'utilisateur sans qu'un identifiant personnel soit associé aux centres d'intérêt en question.

Notons que, comme les catégories ayant été observées sont conservées localement dans l'appareil, un utilisateur possédant deux appareils distincts pourrait ne pas avoir les mêmes centres d'intérêt dans les deux appareils. De plus, l'activité observée étant spécifique aux réseaux de publicités, deux réseaux distincts pourraient ne pas recevoir les mêmes thèmes si ces réseaux ne sont pas présents dans les mêmes applications utilisées par l'utilisateur.

4.6.3 API *Protected Audience*

L'API *Protected Audience* regroupe deux catégories principales: l'API *Custom Audience* et l'API *Ad Selection* qui sont utilisés pour le remarketing, soit la diffusion de publicité ciblée en fonction des interactions passées de l'utilisateur - par exemple une application de billetterie désirant cibler les utilisateurs ayant précédemment acheté un billet pour un concert d'un groupe spécifique - tout en limitant les informations personnelles transmises aux réseaux de publicités. Pour atteindre cet objectif, le processus de sélection des publicités à afficher à l'utilisateur ainsi que la logique de ciblage des utilisateurs est déplacé: au lieu d'avoir lieu sur les serveurs des réseaux de publicités, elle a lieu dans l'appareil de l'utilisateur. De plus, tout comme les thèmes présentés précédemment, les informations sont conservées dans l'appareil de l'utilisateur et ne sont pas transmises aux réseaux de publicités.

4.6.3.1 API *Custom Audience*

Cette API fournit une abstraction pour des groupes d'audience, soit un groupe d'utilisateurs possédant un comportement ou une intention commune – par exemple, les utilisateurs ayant précédemment procédé à une microtransaction ou ayant complété le premier niveau dans un jeu. Contrairement aux thèmes, qui sont partagés au réseau de publicité l'utilisant, l'appartenance à un groupe d'audience ne peut être partagée entre les différentes applications ou sur le web.

Le groupe d'audience n'est pas juste un indicateur binaire: il contient également un ensemble d'informations et de métadonnées détaillées. Parmi celles-ci figurent entre autres le propriétaire (soit le nom du paquet de l'application), le gestionnaire du groupe (soit le réseau de publicité prenant en charge le groupe d'audience), le nom du groupe d'audience, la fenêtre d'activation du groupe, l'URL devant être utilisé pour récupérer les publicités candidates et l'URL devant être utilisé pour récupérer la logique de sélection de la publicité.

4.6.3.2 API *Ad Selection*

L'API *Ad Selection* est responsable de la sélection de la publicité qui sera affichée à l'utilisateur. Elle est conçue pour que ladite sélection soit effectuée sans communiquer à quel groupe d'audience l'utilisateur

appartient. Lorsqu'une publicité doit être affichée dans une application, le SDK utilise l'API *Ad Selection* en plus d'indiquer la liste des acheteurs autorisés à participer à une enchère pour l'achat de l'affichage de la publicité.

Le processus de sélection repose sur deux étapes principales: l'enchère côté achat et la logique de sélection coté vente. Lors de la première étape, l'API parcourt l'appareil de l'utilisateur afin de déterminer quels sont les groupes pertinents pour les acheteurs potentiels auxquels l'utilisateur appartient. Pour chaque groupe identifié, l'URL de l'algorithme d'enchère correspondant est récupéré et est exécuté pour déterminer le "score" de chaque publicité potentielle.

Lorsque l'enchère est terminée, intervient la seconde étape. Lors de cette étape, les scores de chaque publicité ayant participé à l'enchère sont utilisés pour décider la publicité devant être affichée à l'utilisateur. Ce choix est, tout comme l'étape précédente, effectué sur l'appareil de l'utilisateur avec un algorithme fourni par le réseau de publicité. Finalement, une fois que la publicité est choisie, le choix est transmis au réseau de publicité afin que la publicité soit affichée à l'utilisateur. Ainsi, étant donné qu'autant l'enchère que le choix de la publicité à afficher sont effectués dans l'appareil de l'utilisateur, aucune information potentiellement confidentielle de l'utilisateur n'est transmise aux réseaux de publicité tout en permettant néanmoins le remarketing.

4.7 Conclusion

Ce chapitre a présenté plusieurs concepts importants liés aux applications Android. Après avoir décrit le processus de construction d'une application Android en plus des fichiers importants pour ladite construction, le chapitre a présenté les composantes principales des applications Android, soit les activités, les intents, les services, les récepteurs de contenu et les fournisseurs de contenu. Le système de permissions utilisé par le système Android a ensuite été décrit avant d'expliquer les mécanismes d'isolation des applications dans un système Android. Finalement, le chapitre a dressé un portrait d'un projet visant à apporter une meilleure protection des informations personnelles de l'utilisateur.

CHAPITRE 5

COLLECTE ET DÉCOMPILATION DES APPLICATIONS ANDROID DE L'ÉCHANTILLON

5.1 Introduction

L'approche utilisée pour la détection des anti-fonctionnalités repose sur l'analyse statique du code source de l'application analysée. Il a donc été nécessaire de mettre au point un pipeline pour décompiler les fichiers APK des applications devant être analysées. De plus, la majorité des applications de l'échantillon utilisées pour les expérimentations recevant encore des mises à jour régulières durant lesdites expérimentations, il a aussi été nécessaire de développer une méthode permettant d'obtenir une version spécifique d'une application donnée. Ce chapitre commencera par décrire la méthode ayant été mise au point pour obtenir les fichiers APK des applications de l'échantillon avant de décrire les outils et pipelines ayant été utilisés pour décompiler ces applications.

5.2 Collecte des applications

L'échantillon d'applications utilisées pour les expérimentations faites dans ce mémoire a été fourni par une équipe dirigée par la Professeure Maude Bonefant et consiste en un tableur. Ce fichier indique, pour chaque application, des informations sur plusieurs aspects de l'application telles que la présence et le type des publicités, la présence d'un contrôle parental (et si présent, le type du contrôle parental) ou encore certaines mécaniques persuasives du jeu telles que la jouabilité répétitive. Ces informations ont été recueillies après une analyse manuelle de chaque application. L'échantillon liste 235 applications Android gratuites, 95 applications Android payantes, 240 applications Ios gratuites et 95 applications Ios payantes qui avaient été analysées entre le 8 novembre 2024 et le 28 mars 2025. L'application Sky Roller, par exemple, contient une microtransaction permettant de retirer les publicités ainsi qu'un abonnement permettant l'accès au jeu et/ou du contenu jouable supplémentaire en plus de contenir des lootboxes.

Bien que les applications étaient disponibles sur le Google Play Store, l'utilisation d'un outil tiers, APKeep[3], a été nécessaire pour l'obtention des fichiers APKs des applications de l'échantillon. En effet, le Google Play Store ne permet ni de télécharger automatiquement des applications ni de télécharger une

3 https://github.com/EFForg/apkeep

version spécifique d'une application donnée. Cet outil qui télécharge les fichiers APKs à partir d'une source tierce, APKPure[4], a permis de récupérer 85% des applications Android gratuites de l'échantillon. Bien que le tableur fourni indique l'identifiant de la majorité des applications de l'échantillon, cette information n'était pas indiquée pour 103 applications. Étant donné que l'identifiant de l'application est requis pour l'utilisation d'APKeep, une méthode semi-automatique a été utilisée pour l'obtenir à partir du nom de l'application indiqué dans le fichier.

Cette méthode utilise l'API de Google pour effectuer une recherche du nom de l'application sur le moteur de recherche Google. Le format de la requête utilisé pour la recherche était: "<nom de l'application> Google Play Store". Pour chaque résultat retourné par la requête dont l'URL pointait vers une page du Google Play Store, l'URL était analysé pour en retirer l'identifiant de l'application correspondant à la page retournée. Les identifiants ainsi récupérés étaient ensuite conservés comme étant l'identifiant potentiel de l'application présente dans l'échantillon fourni. Google-Play-Scraper, une bibliothèque permettant d'extraire des données depuis le Google Play Store, était ensuite utilisée pour récupérer le nom de l'application correspondant à chaque identifiant trouvé lors de l'étape précédente. Si ce nom correspond à une application présente dans l'échantillon fourni, l'identifiant est considéré comme correspondant à une application de l'échantillon et conservé.
Dans les cas où la recherche automatique ne permettait pas d'obtenir l'identifiant d'une application donnée, une recherche manuelle avait lieu. Les fichiers APKs des applications payantes, qui ne peuvent pas être obtenues par l'entremise des outils précédents, ont été fournis par l'équipe dirigée par Maude Bonenfant.

5.3 Décompilation

Après que les fichiers APK des applications de l'échantillon, deux méthodes distinctes ont été utilisées au cours du projet, la première utilisant les outils APKTool[5], dex2jar[6] et Quiltflower et la seconde utilisant une combinaison des outils Jadx et APKAnalyzer.

4 https://apkpure.com/

5 https://apktool.org/

6 https://github.com/pxb1988/dex2jar

5.3.1 APKTool, dex2jar, Quiltflower

5.3.1.1 APKTool

APKTool est un outil publié en 2010 conçu, entre autres, pour désassembler les fichiers APK des applications avant de décoder les fichiers ressources compilés tels que le fichier resources.arsc ou le fichier Manifest.

5.3.1.2 dex2jar

Dex2jar est un outil ouvert publié en 2011 permettant, entre autres, de convertir les fichiers DEX de l'application en fichiers .class et de désassembler les fichiers dex en smali (et vice-versa).

5.3.1.3 Quiltflower

QuiltFlower[7] est un outil visant à décompiler le code Java. Il est issu d'un fork de Fernflower et a été publié en 2022. Originellement conçu pour être utilisé dans la chaîne d'outil QuiltMC, chaîne d'outils ouverte et destinée à la modification de Minecraft, la portée de Quiltflower s'est rapidement élargie pour devenir un décompilateur Java générique. Quiltflower restera une composante de QuiltMC jusqu'en 2023 avant d'être totalement séparé et d'être renommé Vineflower.

5.3.1.4 Méthode

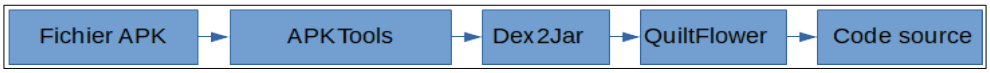

Figure 5.1: Pipeline utilisé pour la décompilation des applications avec dex2Jar

Cette première approche comprend trois étapes principales, illustrées dans la figure 5.1. La première étape visait à désassembler le fichier APK de l'application à l'aide d'APKTool. Ensuite, l'outil dex2jar était utilisé pour convertir chaque fichier DEX de l'application en fichier jar. Finalement, chaque fichier Jar était décompilé individuellement pour obtenir les fichiers Java correspondants avant que les répertoires

7 https://github.com/Vineflower/vineflower

obtenus par chaque fichier Jar soient fusionnés.

Cette méthode comporte plusieurs désavantages. En effet, en raison du nombre d'étapes plus élevé, le temps nécessaire pour décompiler chaque application avec cette approche était très élevé: plusieurs après-midis ont été nécessaires pour la décompilation des applications de l'échantillon fourni. De plus, cette méthode n'offrait pas la possibilité de déobfusquer le code.

5.3.2 Jadx/APKAnalyzer

5.3.2.1 Jadx

Jadx[8] est un outil permettant de décompiler du bytecode Dalvik en code Java. Contrairement à l'outil dex2jar, Jadx est capable de recevoir le fichier APK d'une application en entrée pour, en plus de décompiler les fichiers DEX de celle-ci, décoder le fichier AndroidManifest ainsi que certaines ressources provenant du fichier resources.arsc. Aussi, contrairement à dex2jar, Jadx inclut un déobfuscateur. Cette capacité permet donc de simplifier l'approche décrite précédemment en remplissant les rôles d'APKTool, dex2jar et Quiltflower tout en offrant une déobfuscation du code source – permettant ainsi d'obtenir potentiellement plus d'informations que la méthode précédente. Notons cependant que la déobfuscation faite par Jadx est minimale et que, conséquemment, l'information gagnée avec l'aide de cette capacité reste faible. En effet, la déobfuscation appliquée par Jadx avec les applications analysées n'a pas pour but de retrouver le nom original des variables ou des classes, mais surtout de rendre le code plus lisible.

Malheureusement, cette déobfuscation peut parfois modifier les noms de certains fichiers, classes ou méthodes correspondant à une "même" méthode sans que ce changement soit déterministe (figure 5.2). Similairement, l'obfuscation en elle-même n'est pas déterministe: une même méthode d'une même librairie installée dans deux jeux différents n'aura pas le même nom "obfusqué" durant la construction de l'application. Conséquemment, cette fonctionnalité de Jadx a éventuellement été désactivée afin de garantir une consistance entre le code source des applications de l'échantillon.

8 https://github.com/skylot/jadx

```
package com.facebook.ads;
public interface InterfaceC4116Ad {
   public interface LoadAdConfig {}
   public interface LoadConfigBuilder {
      LoadAdConfig build();
      LoadConfigBuilder withBid(String
str);
            }
   void destroy();
   String getPlacementId();
   boolean isAdInvalidated();void
loadAd();
   @Deprecated
   void setExtraHints(ExtraHints
extraHints)
}
```

```
package com.facebook.ads;
public interface InterfaceC4402Ad {
   public interface LoadAdConfig {}
   public interface LoadConfigBuilder {
      LoadAdConfig build();
      LoadConfigBuilder withBid(String
str);
            }
   void destroy();
   String getPlacementId();
   boolean isAdInvalidated();void
loadAd();
   @Deprecated
   void setExtraHints(ExtraHints
extraHints)
}
```

Figure 5.2: Code décompilé par Jadx avec déobfuscation d'une même interface de deux applications différentes. Notons que la déobfuscation retourne un nom différent.

Pour accomplir cette décompilation, Jadx effectue une multitude de transformations et de passages d'optimisation. En premier lieu, Jadx analyse le ou les fichiers DEX de l'application et consulte la liste des classes, des champs, des méthodes et des chaînes de caractères. À partir de ces informations, Jadx lit ensuite la liste d'instruction de chaque méthode et les segmente en blocs de base pour ensuite construire un graphe de flot de contrôle (CFG). Chaque nœud du graphe représente un bloc basique (soit une séquence d'instruction sans branches internes) et chaque arête représente une transition possible vers d'autres blocs (tel que les sauts conditionnels, les appels ou les retours). Ce graphe est ensuite utilisé pour effectuer les analyses requises pour la décompilation - telles que la détection de boucles et le calcul des frontières de dominance - et construire une représentation intermédiaire de type SSA (Static Single Assignement), qui est une représentation où chaque variable n'est assignée qu'une seule fois; si une variable est utilisée plusieurs fois dans l'algorithme devant être décompilé, chaque usage fait une "variante" différente de la variable. L'utilisation de cette représentation intermédiaire permet de rendre le flot de données explicite et permet ainsi d'éviter des erreurs liées à des réutilisations de registres Dalvik en plus de faciliter certaines optimisations telles que la propagation de constantes ou le code mort.

Contrairement à Java, Dalvik n'est pas structuré et repose uniquement sur des instructions de branchement pour déterminer le flot d'exécution. Jadx doit donc reconstruire le flot de contrôle à partir du CFG précédemment généré. Pour cela, le CFG est analysé dans le but de repérer des motifs connus – par exemple, un branchement arrière avec une condition sera vu comme une boucle while ou for – et les blocs de base sont récursivement regroupés dans des régions de plus haut niveau correspondant chacune à une construction Java reconnaissable. Lorsque les régions sont finalement créées, Jadx procède à quelques optimisations, telles que la simplification des branches triviales ou la factorisation de code répété, avant de générer le code Java final.

5.3.2.2 APKAnalyzer

APKanalyzer[9] est un outil développé par Google et publié en 2016 avec la version 2.2 d'Android Studio. Cet outil est conçu pour analyser les fichiers APK des applications Android et en retirer certaines informations telles que la liste des fichiers de l'application ou la liste des classes présente dans le code de l'application. Bien que cet outil possède une capacité de décompilation, celle-ci reste limitée: l'outil ne peut décompiler l'entièreté du code, mais uniquement une classe spécifique. Aussi, le code retourné lors de la décompilation d'une classe n'est pas du code Java, mais du code Smali.

5.3.2.3 Méthode

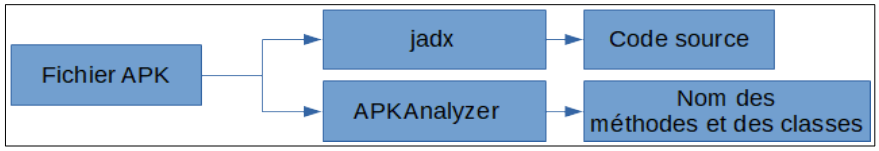

Figure 5.3: Pipeline utilisé pour la décompilation des applications avec Jadx et APKAnalyzer

La nécessité d'utiliser une combinaison des deux outils est une conséquence des limites des deux outils. En effet, bien que Jadx offre de très bonnes performances, il reste qu'une proportion non négligeable de méthodes peuvent ne pas être décompilées avec succès par cet outil. En effet, Mauthe et al. (2021) montre que dans 25% des applications testées, au moins une méthode n'a pas pu être décompilée avec

9 https://developer.android.com/tools/apkanalyzer

succès, ce qui peut arriver lorsque Jadx n'est pas en mesure restaurer la structure du code source. Lorsqu'une classe ou une méthode ne peut être décompilée par Jadx, il est possible que Jadx l'omette du code fourni lors de la décompilation.
De plus, Jadx filtre certains types de méthodes, par exemple les méthodes synthétiques (soit des classes générées par le compilateur mais n'existant pas dans le code source original), afin de réduire le bruit du code décompilé.

Bien qu'APKAnalyzer ne permette pas d'obtenir tout le code source de l'application et, conséquemment, n'est pas en mesure de fournir la totalité des informations nécessaires pour une analyse d'une application et que Jadx possède une option (--show-bad-code) permettant d'inclure le code partiellement décompilé dans la sortie de l'outil, l'approche combinant l'usage de Jadx et d'APKanalyzer a été utilisée, autant pour être en mesure d'obtenir les méthodes filtrées par Jadx que dans un but de redondance. Notons, dans la figure 5.1, que le nombre de classes et méthodes retournées par APKAnalyzer est significativement plus élevé que le nombre obtenu avec les autres outils. Cette différence n'est pas due à une mauvaise performance des outils de décompilation, mais est plutôt due au niveau d'abstraction et de granularité de chaque outil.

	dex2jar/Quiltflower	jadx	apkanalyser
Nombre de méthodes obtenues	19 847 946	23 873 956	41 320 569
Nombre de classes obtenues	4 451 557	4 682 386	6 376 442
Taille fichiers sources (octets)	16 148 917 253	15 341 018 728	n/a
Nombre fichier sources	3 797 367	3 459 998	n/a

Table 5.1: Nombre total de classes, méthodes et fichiers obtenu avec dex2jar/Quiltflower, Jadx et APKanalyzer avec l'ensemble des applications de l'échantillon.

5.4 Conclusion

Ce chapitre a présenté l'échantillon qui a été utilisé dans le cadre de ce projet. Il a ensuite détaillé comment l'API de Google a été utilisé pour récupérer automatiquement l'identifiant des applications

dudit échantillon avant de préciser comment ledit identifiant a été utilisé pour obtenir le fichier APK de ces applications. Le chapitre a ensuite présenté les deux approches ayant été utilisées pour décompiler les applications obtenues, l'une utilisant APKTools, Dex2jar et Quilflower et l'autre utilisant Jadx et APKAnalyzer, tout en présentant les outils utilisés dans chacune de ces approches. Finalement, le chapitre a expliqué pourquoi la seconde approche, soit l'utilisation de Jadx et APKAnalyzer, a été retenue.

CHAPITRE 6

ANTI-FONCTIONNALITÉES, LEURS DÉTECTIONS ET AUTOMATISATION

6.1 Anti-fonctionalitées

Les anti-fonctionnalités recherchées par l'outil sont au nombre de six et se divisent en trois catégories distinctes: les microtransactions (qui sont composées des microtransactions consommables, des microtransactions non consommables et des abonnements), les publicités (qui sont composées des publicités récompensées et des publicités interstitielles) et les mécaniques persuasives (qui contient uniquement les notifications poussées). Le choix de ces anti-fonctionnalités se base principalement sur la "popularité" desdites anti-fonctionnalités dans les applications mobiles ainsi que sur l'impact important desdites anti-fonctionnalités. En effet, les microtransactions peuvent mener à des pertes financières importantes, les librairies de publicités possèdent les même permissions que l'application en elle-même et les notifications poussées sont un mécanisme visant à ramener l'utilisateur vers l'application lorsque celui ci ne l'utilise pas.

6.1.1 Publicités

Les publicités représentent une stratégie de monétisation très souvent utilisée dans les applications mobiles. L'affichage des publicités est aussi, pour les applications gratuites, la principale source de revenus par une marge significative avec, selon l'entreprise data.ai, 67% des revenus de ces applications provenant des publicités contre 33% provenant des microtransactions (Data.ai, 2023).

Bien qu'il soit possible d'intégrer nativement des publicités dans une application, ce n'est généralement pas l'approche utilisée. Les applications mobiles intègrent plutôt un ou plusieurs kits de développement (tel que Admob[10], AppLovin[11] ou Pangle Global[12]), chacun permettant d'interagir avec un ou plusieurs réseaux de publicités, permettant ainsi de faciliter grandement l'intégration des publicités, autant pour le

10 https://admob.google.com/home/

11 https://www.applovin.com/

12 https://www.pangleglobal.com/

développeur de l'application que pour l'acheteur de publicité, en permettant à ceux-ci de déléguer, entre autres, la gestion des enchères et le ciblage des utilisateurs au réseau de publicité en question. Ainsi, un développeur souhaitant intégrer de la publicité dans une application n'a pas besoin de trouver et de négocier avec des acheteurs potentiels et lesdits acheteurs n'ont pas besoin de trouver et négocier avec les développeurs d'application correspondant au marché visé par la publicité tout comme le développeur n'a pas besoin d'implémenter le ciblage publicitaire dans l'application. Plutôt, le développeur implémente l'apparition d'une publicité du type choisi par celui-ci et le réseau de publicité choisi par le développeur prend en charge le choix de la publicité ainsi que l'affichage et, si nécessaire, la collecte de certaines informations telle que les métriques d'interactions avec la publicité. Concrètement, le développeur vend un espace de publicité en spécifiant le type de publicité – par exemple, une publicité interstitielle - à un réseau de publicité qui achète ensuite cet espace pour y afficher une publicité.

nom	Types de publicité supportée	% d'applications
Google Ads AdMob (Google)	Interstitiel, Récompensé	88.68
Facebook Audience Network (Facebook)	Interstitiel	24.49
IAB OM Open Measurement OMSDK (IAB Technology Laboratory)	Interstitiel, Récompensé	21.94
Unity Ads (Unity)	Interstitiel, récompensées	19.43
AppLovin MAX (AppLovin)	Interstitiel, récompensées	17.2
IronSource (IronSource)	Interstitiel, récompensées	10.99
Vungle (Vungle)	Interstitiel, récompensées	9.00
Pangle ByteDance TikTok UnionAD (Pangle)	Interstitiel, récompensées	8.63
Mintegral mbbanner (Mintegral)	Interstitiel, récompensées	8.11
Start.io Starapp (Start.io)	Interstitiel, récompensées	6.3

Table 6.1: Kits de développement les plus populaires, incluant la compagnie offrant le kit, le type de publicité supporté parmi celles recherchées dans ce projet et le pourcentage d'applications intégrant le kit. Une application mobile peut intégrer plusieurs bibliothèques de publicité.

Comme le montre le tableau 6.1, le kit de développement le plus populaire en date de 2025 est de loin le kit fourni par Google (Google Ads AdMob), étant implémenté dans 88.86% des applications possédant un kit de développement de publicité, suivi par Facebook Audience Network (provenant de Meta) avec 24.49% des applications et IAB OM Open Measurement OMSDK (provenant de IAB Technology Laboratory) avec 21.94%. (Tila, 2025)

6.1.2 Microtransactions

L'implémentation des microtransactions est semblable à l'implémentation des publicités dans les applications Android. En effet, tout comme les publicités, le développeur d'une application a la possibilité d'implémenter les microtransactions nativement dans le code, d'utiliser un kit de développement logiciel offrant les fonctionnalités désirées ou encore de l'implémenter nativement mais de passer par un système de payement tiers tel que PayPal. Cependant, il y a, en pratique, une obligation d'utiliser l'un des quelques kits disponibles. Cette obligation est causée non pas de la complexité d'implémentation des microtransactions, mais des politiques des boutiques d'applications.

En effet, contrairement aux publicités, qui n'ont pas de restriction quant à leurs provenances, certaines boutiques d'applications imposent des restrictions au niveau des microtransactions. Le Google Play Store, par exemple, oblige les applications de passer par l'entremise de celui-ci pour procéder à la transaction lors d'un achat – transaction dont Google récupérera une commission variant entre 15 et 30% selon le type de transaction et le revenu annuel des applications développeur.

Boutique d'application	Nombre d'utilisateurs	Politique sur les méthodes de transaction des microtransactions
Amazon Appstore	Fermé le 20 août 2025	Transaction effectuée via Amazon
Google Play Store	> 2.5 milliards	Transaction effectuée via Google
Samsung Galaxy Store	> 400 millions (2019)	Aucune
Huawai App Galery	> 580 millions (fin 2023)	Transaction effectuée via Huawai

Table 6.2: Boutiques d'application Android populaire, ainsi que les politiques de publications au sujet de la transaction dans les applications. (Samsung, 2019; Tila, 2024; Winder, 2024)

6.1.3 Notifications poussées

Les notifications poussées sont une fenêtre apparaissant lorsque l'application n'est pas en cours d'utilisation et visant à informer l'utilisateur des événements prenant place durant leur absence – par exemple, dans une application pour un réseau social, une notification poussée pourrait avertir l'utilisateur de la réception d'un nouveau message. Les notifications poussées, dans les systèmes Android, peuvent être subdivisées en deux catégories: les notifications standard et les notifications interactives.

Les notifications standard sont les notifications les plus simples. Elles consistent simplement en une vignette optionnelle accompagnée de texte. Lorsque l'utilisateur clique sur la notification, il est amené à l'application responsable de la notification. Les notifications interactives possèdent, en plus de la vignette optionnelle et du texte, des boutons permettant à l'utilisateur d'interagir avec l'application responsable de la notification sans avoir à se rendre à ladite application.

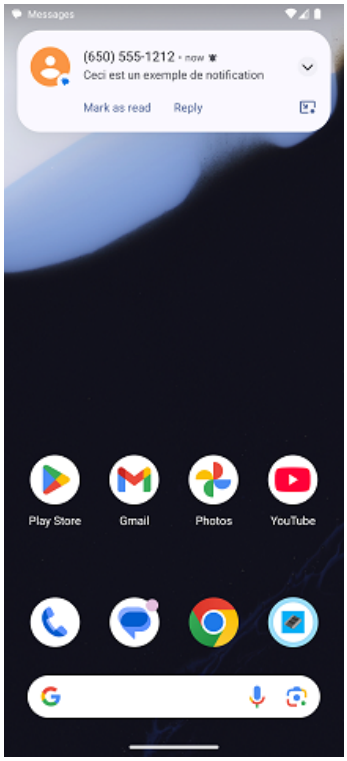

Figure 6.1: Exemple de notification poussée

Pour être en mesure d'afficher une notification poussée, que celle-ci soit une notification standard ou interactive, l'application doit posséder la permission "POST_NOTIFICATIONS", qui est une permission de type *runtime* (section 4.4.2). Cette particularité est, dans le cadre des travaux de ce mémoire, exploité pour obtenir une plus grande précision pour la détection des notifications poussées.

6.2 Détection des anti-fonctionnalités

L'approche utilisée pour la détection consiste en une analyse statique du code source des applications mobiles. Cette analyse consiste en une recherche de motifs, qui consistent en des expressions régulières, dans les fichiers de l'application: un motif est associé à chaque anti-fonctionnalité et est recherché dans des fichiers spécifiques de l'application. Bien que la plupart de ces motifs soient recherchés dans la totalité du code source de l'application, d'autres motifs tels que les notifications possèdent certaines restrictions au niveau des fichiers analysés. Si un motif est présent dans les fichiers pertinents, alors il est présumé que l'aspect correspondant est présent dans l'application. Cette approche a été choisie sur l'hypothèse que les implémentations des diverses anti-fonctionnalités sont relativement standardisées en raison des restrictions et pratiques courantes retrouvées dans le domaine des applications Android: les publicités, par exemple, sont généralement implémentées à travers des bibliothèques de publicités et devraient conséquemment utiliser les APIs et implémentations desdites bibliothèques.

6.2.1 Sélection des motifs

La sélection des motifs utilisés pour la détection des anti-fonctionnalités est effectuée selon deux méthodes de sélection. Une étude de la littérature grise disponible sur Internet et une analyse du score de l'information mutuelle de certains motifs dans le code source des applications.

6.2.1.1 Ressources didactiques

La première approche consiste en un tour d'horizon des ressources didactiques tel que les tutoriels ou forums d'aide disponible sur Internet tels que la documentation de certaines libraires ou StackOverflow. Cette approche est basée sur l'hypothèse que l'implémentation des anti-fonctionnalités recherchées est populaire et, conséquemment, un sujet qui sera extensivement couvert par de telles ressources en plus

d'être relativement simple et, étant donné les pratiques courantes décrites précédemment, relativement standardisée. Bien que subjective, une telle approche permet un tour d'horizon des approches couramment utilisées en plus de vérifier s'il existe des motifs souvent utilisés, que ce soit par des implémentations similaires à travers les bibliothèques implémentant les anti-fonctionnalités, par un nombre restreint de possibilités ou par des implémentations plus faciles.

6.2.1.2 Information mutuelle

La seconde approche consiste en une analyse d'information mutuelle de certains motifs du code source. L'information mutuelle est une mesure entre deux variables indiquant la dépendance statique desdites variables. En d'autres mots, l'information mutuelle indique dans quelle mesure la connaissance d'une variable A réduit l'incertitude associée à la variable B; dans l'application utilisée ci-dessous, elle mesure dans quelle mesure la présence d'un motif (variable A) réduit l'incertitude quant à la présence de l'anti-fonctionnalité associée. Dans l'application actuelle, plus le score de l'information mutuelle d'un motif est élevée, plus on est certain que celui-ci indique la présence de l'anti-fonctionnalité.

L'application courante étant un cadre discret, la formule utilisée est:

$$\sum_{y \in Y} \sum_{x \in X} P_{(x,y)}(x,y) \log \frac{\left(P_{(x,y)}(x,y)\right)}{P_X(X) P_y(Y)}$$

Avec:

- P(x,y) étant la probabilité jointe de X=x et Y=y
- P(x) et P(y) étant les probabilités marginales

La valeur maximale possible du score d'information mutuelle d'un motif donné correspond à l'entropie de l'anti-fonctionnalité en question, soit le degré d'incertitude associé à sa présence ou à son absence. Dans le but de compenser cet aspect de l'information mutuelle, le score de chaque motif a été normalisé sur l'entropie de l'anti-fonctionnalité associée audit motif, permettant ainsi de vérifier si un score faible

d'un motif est dû à une mauvaise discrimination ou s'il est dû à une entropie faible de l'anti-fonctionnalité. Étant donné le besoin de vérité de référence pour être capable de procéder à une telle analyse, celle-ci a été effectuée sur les applications de l'échantillon décrit au chapitre 5.

Une limite significative de l'analyse d'information mutuelle dans l'objectif d'identifier les meilleurs motifs est que cette équation n'est pas un indicateur de causalité, mais de corrélation. Conséquemment, un motif peut posséder un haut score d'information mutuelle sans que ledit motif soit réellement un indicateur robuste de la présence de l'anti-fonctionnalité recherchée. Par exemple, avec l'échantillon fourni, un des motifs possédant le plus haut score d'information mutuelle pour les publicités interstitielles était "WEB_AUTHENTICATION" avec un score de 0.302 et est utilisé non pas pour l'affichage de publicité, mais pour l'affichage de contenu Web. Malgré ce score d'information mutuelle, ce motif, lorsqu'utilisé par l'outil pour rechercher les publicités récompensées, offre une précision de 0.667 et un rappel de 0.845. Cette disparité n'est pas lié au fait que le "WEB_AUTHENTICATION" soit utilisé pour implémenter les publicités interstitielles, mais plutôt parce qu'il est corrélé aux publicités, incluant les publicités interstitielles.

Le calcul de l'information mutuelle des motifs potentiels a été réalisé à l'aide d'un programme Python utilisant la bibliothèque sklearn[13]. Comme mentionné précédemment, afin de permettre une visualisation de la "performance" des motifs potentiels, la valeur de l'information mutuelle de chaque motif a été normalisée avec l'entropie de l'anti-fonctionnalité concernée. Ainsi, un résultat d'une valeur de 1 indique un motif dont la présence dans le code source d'une application permet de prédire parfaitement la présence de l'anti-fonctionnalité associée alors qu'un résultat de 0 indiquerait un motif dont la présence ne donne aucune information sur la présence de ladite anti-fonctionnalité. Aussi, afin d'accélérer l'analyse de l'information mutuelle, le score d'information mutuelle d'un motif donné n'était calculé que si ce motif était présent dans au moins 20 applications différentes.

Cette analyse a été effectuée après plusieurs types des prétraitements. Le premier visait uniquement à optimiser les performances du code effectuant l'analyse et consistait à fusionner le code source de l'application en un seul fichier. Cette fusion permettait d'éliminer le besoin d'ouvrir et de fermer tous les fichiers du code source lors de chaque analyse; ces ouvertures et fermetures des fichiers, en raison du

13 https://scikit-learn.org

haut nombre de fichiers dans les applications analysées, ralentissaient significativement l'analyse. Ce même prétraitement a aussi été effectué avec certaines restrictions au niveau des fichiers devant être fusionnés, tels que d'exclure les fichiers présents ne se trouvant pas dans le répertoire "<apk>/com" du code source de l'application. La raison de cette exclusion est que ce répertoire contient les fonctionnalités natives d'Android. Ces fonctionnalités natives, n'étant pas du code spécifique à l'application étudiée, ont le potentiel d'ajouter du bruit inutile dans l'analyse.

Il y avait aussi d'autres prétraitements utilisés pour l'analyse, chacun visant à analyser un aspect du code différent. Ces prétraitements consistaient en l'extraction des méthodes ainsi que des classes du code source des applications pour, respectivement, calculer le score d'information mutuelle de chaque méthode ou classes présentes dans le code source.
De la même manière que le prétraitement original, les déclarations et corps extraits lors de ces seconds prétraitements ont chacun été fusionnés dans des fichiers distincts dans l'objectif de rendre l'analyse plus rapide. Dans ce cas-ci, par contre, l'objectif n'était pas de réduire le nombre d'ouvertures et de fermetures de fichiers, mais d'éviter de faire l'extraction des classes et des méthodes lors de chaque analyse.

Les classes et les méthodes ont été extraites à l'aide d'un programme développé en Python utilisant des expressions régulières pour extraire la déclaration des méthodes et des classes dans le code et en retirer le nom, le type de retour, les paramètres, les annotations ainsi que les exceptions.

L'aide fournie par cette analyse a grandement varié selon l'anti-fonctionnalité pour laquelle ladite analyse a été effectuée. En effet, bien que certains motifs retournés correspondaient à une méthode/classe causant l'anti-fonctionnalité, la grande majorité des motifs possédait uniquement un lien de corrélation avec l'anti-fonctionnalité.

6.3 Conception de l'outil de détection

L'outil, nommé ZacDroid[14], a deux fonctions principales. Il y a, bien sûr, la détection des anti-fonctionnalités dans une application Android, mais il permet aussi de mesurer le rappel et la précision de

14 https://github.com/HeliumApple/ZacDroid

plusieurs motifs utilisés pour la détection d'une même anti-fonctionnalité – fonction très utile pour vérifier si des motifs potentiels sont viables pour la détection de ladite anti-fonctionnalité. ZacDroid peut être utilisé par le biais d'une interface graphique ainsi que par le biais d'une ligne de commande et peut retourner les résultats autant avec une visualisation graphique des résultats que par un fichier en format JSON.

En entrée, ZacDroid prend les dossiers de chaque application préalablement décompilée selon la méthode décrite précédemment et, s'il est utilisé dans le but de valider des motifs potentiels, il prend aussi en entrée un fichier Excel indiquant, pour chaque application de l'échantillon utilisé, les anti-fonctionnalités présentes dans ladite application.
Si le mode graphique est utilisé, ZacDroid peut comparer les résultats d'une analyse avec les informations indiquées dans le fichier Excel et indiquer, pour chaque anti-fonctionnalité étudiée, si le résultat est erroné (i.e.: le résultat est un faux négatif ou un faux positif) ou si le résultat est véridique (i.e.: le résultat est un vrai positif ou un vrai négatif). Lesdits résultats peuvent autant être obtenus par une nouvelle analyse que par l'importation d'un fichier JSON indiquant les résultats d'une analyse précédente.

Comme mentionné précédemment, l'approche utilisée est une recherche de motifs dans des fichiers de l'application. Cette recherche est implémentée sous la forme d'une recherche REGEX. Pour chaque application, ZacDroid parcourt les fichiers java du code source de l'application et, pour chaque anti-fonctionnalité, recherche le motif associé à ladite anti-fonctionnalité dans le fichier en question. Si le motif est présent, ZacDroid fait l'hypothèse que l'anti-fonctionnalité associée est présente dans l'application et, afin d'améliorer les performances, ne cherchera plus cette anti-fonctionnalité dans les prochains fichiers. Si le motif n'a pas été trouvé dans le code source de l'application, alors ZacDroid suppose que l'anti-fonctionnalité associée n'est pas présente dans l'application. Pour des fins d'optimisation, le motif d'une anti-fonctionnalité donné est uniquement recherché si ladite anti-fonctionnalité n'a pas été détectée dans un autre fichier de l'application.

Une fois les applications analysées, l'étape suivante dépendra du mode utilisé. Si le mode graphique est utilisé, l'outil affiche un tableau avec, comme colonnes, le nom de l'application et les anti-fonctionnalités recherchées dans chaque catégorie. Chaque rangée du tableau indique le nom de l'application ainsi que, pour chaque anti-fonctionnalité, si celle-ci a été détectée dans le code de l'application en question. Si le

mode graphique n'est pas utilisé, l'outil exportera les résultats sous la forme d'un fichier JSON listant les applications et, pour chaque anti-fonctionnalité, si celle-ci est présente ou non.

```
Motif_publicité_récompensé="earnedRewarded"
Motif_publicité_interstitiel="InterstitialAd"

Pour tout les fichiers de l'application:

        si Motif_publicité_récompensé est présent:
                L'application contient des publicités récompensées

        si Motif_publicité_interstiel est présent:
                L'application contient des publicités interstitielles
```

Figure 6.2: Pseudocode démontrant l'approche utilisée pour la recherche d'anti-fonctionnalité.

Si un fichier Excel est fourni et que le mode graphique est utilisé, pour chaque application listée dans le fichier Excel, l'outil indiquera la véracité du résultat avec un système de couleur: une case rouge indique un résultat erroné (i.e: un faux positif ou un faux négatif) et une case verte indique un résultat véridique (i.e: un vrai positif ou un vrai négatif). L'outil indique aussi, pour chaque anti-fonctionnalité, le nombre de faux négatifs, le nombre de faux positifs, le nombre de vrais positifs, le nombre de vrais négatif, le rappel ainsi que la précision.

En plus des anti-fonctionnalités, l'outil peut aussi être utilisé pour mesurer le rappel et la précision d'un nombre important de motifs pour une même anti-fonctionnalité durant une même analyse. En plus de permettre une comparaison plus aisée, en affichant les résultats de chaque motif dans la même fenêtre, la principale fonctionnalité est de réduire le temps requis pour la validation des motifs candidats. Pour cela, il y a trois groupes de huit motifs pouvant être testés, chaque groupe correspondant à une anti-fonctionnalité spécifique.

L'outil peut aussi, optionnellement, offrir des informations additionnelles concernant la détection des motifs. Plus spécifiquement, il indique les fichiers dans lesquels le motif a été détecté ainsi qu'un nombre arbitraire de lignes de code précédant et suivant la ligne contenant le motif en question. La granularité de la recherche est légèrement différente lorsque cette fonctionnalité est utilisée. En effet, alors que ZacDroid applique normalement la recherche REGEX sur l'ensemble de chaque fichier, le besoin

d'identifier les lignes précédant et suivant la ligne contenant le motif recherché impose que l'outil applique cette même recherche REGEX ligne par ligne.

Ce changement, en plus du besoin de rechercher le motif dans tous les fichiers de l'application même si l'anti-fonctionnalité a déjà été détectée précédemment, cause une augmentation significative du temps requis pour faire l'analyse des applications. De ce fait, l'outil permet aussi de fournir uniquement les fichiers dans lesquels se trouvent les motifs trouvés sans indiquer les lignes entourant la ligne contenant ce motif. Ce compromis permet à ZacDroid de ne pas avoir à appliquer la recherche REGEX ligne par ligne et réduit ainsi le temps requis pour compléter l'analyse.

	Ensemble du fichier	Lignes par ligne
Uniquement négatif	1850 secondes	3200 secondes
Uniquement positif	310 secondes	330 secondes

Table 6.3: Temps requis par l'outil pour l'analyse de l'échantillon selon que la recherche REGEX est appliquée sur l'ensemble de chaque fichier ou si elle est appliquée ligne par ligne. Un groupe de huit motifs ont été recherchés dans le cadre de la mesure.

Lorsque cette option est utilisée, l'outil, lorsqu'une anti-fonctionnalité est détectée, compare le résultat avec l'échantillon pour déterminer s'il s'agit d'un vrai positif ou d'un faux positif et écrit les informations désirées (ou bien les lignes entourant la ligne contenant le motif associé ainsi que le fichier concerné ou bien uniquement le fichier concerné) dans des fichiers distincts correspondants à l'anti-fonctionnalité concernée: un fichier indiquant les fichiers ou lignes contenant le motif menant au faux positif et un fichier indiquant les fichiers ou lignes contenant le motif menant au vrai positif. Notons que, dans ce dernier cas, l'outil n'indique pas si le motif menant au vrai positif est "valide", mais uniquement que ce motif est présent dans une application contenant l'anti-fonctionnalité recherchée avec ce motif. Conséquemment, les lignes et fichiers indiqués dans les vrais positifs ne sont pas obligatoirement un bon indicateur de la présence d'une anti-fonctionnalité dans une application.

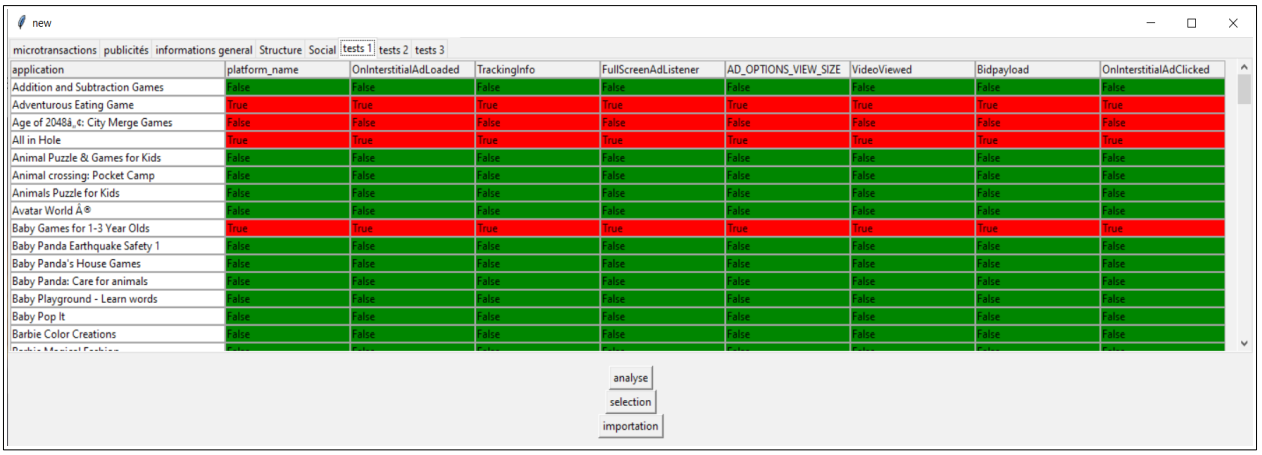

Figure 6.3: Interface de l'outil. Le tableau indique, pour chaque motif, s'il détecte l'anti-fonctionnalité et si le résultat est correct

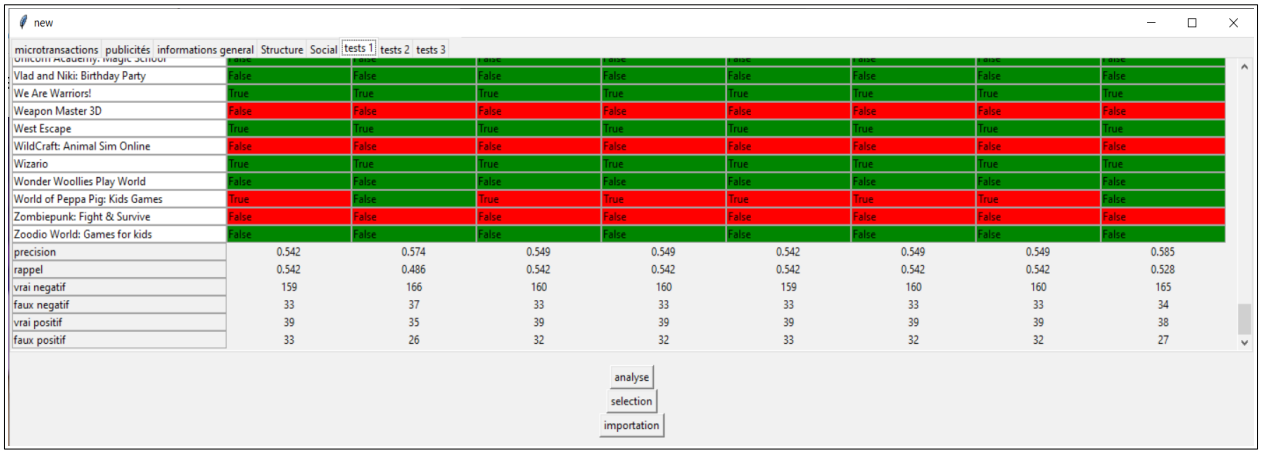

Figure 6.4: interface de l'outil lors du même test que l'image précédente. Les statistiques, telles que la précision ou le rappel, sont indiquées en bas de la fenêtre.

CHAPITRE 7
RÉSULTATS

Les deux objectifs étaient de vérifier s'il était possible d'identifier des motifs couramment utilisés pour l'implémentation de certaines anti-fonctionnalités dans les applications destinées aux enfants et, si oui, d'identifier lesquels de ces motifs permettait d'obtenir, tout en gardant une précision et un rappel satisfaisant, le plus haut rappel, la plus haute précision et le score F1 le plus élevé.

7.1 Publicités

La méthode proposée offre des résultats satisfaisants pour la détection et la classification des publicités dans les applications mobiles destinées aux enfants. En effet, les figures 7.1 et 7.3, montrent qu'une telle approche permet d'obtenir autant un score F1 élevée, mais aussi des motifs offrant des précisions et des rappels différents, permettant ainsi le choix de prioriser la précision ou le rappel selon les besoins.

7.1.1 Publicités interstitielles

Au niveau des publicités interstitielles, on remarque que le motif permettant d'obtenir le score F1 le plus élevé est le motif "platform_name" – utilisé par Admob pour paramétrer les publicités - qui offre une précision de 0.690 et un rappel de 0.816 pour un score F1 de 0.748. On remarque aussi que, parmi les motifs offrant un rappel supérieur à 0.7, la plage de précision est de 0.120, allant de 0.602 (pour "onInterstitialAdLoaded") à 0.722 (pour "platform_name"). Les motifs dans cette plage donnent un rappel allant de 0.732 ("TrackingInfo") à 0.873 ("onInterstitialAdLoaded").
Parmi les motifs offrant une précision supérieure à 0.7, on remarque que le rappel varie de 0.634 (pour "FullScreenAdListener") à 0.873 (pour "OnInterstitialAdLoaded") et dont les motifs offrent une précision allant de 0.701 (avec "AD_OPTIONS_VIEW_SIZE") à 0.725 (avec "FullScreenAdListener"). Si l'on considère uniquement les motifs avec un score F1 supérieure à 0.7, on remarque que la plage de rappel devient 0.183, avec des motifs dont la précision est entre 0.817 et 1.00.

Bien que bonne, la détection des publicités interstitielles par la recherche des motifs du tableau 7.1 peut ne pas être stable au long terme. En effet, les motifs utilisés correspondent moins à des motifs “génériques”, mais majoritairement à des variables, méthodes et classes associées à des kits de développement fournis par des réseaux de publicités spécifiques.

motif	nature	précision	rappel	F1
platform_name (Admob)	Variable	0.690	0.816	0.748
OnInterstitialAdLoaded (Unity)	Méthode	0.602	0.873	0.713
TrackingInfo (Vungle)	Classe	0.722	0.732	0.727
FullScreenAdListener (BidMachine)	Interface	0.725	0.634	0.677
AD_OPTIONS_VIEW_SIZE (Vungle)	Variable	0.701	0.662	0.681
VideoViewed (Vungle)	Variable	0.691	0.662	0.676
Bidpayload (BidMachine)	Classe	0.688	0.620	0.652
OnInterstitialAdClicked (AppLovin)	Méthode	0.612	0.845	0.710

Table 7.1: Motifs utilisés pour la détection des publicités interstitielles et leurs performances.

Conséquemment, la précision ainsi que le rappel obtenu avec chaque motif dépendront de la popularité respective de chaque réseau de publicités. Par exemple, le motif “Bidpayload” est utilisé par le réseau de publicité Vungle et ne correspond pas à un motif “générique”: il est donc possible que, dans l’hypothèse que la popularité de Vungle baisserait, la qualité des résultats obtenus se dégrade et que l’outil ait besoin d’être mis à jour pour être en mesure de fournir des résultats de qualité satisfaisante.

Dans le but d’augmenter le rappel, la détection est aussi effectuée en cherchant non pas un motif unique mais une disjonction sur un groupe de motifs présélectionnés - avec la présence de la publicité supposée si un des motifs du groupe est présent dans le code. Ce groupe est composé des deux motifs possédant la plus haute précision pour les publicités interstitielles (soit “FullScreenAdListener” et “TrackingInfo”): si l’on tente d’ajouter le troisième motif possédant la plus haute précision (soit “AD_OPTIONS_VIEW_SIZE”), on remarque que le rappel reste le même alors que, sans surprise, la précision baisse; contrairement à

l'ajout du deuxième motif avec la plus haute précision qui, bien que faisant baisser la précision, augmente le rappel suffisamment pour que le score F1 augmente significativement (passant de 0.667 à 0.735). L'utilisation d'une combinaison de plusieurs motifs est donc une alternative envisageable à l'utilisation d'un motif unique selon l'importance accordée à la précision et au rappel.

Motifs ajoutés au groupe	Précision	Rappel	F1
FullScreenAdListener	0.725	0.634	0.677
TrackingInfoTrackingInfo	0.720	0.750	0.735
AD_OPTIONS_VIEW_SIZE	0.701	0.750	0.726
VideoViewed	0.692	0.750	0.721

Table 7.2: Performances du groupe de motifs après l'ajout de chaque motif, en partant du haut, pour les publicités interstitielles.

7.1.2 Publicité récompensée

Au niveau des publicités récompensées, on remarque que le motif offrant le score F1 le plus élevé est "showRewardedAd" (offrant une précision de 0.859 et un rappel de 0.847 pour un score F1 de 0.853).

On remarque aussi que, parmi les motifs offrant un rappel supérieur à 0.7, la plage de précision est de 0.227, allant de 0.632 (pour "OnUserEarnedReward") à 0.859 (pour "showRewardedAd"). Les motifs dans cette plage donnent un rappel allant de 0.847 ("showRewardedAd") à 1.000 (pour "OnUserEarnedReward" et "new PAGRewardedRequest"); tous les motifs offrant un rappel supérieur à 0.7 ont aussi un score F1 supérieur à 0.7. On remarque aussi que, similairement aux publicités interstitielles, il y a peu de motifs offrant une précision supérieure à 0.7: "showRewardedAd" (avec une précision de 0.859 et un rappel de 0.847), "new RewardedAd" (avec une précision de 0.889 et un rappel de 0.670) et "new RewardedAd " (avec une précision de 0.726 et un rappel de 0.847) sont les seuls motifs remplissant ce critère parmi les motifs sélectionnés, offrant, pour la précision et pour le rappel, une plage de 0.163.

Finalement, parmi les motifs offrant un score F1 supérieur à 0.7. La plage de précision est de 0.257, allant de 0.632 (pour "OnUserEarnedReward") à 0.889 (pour "RewardedAd.load()") et la plage de rappel est de

0.330 allant de 0.670 (pour “RewardedAd.load()”) à 1.000 (pour “OnUserEarnedReward” et “RewardedAd.load()”).

Motif	nature	précision	rappel	f1
onUserEarnedReward (google)	Méthode	0.632	1.000	0.816
RewardedVideoAdListener (Facebook)	Classe	0.633	0.972	0.803
DisplayRewardedVideo (Unity)	Méthode	n/a	n/a	n/a
createRewardedAd (AppLovin)	Méthode	0.4	0.056	0.228
showRewardedAd (IronSource)	Méthode	0.859	0.847	0.853
new RewardedAd (Vungle)	Création d’objet	0.726	0.958	0.842
new PAGRewardedRequest (Pangle)	Création d’objet	0.643	1.000	0.822
RewardedAd.load() (Mintegral)	Méthode	0.889	0.670	0.780

Table 7.3: Motifs utilisés pour la détection des publicités récompensées et leurs performances. L’expression régulière utilisée pour la recherche de chaque motif suit le format suivant: “[^a-zA-Z0-9]{motif utilisé}[^a-zA-Z0-9]“. La casse n’est pas respectée.

Similairement aux motifs employés pour la détection des publicités interstitielles, les motifs utilisés pour détecter les publicités récompensées se basent sur les APIs des kits de développement fournis par les réseaux de publicités et non pas à des motifs “génériques”. Conséquemment, tout comme pour la détection des publicités interstitielles, les performances de détection pourraient se détériorer si un des réseaux de publicité donnant l’API utilisé devenait moins utilisé dans les applications mobiles. Ainsi, tout comme pour les publicités interstitielles, l’outil pourrait nécessiter des mises à jour ainsi que des tests réguliers afin d’assurer des bonnes performances au long terme.

Similairement aux publicités interstitielles, la détection peut aussi être effectuée en cherchant non pas un motif spécifique mais avec une disjonction sur un groupe de motifs présélectionnés.

Tout comme pour les publicités interstitielles, ce groupe est composé des deux motifs possédant la plus haute précision pour les publicités interstitielles (soit “RewardedAd.load()”, “showRewardedAd”). Cependant, contrairement aux publicités interstitielles, on remarque que cet ajout n’est pas aussi avantageux que dans le cas des publicités interstitielles. En effet, alors que la précision ne baisse que de 0.05 pour les publicités interstitielles, dans le cas des publicités récompensées cette baisse est de 0.252. Notons, cependant, que le rappel augmente très significativement, passant de 0.670 à 1.000 (soit une augmentation de 0.330) et que le score F1 augmente malgré la baisse significative de la précision : cette sélection est donc surtout utile dans les cas où garantir la détection des publicités récompensées justifie le « coût » d’une précision plus faible. Il est important de noter, par contre, que le motif « new RewardedAd » offre un rappel très proche de celui offert par le « groupe » de motif (offrant un rappel de 0.958) mais offre néanmoins une précision plus élevée (soit 0.726).

Motif ajouté	précision	rappel	F1
RewardedAd.load()	0.889	0.67	0.780
showRewardedAd	0.637	1.000	0.819

Table 7.4: Performances du groupe de mots après l'ajout de chaque motif, en partant du haut, pour les publicités interstitielles

Pour les deux types de publicités, il n’y a pas d’application avec laquelle tous les motifs résultent en un faux négatif. Par exemple, dans le cas des publicités récompensées et de l’application *Octonauts and the Giant Squid* de l’échantillon, les motifs (« showRewardedAd », « new PAG... » et « RewardedAd ») indiquent correctement la présence de la publicité récompensée alors que les cinq autres motifs résultent en un faux négatif, indiquant ainsi que les motifs choisis peuvent être « spécifiques » à des bibliothèques de réseaux de publicité et que les faux négatifs de chaque motif peuvent être dus au fait qu’ils correspondent à une bibliothèque qui n’est pas utilisée par l’application.

Quant aux faux positifs, ceux-ci étaient majoritairement dus à leur présence dans les fichiers sources des bibliothèques de publicités, indiquant ainsi que certaines applications intègrent des bibliothèques de publicités sans en faire l’usage. Étant donné que R8 obfusque entre autres le nom des méthodes et des classes de l’application durant la compilation, une analyse plus approfondie serait nécessaire pour s’assurer que la publicité est bel et bien utilisée dans l’application – par exemple avec une analyse d’accessibilité pour vérifier qu’il existe un chemin dans le code menant à l’utilisation de la bibliothèque

de publicité ou bien en une déobfuscation détaillée du code source pour être en mesure de rechercher les motifs dans le code de l'application avec une plus grande fiabilité.

7.2 Mécaniques persuasives

7.3 Notifications poussées

motif	nature	précision	rappel	F1
NotificationCompat.Builder	Méthode	0.438	0.457	0.448
checkNotificationStatus	Méthode	0.458	0.717	0.588
showNotification	Méthode	0.459	0.739	0.599
checkIfPendingNotificationIsRegistered	Méthode	0.458	0.717	0.588
enqueueNotification	Méthode	0.508	0.717	0.613
onMessageReceived	Méthode	0.368	0.696	0.532

Table 7.5: Motifs utilisés pour la détection des notifications poussées et leurs performances sans vérification des permissions. L'expression régulière utilisée suit le format suivant: "[^a-zA-Z0-9] {motif }[^a-zA-Z0-9]"

La méthode utilisée pour la détection des notifications diffère de la méthode utilisée pour les publicités et les microtransactions. En effet, contrairement à ces deux catégories d'anti-fonctionnalités, les notifications poussées nécessitent la permission "android.permission.POST_NOTIFICATIONS" pour pouvoir être affichées à l'utilisateur. Cette permission *runtime* est, à l'instar des autres permissions, déclarées dans le fichier Manifest de l'application. L'outil, dans l'objectif de réduire le nombre de faux positifs, profite de cette particularité et suppose que l'application contient des notifications si le motif

recherché est présent ET que la permission requise pour l'affichage des notifications est déclarée dans le fichier Manifest de l'application.

motif	nature	précision	rappel	F1
NotificationCompat.Builder	Méthode	0.656	0.457	0.557
checkNotificationStatus	Méthode	0.472	0.739	0.606
showNotification	Méthode	0.508	0.647	0.578
checkIfPendingNotificationIsRegistered	Méthode	0.465	0.717	0.591
enqueueNotification	Méthode	0.508	0.717	0.613
onMessageReceived	Méthode	0.447	0.674	0.561

Table 7.6: Motifs utilisés pour la détection des notifications poussées et leurs performances avec vérification des permissions. L'expression régulière utilisée suit le format suivant: "[^a-zA-Z0-9] {motif }[^a-zA-Z0-9]"

De plus, contrairement aux publicités, les motifs ne sont pas cherchés dans l'entièreté du code source de l'application. En effet, l'affichage de notification étant une fonctionnalité native d'Android, beaucoup de motifs utilisés pour détecter les notifications sont aussi présents dans les fichiers de bibliothèques de l'application, créant ainsi de faux positifs. Conséquemment, afin d'éviter ces fichiers, l'outil considère uniquement les fichiers contenus dans le dossier "<paquet>/sources/com". Similairement, afin d'éviter les faux positifs dus aux bibliothèques de publicités, les fichiers dont le chemin contient les termes suivants sont exclus de la recherche: "ads", "pangle", "vungle", "ironsource", "applovin" et "google". Ces termes correspondent à divers dossiers présents dans certaines bibliothèques de publicités. Contrairement aux publicités, cette approche ne donne pas des résultats satisfaisants pour la détection des notifications poussées. En effet, bien qu'il soit possible d'obtenir un rappel satisfaisant avec certains

motifs (par exemple, “checkNotificationStatus” donnant un rappel de 0.739), la précision reste très faible (avec la précision la plus haute étant de 0.656 avec “NotificationCompat.Builder”). Cela ne change pas lorsque la déclaration de la permission requise pour les notifications n’est pas considérée comme étant nécessaire pour la présence d’une notification. Dans ce cas de figure, l’approche permet aussi un rappel élevé (avec le motif “showNotification” offrant un rappel de 0.739), mais la précision reste faible (avec le motif offrant la plus haute précision, “enqueueNotification”, n’offrant une précision que de 0.508).
Autant lorsque la présence de la permission est vérifiée que lorsqu’elle est ignorée, on remarque que le motif permettant d’obtenir le score F1 le plus élevé est le motif “enqueueNotification”, qui offre une précision de 0.508 et un rappel de 0.717 pour un score F1 de 0.613.

Cette mauvaise performance peut provenir de plusieurs sources: les bibliothèques de publicités ainsi que des données possiblement erronées dans l’échantillon fourni.
La cause principale de faux positif est que les bibliothèques de publicités implémentent leur propre système de notification interne pour, entre autres, suivre le téléchargement ou l’affichage de récompenses. Ces bibliothèques, afin d’implémenter leur système de notification, font usage de l’API de notification d’Android et, conséquemment, les motifs utilisés par une application pour afficher une notification peuvent aussi être utilisés par une bibliothèque de publicité même si l’application ne contient pas de notifications. Ces motifs étant présents dans les mêmes fichiers que l’application en elle-même, une analyse plus approfondie du code source serait nécessaire pour être en mesure d’ignorer les faux positifs causés par ces implémentations.

De plus, lors de la vérification manuelle pendant la validation des motifs, quelques applications contenaient des notifications même si le fichier Excel indiquait que l’application n’en contenait pas. Pour confirmer la présence de notification, certaines de ces applications ont nécessité plusieurs heures d’attente après une période d’utilisation active alors que d’autres nécessitaient plusieurs heures d’utilisations avant que les notifications puissent apparaître.

Bien qu’il soit possible que ces erreurs dans l’échantillon fourni ne soient que des exceptions, ces erreurs montrent le besoin d’automatisation pour la détection des anti-fonctionnalités dans les applications mobiles. En effet, les notifications des applications faussement identifiées comme ne contenant aucune notification ne sont apparues qu’après plusieurs heures d’inactivité après l’analyse manuelle ou ont

requis une longue interaction avec l'application avant que celle-ci n'ait le potentiel d'utiliser les notifications. Une telle demande de temps rend une analyse manuelle plus que déraisonnable pour identifier à large échelle les applications contenant des notifications – surtout si ladite analyse doit être réalisée lors de chaque mise à jour d'une application. Aussi, le temps requis avant qu'une notification soit envoyée par une application n'est pas constant et peut varier significativement entre les applications, ce qui peut compliquer l'analyse manuelle d'une application: pour garantir qu'une application ne contienne pas de notifications, il est nécessaire d'interagir avec l'application de manière prolongée en plus d'attendre un long moment; dans des cas extrêmes, il est même possible que l'application nécessite plusieurs sessions d'utilisation distinctes avant que l'utilisateur n'ait accès aux fonctionnalités de l'application ayant le potentiel d'utiliser des notifications ou que l'application utilise les notifications pour rappeler son existence et que plusieurs jours d'inutilisation soient nécessaires pour recevoir une telle notification.

7.4 Microtransactions

L'approche utilisée offre des résultats mitigés pour les microtransactions. En effet, bien que les microtransactions peuvent être détectées, il y a peu de motifs permettant de spécialiser la détection vers une précision ou un rappel supérieur. Cela provient principalement de trois facteurs: les politiques de publication du Google Play Store, la part de marché de cette boutique d'application et la gestion des microtransactions sur celle-ci.

Comme mentionné précédemment, une application offrant des microtransactions doit, pour être publiée sur le Google Play Store, passer par l'entremise de la boutique d'application pour procéder aux microtransactions. Ainsi, les motifs pouvant être utilisés pour l'implémentation des microtransactions sont, pour les applications publiées sur le Google Play Store, relativement restreintes - surtout lorsque comparés à l'implémentation de publicités, qui ne possèdent pas de restrictions quant à leur provenance.

La plage de motifs pouvant être utilisée pour la classification de chaque type de microtransactions est cependant fortement limitée par la gestion des microtransactions de Google.
En effet, au niveau du traitement des microtransactions par Google, les microtransactions consommables et non consommables ne sont pas des catégories distinctes de microtransactions: ces deux catégories

consistent en un achat unique. Celles-ci sont donc implémentées avec les mêmes méthodes pour les fonctionnalités entourant les microtransactions telles que l'affichage des produits disponibles dans une application.

La principale différence entre les deux catégories se trouve lorsqu'une microtransaction consommable est consommée: une microtransaction ne pouvant être achetée qu'une seule fois, il est nécessaire, pour le développeur, d'utiliser "purchases.products.consume" pour signaler que la microtransaction spécifiée peut maintenant être achetée une nouvelle fois par le même utilisateur – cette utilisation ne se fait pas à partir de l'application en elle-même mais à partir d'un serveur opéré par le développeur de l'application.

	rappel	précision	F1
consumePurchase	0.686	0.632	0.659

Table 7.7: Motif utilisé pour la détection des microtransactions consommables et sa performance

Les microtransactions non consommables représentent aussi un défi dans le sens où il faut s'assurer que celle-ci ne puisse pas être consommée. Bien que cela soit plus facile dans les cas où une application ne contient aucune microtransaction consommable, puisque le problème revient simplement à déterminer si une application possède des microtransactions n'étant pas un abonnement, la détection devient plus difficile lorsqu'une application contient aussi des microtransactions consommables: il faut alors non seulement déterminer quels achats peuvent être effectués, mais aussi, pour chacun d'entre eux, s'ils peuvent être consommés. Une telle détermination demanderait une analyse plus en profondeur du code de l'application et ne peut être réalisée uniquement par le biais de recherche de motifs dans le code.

Similairement, la détection des abonnements représente aussi un défi important. En effet, bien que l'implémentation des abonnements diffère de l'implémentation des autres types de microtransactions, la grande majorité de ces différences se retrouvent dans les fichiers de la bibliothèque externe responsable des microtransactions – fichiers qui contient le code prenant en charge les abonnements même si l'application ne propose pas de telles microtransactions. Conséquemment, exclure ces fichiers de la recherche cause un rappel extrêmement bas alors que les inclure cause une précision extrêmement basse.

En plus de cette difficulté, la gestion des aspects de l'abonnement qui leur est propre, telle que la possibilité de renouveler l'abonnement, n'est pas gérée par l'application en elle-même, mais par le

serveur de la boutique d'applications et n'est, conséquemment, pas détectable dans le code de l'application avec l'approche utilisée par l'outil: l'application se contente d'acheter l'abonnement et vérifie, via les mêmes API que les autres types de microtransactions, si l'utilisateur possède la microtransaction correspondant à l'abonnement en question.

7.5 Conclusion

Les résultats obtenus montrent ainsi que la recherche de motifs dans le code source d'une application Android permet une certaine détection d'anti-fonctionnalités. La capacité de détection varie grandement selon l'antifonctionalité concernée, avec les publicités offrant la meilleure capacité de détection et les notifications poussées offrant la plus faible capacité de détection. De plus, on remarque aussi qu'une recherche basée non pas sur un motif unique mais sur un groupe de motifs peut aussi permettre une bonne capacité de détection – voire une meilleure capacité de détection selon l'importance accordée respectivement à la précision et au rappel pour l'antifonctionalité en question.

Il y avait deux questions de recherches: l'identification de motifs courants pour chaque anti-fonctionnalité étudiée (question 1) ainsi que l'identification des motifs offrant une meilleure précision au prix d'un rappel inférieur et vice-versa (question 2).

Au niveau de la première question, on remarque qu'une telle identification n'a pas été possible de manière satisfaisante pour toutes les anti-fonctionnalités. En effet, alors que de tels motifs ont pu être identifiés pour tous les types de publicités, il n'a pas été possible de faire une telle identification pour les notifications poussées. Aussi, bien qu'un motif a pu être identifié pour les microtransactions, il n'a pas été possible d'identifier des motifs spécifiques aux microtransactions consommables, non consommables ainsi qu'aux abonnements.

Au niveau de la seconde question, soit l'identification de motifs offrant des rappels et précisions différents, on remarque que les publicités ont permis d'obtenir une vaste "plage" de précision et de rappels alors que l'on retrouve le cas opposé pour les microtransactions, pour lesquelles un seul motif a pu être identifié. Les motifs identifiés pour les notifications poussées, en plus d'offrir des précisions et rappels laissant à désirer, n'ont pas permis d'obtenir une plage de précision et rappel aussi vaste que celle obtenue avec les publicités.

CHAPITRE 8
CONCLUSION

La présence d'anti-fonctionnalités telle que les publicités ou les microtransactions dans les applications mobiles pour enfants est un problème très important, autant en raison de la proportion d'enfants qui interagissent régulièrement avec un appareil mobile que par leur vulnérabilité a ces anti-fonctionnalités.

Une automatisation de la détection des anti-fonctionnalités présente dans une application est une nécessité en raison du grand nombre d'applications étant déjà présentes dans les boutiques d'application, ainsi que la quantité de nouvelles applications publiées sur ces boutiques chaque jour. Ce mémoire a présenté une stratégie simple et flexible permettant de détecter les anti-fonctionnalités présentes dans une application Android, en plus d'implémenter cette stratégie dans un outil permettant d'appliquer l'approche décrite dans ce mémoire à un ensemble d'applications Android.

L'approche utilisée consiste en une recherche de motifs dans l'ensemble des fichiers du code source d'une application donnée. Chaque anti-fonctionnalitée est associée à un motif distinct et, lorsque l'outil détecte la présence de ce motif dans le code source d'une application, l'anti-fonctionnalité en question est considérée comme étant présente dans l'application analysée. Cette approche permet une grande flexibilité pour la détection des anti-fonctionnalités. En effet, les changements requis pour une mise à jour de l'outil, par exemple si certaines bibliothèques de publicités deviennent moins populaires ou si les priorités de détection changent, sont minimes: les seuls changements requis sont la modification des motifs utilisés par l'outil ainsi que, si nécessaire, la modification des fichiers parcourus lors de l'analyse.

Les motifs utilisés ont été choisis avec deux approches, la première consistant en une recherche qualitative des ressources didactiques sur Internet afin d'identifier, pour chaque anti-fonctionnalité recherchée, les implémentations "populaires" et la seconde consistant en une analyse de l'information mutuelle dans le code source des applications de l'échantillon. Parmi ces deux approches, la première recherche a été la plus concluante; l'analyse de l'information mutuelle étant principalement utile pour avoir un aperçu permettant de guider les recherches manuelles. La principale raison pour cela est que l'information mutuelle n'est pas un indicateur de causalité en tant que tel, mais simplement de corrélation. Conséquemment, un motif peut posséder un haut score d'information mutuelle non pas

parce qu'il est utilisé pour implémenter une anti-fonctionnalité, mais plutôt parce que le motif est souvent utilisé dans les applications possédant ladite anti-fonctionnalité. On retrouve notamment le cas du motif "getDelegateFromInvocationHandler" pour les publicités interstitielles: malgré un score élevé, ce motif n'est pas utilisé pour implémenter ce type de publicité spécifiquement, mais est plutôt utilisé par WebKit, qui est un moteur de navigateur internet ouvert.

Bien que des motifs étant souvent utilisés pour implémenter les anti-fonctionnalités recherchées ont pu être identifiés, la capacité de discrimination de ces motifs varie significativement selon les anti-fonctionnalités en question. En effet, alors que les publicités interstitielles et des publicités récompensées peuvent être détectées avec un rappel et une précision relativement élevés, cela est moins le cas avec les notifications poussées et les microtransactions. Dans le cas des microtransactions, cette capacité moindre de détection est due à leurs implémentations, qui rendent la classification difficile avec une analyse statique, alors que dans le cas des notifications poussées, elle provient majoritairement de l'implémentation des bibliothèques de publicités.

De la même façon, la plage de valeurs varie significativement selon l'anti-fonctionnalité recherchée, avec les publicités offrant une plage de valeur vaste et les microtransactions n'en offrant aucune – ce dernier cas étant dû au fait qu'un seul motif permettant d'obtenir une discrimination acceptable a été identifié pour la détection.

8.1 Limites

Il y a plusieurs limites, autant au niveau du pipeline utilisé qu'à l'approche en elle-même. En effet, avec le pipeline actuel, les fichiers générés par les moteurs de jeu tels que Unity, Unreal Engine ou Godot ne sont pas décompilés. Cela peut conduire à des faux négatifs lorsque l'implémentation d'une anti-fonctionnalité est effectuée dans le moteur de jeu puisque ladite implémentation ne fera pas partie du code qui sera décompilé et ne sera donc pas analysé par l'outil. De plus, lorsqu'un motif est détecté, l'outil ne prend pas en compte si ledit motif est présent dans une portion du code qui est utilisé par l'application: il est supposé que la présence d'un motif dans les fichiers analysés pour ladite anti-fonctionnalité indique la présence de l'anti-fonctionnalité associée. Cette limite peut conduire à de faux positifs, par exemple si le code en question est un vestige d'une fonctionnalité passée de l'application. De

plus, les résultats obtenus par chaque motif peuvent être influencés par la popularité des différentes implémentations possibles d'une anti-fonctionnalité donnée. Cet aspect est d'autant plus important pour la détection des publicités: la majorité de celles-ci étant implémentée par le biais d'une bibliothèque externe provenant d'un réseau de publicités, il est possible que la précision ou le rappel diminuent pour un motif donné si la popularité des différents réseaux de publicité change ou si un réseau de publicités modifie l'implémentation de certains types de publicités.

8.2 Recherches futures

Dans un cadre de recherche future, il serait intéressant de modifier le pipeline de décompilation afin que les fichiers générés par les moteurs de jeu, si utilisés par une application, soient aussi décompilés pour être analysés par l'outil. Étant donné que chaque moteur nécessiterait un outil distinct pour la décompilation, il serait nécessaire d'identifier le moteur de jeu utilisé par l'application afin que l'outil de décompilation correspondant soit utilisé. Cette identification permettrait aussi d'identifier, pour une même anti-fonctionnalité, des motifs distincts pour chaque moteur de jeu, ce qui pourrait potentiellement contribuer à une meilleure détection et classification.

Une identification similaire pourrait être utilisée pour améliorer les résultats de détection pour les anti-fonctionnalités souvent implémentées avec des bibliothèques tierces, telles que les publicités. S'il est possible d'identifier chaque bibliothèque concernant les publicités utilisées par l'application, il pourrait être possible de spécialiser les motifs pour viser spécifiquement les bibliothèques utilisées par l'application et ainsi potentiellement réduire le nombre de faux positifs.

Aussi, quelques erreurs ont été remarquées dans l'échantillon fourni au niveau de la présence de notification – plus spécifiquement, deux applications ayant faussement été identifiées comme ne contenant pas de notifications poussées. Il n'a pas été déterminé si ces erreurs n'étaient que de rares exceptions ayant une influence négligeable sur les résultats obtenus ou si la quantité d'erreurs pour ce type d'anti-fonctionnalité était suffisamment élevée pour avoir une forte influence sur lesdits résultats.

Finalement, il serait intéressant d'utiliser une approche basée sur une analyse dynamique des applications Android pour la détection d'anti-fonctionnalités. En effet, une telle approche permettrait

non seulement la détection d'anti-fonctionnalités plus difficiles à détecter par une analyse statique, telle que certains enjeux au niveau de la confidentialité, mais aussi de vérifier si une anti-fonctionnalité est véritablement utilisée dans l'application. Une analyse dynamique des applications Android permettrait aussi de capturer le trafic Internet de l'application analysé, ce qui pourrait potentiellement permettre d'améliorer la fiabilité de la détection des publicités par le biais de l'analyse dudit trafic dans le but de détecter les communications avec les réseaux de publicités populaires ou encore la détection de la présence de microtransactions via une analyse du trafic entre l'application et les boutiques d'applications.

BIBLIOGRAPHIE

42matters. (2025a, July 10). *Google Play Statistics and Trends 2025*. Google Play Statistics and Trends 2025. https://42matters.com/google-play-statistics-and-trends

42matters. (2025b, July 10). *IOS Apple App Store Statistics and Trends 2025*. Google Play Statistics and Trends 2025. https://42matters.com/ios-apple-app-store-statistics-and-trends

Alamri, H., Maple, C., Mohamad, S., & Epiphaniou, G. (2022). Do the Right Thing: A Privacy Policy Adherence Analysis of over Two Million Apps in Apple iOS App Store. *Sensors (Basel, Switzerland)*, *22*(22), 8964. https://doi.org/10.3390/s22228964

Amrita. (2023, May 18). What is Android Treble? *HSC*. https://www.hsc.com/resources/blog/what-is-android-treble/

Android Open Source Project. (2025a). *Android 13 for TV*. Android Developers. https://developer.android.com/tv/release/13

Android Open Source Project. (2025b). *Hardware abstraction layer (HAL) overview*. Android Open Source Project. https://source.android.com/docs/core/architecture/hal

Android Open Source Project. (2025c). *What is Android Automotive?* Android Open Source Project. https://source.android.com/docs/automotive/start/what_automotive

Apache Software Foundation. (2004). *Apache License, Version 2.0*. https://www.apache.org/licenses/LICENSE-2.0

Apple. (n.d.). *Creating Your Product Page*. Apple Developer. Retrieved 1 September 2025, from https://developer.apple.com/app-store/product-page/

Apple. (2025). *App Review Guidelines*. Apple Developer. https://developer.apple.com/app-store/review/guidelines/

Auxier, B., Anderson, M., Perrin, A., & Turner, E. (2020, July 28). 1. Children's engagement with digital devices, screen time. *Pew Research Center*. https://www.pewresearch.org/internet/2020/07/28/childrens-engagement-with-digital-devices-screen-time/

Brooks, G. A., & Clark, L. (2019). Associations between loot box use, problematic gaming and gambling, and gambling-related cognitions. *Addictive Behaviors*, *96*, 26–34. https://doi.org/10.1016/j.addbeh.2019.04.009

Cen, L., Si, L., Li, N., & Jin, H. (2014). User comment analysis for android apps and CSPI detection with comment expansion. *CEUR Workshop Proceedings*, *1225*, 25–30.

Chen, Y., Zhu, S., Xu, H., & Zhou, Y. (2013). Children's Exposure to Mobile In-App Advertising: An Analysis of Content Appropriateness. *2013 International Conference on Social Computing*, 196–203. https://doi.org/10.1109/SocialCom.2013.36

Crussell, J., Stevens, R., & Chen, H. (2014). MAdFraud: Investigating ad fraud in android applications. *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '14*, 123–134. https://doi.org/10.1145/2594368.2594391

Data.ai. (2023, June 17). *Advertising Drives $2 Out of Every $3 Spent on Mobile*. https://web.archive.org/web/20230617135947/https://www.data.ai/en/insights/market-data/state-of-app-revenue-2023/

Dong, F., Wang, H., Li, L., Guo, Y., Bissyandé, T. F., Liu, T., Xu, G., & Klein, J. (2018). FraudDroid: Automated ad fraud detection for Android apps. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 257–268. https://doi.org/10.1145/3236024.3236045

Google. (n.d.). *Prepare your app for review—Play Console Help*. Retrieved 27 March 2025, from https://support.google.com/googleplay/android-developer/answer/9859455?hl=en

Gorla, A., Tavecchia, I., Gross, F., & Zeller, A. (2014). Checking app behavior against app descriptions. *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, 1025–1035. https://doi.org/10.1145/2568225.2568276

Grace, M. C., Zhou, W., Jiang, X., & Sadeghi, A.-R. (2012). Unsafe exposure analysis of mobile in-app advertisements. *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks, WISEC '12*, 101–112. https://doi.org/10.1145/2185448.2185464

Hamari, J., Alha, K., Järvelä, S., Kivikangas, J. M., Koivisto, J., & Paavilainen, J. (2017). Why do players buy in-game content? An empirical study on concrete purchase motivations. *Computers in Human Behavior*, *68*, 538–546. https://doi.org/10.1016/j.chb.2016.11.045

Hecht, G. (2015). An approach to detect Android antipatterns. *Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15*, 766–768. https://dl.acm.org/doi/10.5555/2819009.2819161

Huang, J., Zhang, X., Tan, L., Wang, P., & Liang, B. (2014). AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, 1036–1046. https://doi.org/10.1145/2568225.2568301

Mahipal, A. (2020, March 3). Kids mobile gaming report: More than two-thirds of parents worry kids overspending on in-app purchases. *SellCell.Com Blog*. https://www.sellcell.com/blog/more-than-two-thirds-of-parents-worry-kids-overspending-on-in-app-purchases/

Mauthe, N., Kargén, U., & Shahmehri, N. (2021). A Large-Scale Empirical Study of Android App Decompilation. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 400–410. https://doi.org/10.1109/SANER50967.2021.00044

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., & De Lucia, A. (2017). Lightweight detection of Android-specific code smells: The aDoctor project. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 487–491. https://doi.org/10.1109/SANER.2017.7884659

Pandita, R., Xiao, X., Yang, W., Enck, W., & Xie, T. (2013). WHYPER: Towards automating risk assessment of mobile applications. *Proceedings of the 22nd USENIX Conference on Security, SEC'13*, 527–542.

Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., & Chen, Z. (2014). AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 1354–1365. https://doi.org/10.1145/2660267.2660287

Reyes, I., Wijesekera, P., Reardon, J., On, A. E. B., Razaghpanah, A., Vallina-Rodriguez, N., & Egelman, S. (2018). "Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale. *Proceedings on Privacy Enhancing Technologies*, *2018*(3), 63–83. https://doi.org/10.1515/popets-2018-0021

Salehudin, I., & Alpert, F. (2021). To pay or not to pay: Understanding mobile game app users' unwillingness to pay for in-app purchases. *Journal of Research in Interactive Marketing*, *16*(4), 633–647. https://doi.org/10.1108/JRIM-02-2021-0053

Samsung. (n.d.). *App Distribution Guide*. Samsung Developer. Retrieved 1 September 2025, from https://developer.samsung.com/galaxy-store/distribution-guide.html

Samsung. (2019). *Samsung Celebrates Best of Galaxy Store Awards at SDC 2019*. Samsung Global Newsroom. https://news.samsung.com/global/samsung-celebrates-best-of-galaxy-store-awards-at-sdc-2019

Shah, S. A., & Phadke, V. D. (2023). Mobile phone use by young children and parent's views on children's mobile phone usage. *Journal of Family Medicine and Primary Care*, *12*(12), 3351–3355. https://doi.org/10.4103/jfmpc.jfmpc_703_23

Stevens, R., Gibler, C., Crussell, J., Erickson, J., & Chen, H. (2012). Investigating User Privacy in Android Ad Libraries. *IEEE Mobile Security Technologies (MoST)*.

Story, P., Zimmeck, S., & Sadeh, N. (2018). Which Apps Have Privacy Policies? In M. Medina, A. Mitrakas, K. Rannenberg, E. Schweighofer, & N. Tsouroulas (Eds.), *Privacy Technologies and Policy* (pp. 3–23). Springer International Publishing. https://doi.org/10.1007/978-3-030-02547-2_1

Su, M.-Y., Wei, H.-S., Chen, X.-Y., Lin, P.-W., & Qiu, D.-Y. (2018). Using Ad-Related Network Behavior to Distinguish Ad Libraries. *Applied Sciences*, *8*(10), Article 10. https://doi.org/10.3390/app8101852

Tila, C. (2024). *Huawei: MAUs of Huawei AppGallery*. Statista. https://www.statista.com/statistics/1269510/monthly-active-users-of-huawei-appgallery/

Tila, C. (2025). *Android top mobile app ad network SDKs 2025*. Statista. https://www.statista.com/statistics/1035623/leading-mobile-app-ad-network-sdks-android/

Winder, D. (2024). *New Google Play Store Leak—2.5 Billion Users Could Soon See Apps Vanish*. Forbes. https://www.forbes.com/sites/daveywinder/2024/11/23/google-play-store-leak-25-billion-users-could-soon-see-apps-vanish/

Yu, L., Luo, X., Qian, C., Wang, S., & Leung, H. K. N. (2018). Enhancing the Description-to-Behavior Fidelity in Android Apps with Privacy Policy. *IEEE Transactions on Software Engineering*, *44*(9), 834–854. https://doi.org/10.1109/TSE.2017.2730198

Yue, C., Zhong, C., Chen, K., Zhang, Z., & Lee, Y. (2024, August 14). DARKFLEECE: Probing the Dark Side of Android Subscription Apps. *33rd USENIX Security Symposium (USENIX Security 24)*.

Zendle, D., & Cairns, P. (2018). Video game loot boxes are linked to problem gambling: Results of a large-scale survey. *PLOS ONE*, *13*, e0206767. https://doi.org/10.1371/journal.pone.0206767

Zendle, D., & Cairns, P. (2019). Loot boxes are again linked to problem gambling: Results of a replication study. *PLoS ONE*, *14*(3), e0213194. https://doi.org/10.1371/journal.pone.0213194

Zhao, Y., Liu, T., Wang, H., Liu, Y., Grundy, J., & Li, L. (2023). Are Mobile Advertisements in Compliance with App's Age Group? *Proceedings of the ACM Web Conference 2023, WWW '23*, 3132–3141. https://doi.org/10.1145/3543507.3583534

Zhou, Y., Yue, T., Liu, X., Shen, C., Tong, L., & Ding, Z. (2021). *Payment-Guard*: Detecting fraudulent in-app purchases in iOS system. *Neurocomputing*, *422*, 263–276. https://doi.org/10.1016/j.neucom.2020.10.007

ANNEXE A

Android Runtime (ART) et Dalvik

Dalvik était un environnement d'exécution conçu spécialement pour des appareils avec des faibles ressources matérielles et performances. Il se basait sur une compilation « à la volée » des applications Android: au lieu de compiler l'intégralité du bytecode de l'application, il compilait uniquement les parties du code au moment où elles étaient utilisées. Ce comportement proche de celui d'un interpréteur, mais avec une étape de compilation avant l'exécution, permettait ainsi de réduire le temps d'installation des applications, puisque le code n'avait pas besoin d'être compilé, tout en réduisant l'espace disque requis étant donné qu'il n'y avait pas besoin de conserver le code compilé des applications. Cette approche, par contre, causait une réduction des performances des applications lors de l'exécution étant donné le besoin de compiler continuellement le bytecode. Cette perte de performance était cependant jugée un sacrifice acceptable pour éviter de long temps d'installations, tout en réduisant l'utilisation de la mémoire de l'appareil.

La montée de performances des appareils Android, autant en termes de mémoire que de processeurs, a rendu les objectifs initiaux de Dalvik moins importants et, conséquemment, la perte de performance due à la compilation "à la volée" n'était plus un compromis nécessaire: la puissance des processeurs modernes permettait une compilation dans des délais raisonnables et la plus grande mémoire permettait de conserver le code compilé de l'application. Les désavantages de l'approche utilisée par Dalvik - notamment la perte de performance due à la compilation ou l'usage plus élevé de la batterie en conséquence de l'usage plus important de la CPU – causaient un désagrément plus important que la satisfaction causée par le temps d'installation réduit et le plus faible usage de la mémoire. Pour cette raison, Dalvik fut remplacé par l'ART (pour *Android RunTime*) avec le déploiement de la version 5 d'Android en 2014.

L'ART utilisé par les versions actuelles d'Android utilise une approche hybride combinant la compilation "à la volée" et "anticipée" (consistant à compiler le code avant le démarrage de l'application). Lors de l'installation d'une application, seules les portions les plus essentielles du code – comme la séquence de démarrage ou les classes fréquemment utilisées – sont compilées et conservées dans le stockage de l'appareil alors que le reste du code reste en bytecode. Durant l'exécution de l'application, l'ART observe

le comportement de l'application, établit un profil des portions du code de l'application fréquemment utilisées et conserve ces données dans un fichier dédié situé dans les fichiers de l'application. Lorsque l'application n'est pas en cours d'exécution – et que certaines conditions, telles que l'appareil étant en cours de chargement, sont remplies - le système utilise ensuite ce fichier pour compiler les portions du code fréquemment utilisé par l'application et les conservera dans la mémoire de l'appareil. Ainsi, l'approche hybride utilisée par cet environnement d'exécution permet d'offrir un démarrage rapide de l'application (le code requis pour le démarrage étant déjà compilé) ainsi qu'une exécution minimalement impactée par la compilation "à la volée" (puisque celle-ci n'a lieu que dans les portions du code moins fréquemment utilisées). Cette même approche permet aussi de minimiser l'utilisation du stockage du téléphone, puisque le code compilé stocké dans l'appareil est uniquement le code fréquemment utilisé.

Notons que cette approche hybride n'a pas été introduite dès le déploiement de l'ART. En effet, l'approche originellement utilisée par cet environnement était une approche uniquement "anticipée" : l'entièreté du code de l'application était compilée lors de l'installation et conservée dans la mémoire de l'appareil, ce qui pouvait causer des longs temps d'installation ainsi qu'une haute utilisation de la mémoire. L'approche actuellement utilisée, soit l'approche hybride, a été introduite lors de la version 7 d'Andoid.

ANNEXE B

HARDWARE ABSTRACTION LAYER

Le HAL (pour *Hardware Abstraction Layer*) actuellement utilisé dans l'architecture Android a connu des évolutions importantes au cours du développement du système Android. En effet, bien que cette couche existait avant la version 8 d'Android, celle-ci était significativement moins modulaire que le HAL présent dans les versions suivantes.

Avant Android 8, cette couche était constituée d'un ensemble de bibliothèques partagées écrites en C ou en C++ que le cadre Android appelait directement. Contrairement à l'implémentation actuelle, les appels n'étaient pas définis par une interface standardisée, mais étaient simplement exposés par le biais des fichiers d'en-tête et non versionnés. Cette architecture rendait l'implémentation du HAL et le système Android fortement couplés. Ce couplage ralentissait le déploiement des mises à jour d'Android aux utilisateurs d'appareil Android: le cadre logiciel et cadre matériel faisaient partie du même code et, pour déployer une mise à jour, un manufacturier d'appareils Android devait attendre que les fournisseurs des composantes matérielles de l'appareil publient une mise à jour des implémentations des composantes matérielles dont ils étaient responsables (Amrita, 2023).

Lors du déploiement de la version 8 d'Android, la conception de cette couche a été profondément restructurée dans le but de contrer ces problèmes et ainsi réduire le coût et le temps requis pour le déploiement des mises à jour des nouvelles versions d'Android pour les vendeurs d'appareils Android. Cette refonte repose principalement sur l'ajout d'une interface stable entre le cadre Android et l'implémentation matérielle. Grâce à cette interface, définie avec un langage de description d'interfaces, le système Android peut être modifié sans nécessiter une modification des implémentations du code. De plus, afin de garantir une plus grande stabilité lors des mises à jour du système, un système de versionnage a été ajouté.