UNIVERSITÉ DU QUÉBEC À MONTRÉAL

DYNAMICS : ÉTUDE EMPIRIQUE, SPÉCIFICATION ET DÉTECTION DYNAMIQUE DES DÉFAUTS DE CODE COMPORTEMENTAUX DANS LES APPLICATIONS MOBILES

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

DIMITRI PRESTAT

JUIN 2023

REMERCIEMENTS

Je tiens à exprimer ma sincère gratitude à mes directeurs de recherche, Naouel Moha et Roger Villemaire, pour leur soutien et leurs encouragements continus durant mon doctorat. Cette thèse n'aurait pas été possible si Naouel Moha ne m'avait pas proposé un sujet correspondant à mes attentes et aux domaines de recherche que je visais. Merci également à Florent Avellaneda qui m'a encadré vers la fin de ma thèse afin d'ajouter son expertise et son aide. Merci à vous tous pour votre patience et les très nombreuses itérations qu'il y avait à faire lors de la rédaction des articles. Si Omer Nguena-Timo ne m'avait pas accepté en stage pour valider mon master précédant cette thèse, je n'aurai jamais eu l'opportunité de découvrir le Québec ni eu l'opportunité d'effectuer ma thèse ici, je le remercie spécifiquement pour cela. Je n'aurai pas eu une telle envie d'effectuer une thèse si plusieurs de mes professeurs à l'Université de Bordeaux en France ne m'avaient pas donné l'envie et encouragé à me lancer dans la recherche, je tenais à les remercier.

Je remercie également les membres du jury : Hafedh Mili, Quentin Stievenart et Petko Valtchev, tous professeurs à l'Université du Québec à Montréal, ainsi que Raphaël Khoury, professeur à l'Université du Québec en Outaouais.

Je tiens à remercier Nour Khezemi pour son assistance durant son stage lors de la phase d'expérimentation de l'outil présenté dans cette thèse ainsi que son aide dans l'étude des résultats. Je remercie également toutes les personnes qui ont participé à la validation des résultats.

J'aimerais remercier toutes les personnes que j'ai pu rencontrer à Montréal, notamment Rébecca pour m'avoir soutenu une bonne partie de ma thèse et Eugénie

ainsi que sa famille pour leur accueil lors de mon arrivée sur le continent.

J'aimerais remercier tous mes amis qui m'ont accompagné et encouragé pendant cette longue thèse : Théo, Nikita, Evelyne, Elisa, Augustin, Saïfallah, Paul, Lisa, Quentin, Imane, Thomas, Amadou, Lucile, Bérénice et Marion. Remerciements spécifiques à Geoffroy qui m'a accompagné lors de chaque étape de ma vie et Anne qui m'a particulièrement aidé lors de la relecture de cette Thèse. Merci à tout le monde, désolé à ceux que j'ai oubliés et ceux dont je me suis éloigné pendant ces 4 ans. Merci de m'avoir aidé à tenir pendant les moments difficiles, même malgré la distance pour beaucoup d'entre vous, particulièrement lors de cette pandémie.

Un énorme merci à toute ma famille. Tout particulièrement à ma mère qui m'a tant donné, et à qui je m'excuse d'avoir été aussi loin lorsqu'elle a dû affronter de lourds problèmes de santé. Merci pour tout ce que tu m'as apporté.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

Les défauts de code sont le résultat de mauvais choix de conception au sein des systèmes logiciels qui complexifient le code source et entravent l'évolution et les performances. Par conséquent, la détection des défauts de code dans les systèmes logiciels est une priorité importante pour réduire la dette technique. En outre, l'émergence des applications mobiles (apps) a fait apparaître de nouveaux types de défauts de code spécifiques à Android, qui sont liés aux limitations et aux contraintes sur les ressources comme la mémoire, les performances et la consommation d'énergie. Parmi ces défauts de code spécifiques à Android, on trouve ceux qui décrivent un comportement inapproprié pendant l'exécution et qui peuvent avoir un impact négatif sur la qualité du logiciel. Ces derniers sont considérés comme des défauts de code *comportementaux*. Les outils de détection des défauts de code utilisant de l'analyse statique présentent toutefois des limites pour la détection de ces défauts de code comportementaux, et la détection correcte de ces derniers nécessite la prise en compte du comportement dynamique des applications.

Tout d'abord, nous fournissons une étude empirique qui compare les résultats des outils de détection de la littérature, avec les définitions textuelles de sept défauts de code comportementaux décrits dans la littérature. Cette étude empirique vise à répondre à deux questions de recherche. Premièrement, les outils de détection sont-ils efficaces pour détecter les défauts de code comportementaux ? Deuxièmement, les défauts de code comportementaux détectés par les outils sont-ils cohérents avec leur définition littérale originale ? Enfin, afin de détecter dynamiquement les défauts de code comportementaux, nous proposons trois contributions : (1) une méthode, la méthode Dynamics, une méthode étape par étape pour la spécification et la détection dynamique des défauts de code comportementaux d'Android ; (2) un outil, l'outil Dynamics, mettant en œuvre cette méthode sur sept défauts de code ; et (3) une validation de notre approche.

Notre méthode se compose de quatre étapes : (1) la spécification des défauts de code ; (2) l'instrumentation de l'application ; (3) l'exécution des applications ; et (4) la détection des défauts de code comportementaux. Nos résultats montrent que de nombreux cas de défauts de code, qui ne peuvent pas être détectés avec des outils de détection statiques, sont effectivement détectés de façon précise avec notre approche dynamique.

Mots-clés : Android, défaut de code, défaut de code comportemental, spécification, détection, étude empirique, applications mobiles, comportement, analyse dynamique

CHAPITRE I

INTRODUCTION

Dans ce chapitre, nous introduisons le contexte nécessaire à la compréhension du sujet, les motivations ainsi que les problématiques qui ont mené à l'élaboration de cette thèse. Nous présentons également les principales contributions réalisées durant cette thèse.

1.1 Contexte

Dans cette section, nous posons le contexte de notre thèse. Plus précisément, nous allons présenter les différents mots-clés qui composent le titre de notre thèse, à savoir : analyse *dynamique*, *défauts de code*, *comportementaux*, *applications mobiles* et *spécification*.

1.1.1 Défauts de code

Les défauts de code (Fowler, 1999) (*Code smells* en anglais) sont des mauvaises pratiques de conception logicielles issues de mauvais choix d'implémentation ou de conception. Ces défauts conduisent à une complexification du code source, entravent l'évolutivité et pénalisent la maintenance de celui-ci. Les défauts de code sont souvent issus du développement à un rythme rapide des logiciels qui évoluent

constamment pour répondre aux nouvelles exigences des utilisateurs. Ces développements rapides visant à résoudre des bogues ou à ajouter des fonctionnalités manquantes dans un délai restreint peuvent conduire à l'introduction de défauts de code.

Ces défauts de code ne sont pas des problèmes fonctionnels ni des bogues, mais des problèmes renforçant la dette technique (Suryanarayana *et al.*, 2014). Ils participent donc à la dégradation de la qualité du logiciel et à la génération de coûts de développement supplémentaires. En effet, comme les défauts de code ne sont pas forcément des erreurs, ils peuvent être ignorés et donc persister sans perspective d'évolution ou de correction dans un logiciel. Les défauts de code sont par définition corrigibles par le réusinage (refactoring en anglais). Le réusinage est le processus de modification d'un logiciel sans altérer le comportement externe du code tout en améliorant sa structure interne. C'est une façon disciplinée de nettoyer le code (d'où le terme "*code smells*" en anglais) qui minimise les risques d'introduction de bogues. Le réusinage permet donc d'améliorer la conception du code après qu'il ait été écrit.

Les défauts de code, initialement définis par Fowler (Fowler, 1999), sont originalement orientés objet. Ils peuvent correspondre par exemple à de la duplication de code (Fowler, 1999), qui est un défaut de code présent lorsqu'une portion de code est présente à plusieurs endroits. La qualité du programme peut être augmentée par une factorisation, par exemple en créant une méthode commune. Il peut également s'agir d'une méthode trop longue ou d'une classe trop large (Fowler, 1999). Un tel défaut de code rend difficile la compréhension du code en donnant trop de responsabilités au sein d'une même classe. Une refactorisation possible pour pallier ce défaut est la séparation en plusieurs méthodes ou plusieurs classes. Ces défauts de code concernent principalement la structure du code, mais d'autres catégories de défauts de code ont été identifiées et définies. Les défauts de code

peuvent affecter plusieurs aspects, comme l'architecture (Fontana *et al.*, 2017), la performance (Chambers et Scaffidi, 2013) ou bien encore la sécurité (Rahman *et al.*, 2019). Il est intéressant de noter que d'une recherche à l'autre, les termes "code smells", "bad smells", "antipatterns" sont utilisés pour désigner la même chose. Nous résumons l'ensemble de ces appellations par la traduction "défauts de code".

1.1.2 Défauts de code spécifiques aux applications mobiles

Au cours des dernières décennies, le marché des applications mobiles a connu une croissance fulgurante et les applications mobiles sont passées de simples applications à des systèmes complexes en constante évolution. Par exemple, en 2021, il y avait plus de cinq millions d'applications disponibles dans divers magasins d'applications [1], avec plus de 230 milliards de téléchargements en 2021 [2]. Par *applications mobiles*, nous désignons uniquement les logiciels développés spécifiquement pour être exécutés sur des périphériques tels que des téléphones ou des tablettes. Ces applications ont donc un système d'exploitation à part, comme Android ou iOS, mais sont développées en utilisant des langages orientés objet classiques, comme le Java ou le C#. Toutefois, ces langages sont adaptés aux systèmes d'exploitation par l'introduction de classes et d'interfaces spécifiques.

Les applications mobiles suscitent de nouvelles préoccupations, telles que la consommation d'énergie, la mémoire limitée, les performances restreintes, l'affichage réactif ou encore les permissions autorisées par application. En raison de ces nouvelles

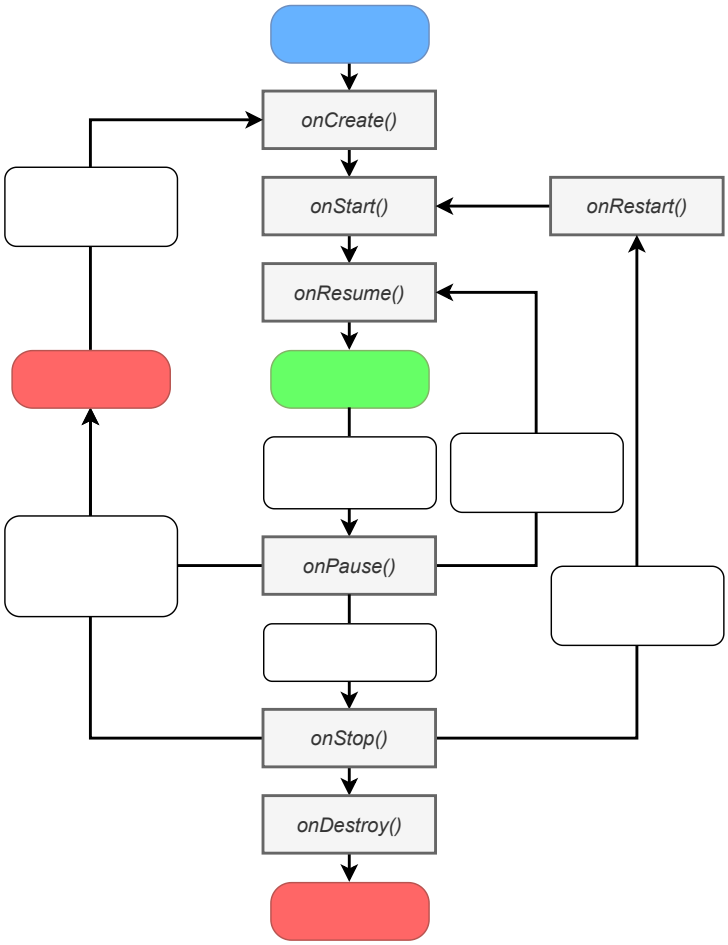
---

1. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

2. https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/

préoccupations, la communauté des chercheurs a relevé de nouveaux *défauts de code spécifiques aux applications mobiles*, les décrivant soigneusement pour les détecter et les corriger (Reimann *et al.*, 2014) (Hecht *et al.*, 2015b) (Palomba *et al.*, 2017). De plus, les applications mobiles semblent plus propices à l'apparition de défauts de code. En effet, les applications mobiles sont mises à jour de façon plus régulière que les applications classiques, afin de répondre aux demandes du marché (McIlroy *et al.*, 2015). Or, il a été montré que les développeurs soumis à une charge de travail et à une pression de publication élevées sont plus enclins à introduire des défauts de code (Tufano *et al.*, 2015).

FIGURE 1.1 Cycle de vie d'une activité Android.



Dans le cadre de cette thèse, nous nous concentrons particulièrement sur les applications mobiles Android. Nous avons fait ce choix puisqu'Android est un système d'exploitation libre. Une grande majorité des applications mobiles, environ 63%, sont publiées sur Google Play, le magasin d'applications pour Android [3]. Enfin, les principaux travaux sur la détection des défauts de code dans les applications mobiles (Hecht *et al.*, 2015b) (Palomba *et al.*, 2017) et les défauts de code spécifiques aux applications mobiles de la littérature (Rasool et Ali, 2020) se concentrent déjà sur Android.

Android se développe en utilisant le langage Java ou Kotlin. Cependant, Android n'utilise pas la machine virtuelle Java classique, mais une machine virtuelle optimisée pour les périphériques mobiles, comme Dalvik et ART. Les applications Android sont composées d'activités ayant des cycles de vie spécifiques tels que représentés à la figure 1.1. Le cycle de vie d'une application passe par le démarrage de l'application, son exécution, sa mise en pause lorsqu'elle est par exemple en arrière-plan ainsi que son arrêt. De nombreux défauts de code sont issus de ce cycle de vie, comme de mauvaises pratiques de programmation entraînant une consommation énergétique trop importante lorsque l'application est en arrière-plan. Les défauts de code spécifiques à Android sont divisés en plusieurs catégories. Nous retrouvons par exemple les défauts de code affectant la qualité (quality code smells) (Reimann *et al.*, 2014), les défauts de code affectant la consommation énergétique (energy code smells) (Gottschalk *et al.*, 2012) ou encore les défauts de code concernant la sécurité (security code smells) (Gadient *et al.*, 2018).

---

3. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

1.1.3 Défauts de code comportementaux

Les contraintes introduites par Android et son cycle de vie spécifique sont propices à l'introduction de défauts liés au comportement de l'application. Certains de ces défauts de code spécifiques à Android sont considérés comme des défauts de code *comportementaux*. Nous introduisons la catégorie des défauts de code *comportementaux* afin de catégoriser ces défauts de code spécifiques à Android liés au comportement de l'application. L'ensemble des défauts de code comportementaux est un ensemble de défauts de code contenant une partie des défauts de code spécifiques à Android, mais il n'est pas exclu qu'ils contiennent des défauts de code spécifiques à d'autres plateformes.

Nous définissons un défaut de code comportemental comme des caractéristiques dans le code source induisant un comportement inapproprié du code pendant l'exécution qui peut avoir un impact négatif sur la qualité du logiciel en ce qui concerne la consommation d'énergie, la mémoire ou la performance. Le terme "comportement" fait spécifiquement référence au comportement d'exécution du code, c'est-à-dire une occurrence ou une séquence d'événements ou d'actions observables du code pendant son exécution. Il n'est pas impossible que des défauts de code comportementaux induisent un comportement inapproprié sur d'autres aspects, comme la sécurité ou l'interface utilisateur. Les défauts de code comportementaux sont un sous-ensemble des défauts de code spécifiques à Android. L'ensemble des défauts de code comportementaux contient des défauts de code relatifs à la performance, la consommation énergétique ou la mémoire, sans forcément contenir tous les défauts de code relatifs à ces aspects. Plus précisément, cela veut dire qu'il existe des défauts de code relatifs à la performance, la consommation énergétique ou la mémoire qui ne sont pas comportementaux.

À notre connaissance, les défauts de code comportementaux ne sont pas expli-

citement mentionnés ou définis dans la littérature. Les défauts de code comportementaux connus se classent dans trois catégories : 1) les défauts caractérisés par l'utilisation d'un appel de méthode ou d'une séquence d'appels de méthodes pendant l'exécution ; 2) les défauts caractérisés par des problèmes pendant l'exécution d'une méthode, comme le temps d'exécution d'une méthode trop long ou une utilisation excessive de la mémoire ; 3) les défauts caractérisés par des variations de données indésirables pendant l'exécution, comme la taille d'une structure devenant trop grande.

1.1.4 Analyse dynamique

Comme vu dans la section précédente, les défauts de code comportementaux prennent en compte le comportement de l'application et sont liés à l'exécution de l'application. L'*analyse dynamique* est une analyse de logiciel qui implique l'exécution du programme en question. L'analyse dynamique est en opposition à l'analyse statique qui ne nécessite pas l'exécution, mais l'analyse du code source. L'analyse dynamique permet d'étudier le comportement de l'application ainsi que les effets de son exécution. Une analyse dynamique peut, de plus, être soit en ligne, c'est-à-dire pendant l'exécution de l'application, soit hors ligne, c'est-à-dire après l'exécution de l'application en analysant ce qu'il s'est passé. L'analyse dynamique est souvent utilisée dans le contexte de l'application mobile, mais très principalement et majoritairement dans la sécurité pour la détection de logiciels malveillants (Amin *et al.*, 2019) (Thangaveloo *et al.*, 2020) (Razgallah et Khoury, 2021).

1.1.5 Méthodes formelles

Les *méthodes formelles* sont des techniques utilisant de la logique mathématique sur un programme informatique afin de démontrer leur validité par rapport à

une spécification. Bien que les *méthodes formelles* n'apparaissent pas directement dans le titre, c'est par leur biais que nous allons faire la *spécification* des défauts de code comportementaux. Différentes approches formelles existent, mais pour prendre en compte le comportement d'un système les plus communes sont la vérification de modèles et la vérification à la volée. La *vérification de modèles* (*model checking* en anglais) se concentre sur la représentation du comportement du système à l'aide d'un modèle et de propriétés. La spécification formelle est souvent spécifiée à l'aide de propriétés, souvent temporelles, qui ne doivent pas être violées. Le modèle représente le comportement du système ou une partie du comportement du système réel, les modèles les plus communs sont des automates ou une machine à états finis, plus ou moins étendus. La spécification formelle et le modèle sont, respectivement, des abstractions de la spécification informelle et du système. La vérification à la volée (*runtime verification* en anglais) est une technique des méthodes formelles pour détecter et réagir à des comportements respectant ou violant certaines propriétés sur des systèmes en pleine exécution. Les propriétés sont très souvent exprimées formellement à l'aide de logique. La plus répandue est la logique temporelle linéaire. Ces propriétés peuvent également être représentées à l'aide de modèles comme des machines à états finis.

1.2 Motivation : Améliorer la qualité logicielle des applications mobiles en détectant les défauts de code comportementaux

Comme énoncé dans le contexte, cette thèse est motivée par la présence de plus en plus importante des applications mobiles, particulièrement Android. Nous avons également vu que les applications mobiles étaient propices à l'apparition de défauts de code. Nous nous concentrons sur les défauts de code comportementaux dans les applications mobiles pour deux raisons principales. Premièrement, ils peuvent nuire à la qualité logicielle des applications mobiles, et plus particulièrement en

ce qui concerne la consommation d'énergie, la mémoire et la performance. Les défauts de code peuvent avoir un impact sur la maintenance et l'évolutivité, mais les défauts de code spécifiques à Android, particulièrement les comportementaux, peuvent également avoir un impact lourd sur la qualité du logiciel et donc l'expérience utilisateur. En effet, une application trop lente ou qui ne répond pas peut repousser de potentiels utilisateurs. Deuxièmement, les recherches existantes n'ont pas abordé spécifiquement leur détection. Par conséquent, nous voulons évaluer si les outils actuels de détection des défauts de code sont efficaces pour identifier ces défauts de code ainsi que proposer un outil pour pouvoir efficacement les spécifier et les détecter. Enfin, nous voulons spécifiquement détecter les défauts de code par le biais de l'analyse dynamique, puisque c'est l'une des solutions les plus adaptées pour prendre en compte le comportement de l'application pendant l'exécution. Pour appuyer ce propos, une thèse réalisée sur la compréhension des défauts de code spécifiques aux applications mobiles (Habchi, 2019) indique que les outils de détection actuels devraient être améliorés à l'aide de l'analyse dynamique lorsque les défauts de code décrivent des scénarios d'exécution. En effet, certains défauts ont besoin de métriques comme le temps d'exécution pour être correctement détectés.

1.3 Problèmes

Dans cette section, nous présentons les deux problèmes qui ont guidé cette thèse. Le premier problème concerne les approches actuelles qui ne permettent pas de détecter correctement les défauts de code comportementaux. Le deuxième problème concerne l'absence d'approches dynamiques pour la détection des défauts de code comportementaux.

1.3.1 Problème n°1 : Les approches actuelles ne permettent pas de détecter correctement les défauts de code comportementaux

Les défauts de code spécifiques à Android ont fait l'objet de nombreuses études (Habchi, 2019) (Reimann *et al.*, 2014) (Hecht *et al.*, 2015a) et sont détectés par de nombreux outils (Hecht, 2016) (Palomba *et al.*, 2017) (Rasool et Ali, 2020). Cependant, tous ces outils sont basés sur l'analyse statique, utilisant la recherche par expressions régulières et le calcul de métriques. Par conséquent, ces outils ne prennent pas en compte le comportement de l'application, ce qui semble crucial pour la détection des défauts de code comportementaux.

Comme énoncé dans le contexte, les recherches existantes n'ont pas abordé spécifiquement la détection des défauts de code comportementaux. Par conséquent, nous voulons évaluer si les approches actuelles de détection des défauts de code spécifiques à Android sont efficaces pour identifier les défauts de code comportementaux. Cette problématique vise également à étudier si les défauts de code comportementaux détectés par ces approches sont cohérents avec leur définition littérale originale.

1.3.2 Problème n°2 : L'absence d'approches dynamiques pour la détection des défauts de code comportementaux

Comme l'annonce la problématique précédente, les outils existants ne prennent pas en compte le comportement de l'application. Il n'existe aucun outil permettant la détection de défauts de code spécifiques aux applications mobiles, notamment les défauts de code comportementaux, par analyse dynamique. De nombreuses recherches ont porté sur la détection des défauts de code dans les applications mobiles (Reimann *et al.*, 2014) (Rasool et Ali, 2020) (Palomba *et al.*, 2017) (Hecht *et al.*, 2015b) et de nombreuses recherches sur l'analyse dynamique des

applications mobiles (Bierma *et al.*, 2014) (Zheng *et al.*, 2014) (Amin *et al.*, 2019) (Thangaveloo *et al.*, 2020), ce qui démontre l'intérêt de la communauté pour ces sujets. Cependant, peu de travaux existent sur la détection des défauts de code via l'analyse dynamique et aucun sur la détection des défauts de code spécifiques aux applications mobiles via l'analyse dynamique. À notre connaissance, les seuls travaux concernant l'analyse dynamique sur la détection des défauts de code concernent le Javascript (Fard et Mesbah, 2013) et l'orienté objet (Kumar et Chhabra, 2014). Il y a donc un manque d'approches et d'outils pour détecter les défauts de code, en particulier les défauts de code comportementaux, en utilisant l'analyse dynamique dans les applications mobiles. C'est appuyé par le fait que des recherches évoquent l'importance de coupler les outils de détection statiques avec de l'analyse dynamique dans les travaux futurs envisagés (Habchi, 2019).

1.4 Thèse : Fournir une méthode et un outil pour la spécification et la détection des défauts de code comportementaux dans les applications mobiles

Notre thèse est qu'il est possible de fournir une approche dynamique qui permet de détecter correctement les défauts de code comportementaux dans les applications mobiles. Ainsi, nous proposons dans un premier temps une étude empirique pour montrer que les approches actuelles de la littérature ne sont pas efficaces pour la détection des défauts de code comportementaux. Ensuite, nous proposons une approche dynamique pour la spécification et la détection dynamique des défauts de code comportementaux.

Notre méthode dynamique s'articule en deux parties, tout d'abord nous proposons une méthode, la méthode DYNAMICS, qui contient toutes les étapes essentielles à la détection des défauts de code comportementaux en considérant le comportement de l'application. Notre méthode est basée sur quatre étapes séquentielles qui seront décrites au chapitre 5 : *Spécification*, *Traitement*, *Exécution* et *Détection*.

En deuxième lieu, nous avons implémenté concrètement la méthode DYNAMICS dans un outil, appelé l'outil DYNAMICS. L'outil DYNAMICS est basé sur quatre étapes qui sont des instances des étapes de la méthode DYNAMICS. Ces étapes seront décrites dans le chapitre 5 : *Spécification*, *Instrumentation*, *Exécution*, *Détection*.

1.5 Contributions

Dans cette section, nous présentons brièvement les contributions apportées au sein de notre thèse. Les contributions sont là pour répondre aux problématiques. Nous avons deux contributions principales chacune composée de plusieurs contributions secondaires. La contribution principale n°1 est une étude empirique et la contribution principale n°2 est la méthode et l'outil DYNAMICS.

1.5.1 Contribution n°1 : Une étude empirique sur la détection des défauts de code comportementaux dans la littérature

Afin de répondre à la première problématique, nous avons réalisé une étude empirique sur la détection des défauts de code comportementaux dans les applications mobiles. Cette contribution se compose de trois contributions secondaires afin d'étudier si les approches acutelles sont efficaces pour détecter les défauts de code comportementaux.

Premièrement, nous fournissons une étude empirique qui compare les résultats des outils de détection de la littérature, ADOCTOR et PAPRIKA, avec les définitions textuelles de sept défauts de code comportementaux décrits dans la littérature. Nous étudions si les outils en question sont efficaces pour détecter les défauts de code comportementaux. Ensuite, nous étudions si les défauts de code comportementaux détectés par les outils en question sont cohérents avec leur définition

littérale originale.

Deuxièmement, nous déduisons de cette étude une analyse approfondie des techniques de détection des outils afin d'identifier leurs limites et de partager les leçons tirées de notre étude empirique. Plus précisément, nous avons identifié les problèmes des règles de détection statique pour chaque défaut de code et nous avons apporté une solution dynamique pouvant résoudre les problèmes pour chacun d'entre eux.

Troisièmement, afin de traiter les défauts de code comportementaux qui ne sont ni définis ni étudiés dans l'état de l'art, il était important de les classifier. Nous avons donc identifié quels sont les défauts de code comportementaux parmi les 24 défauts de code spécifiques à Android détectés par les outils de référence de l'étude. Nous avons également divisé les défauts de code comportementaux en trois catégories.

1.5.2 Contribution n°2 : DYNAMICS, une méthode et un outil pour la détection des défauts de code comportementaux

Afin de répondre à la deuxième problématique, nous proposons DYNAMICS, une méthode et un outil capable de détecter les défauts de code comportementaux. Cette contribution se compose de trois contributions secondaires pour répondre au manque d'approches dynamiques pour la détection des défauts de code comportementaux.

Premièrement, nous proposons une méthode qui contient toutes les étapes nécessaires à la spécification et à la détection des défauts de code comportementaux. Il s'agit d'une contribution majeure, puisque tous les travaux récents sur la détection des défauts de code spécifiques à Android sont uniquement statiques(Hecht *et al.*, 2015a), (Palomba *et al.*, 2017) (Rasool et Ali, 2020).

Deuxièmement, nous implémentons notre méthode par le biais d'un outil, appelé Dynamics (DYNamic Analysis of Mobile app by Instrumentation for Code Smells). Dynamics permet d'effectuer toutes les étapes de spécification et de détection de manière automatique. Dynamics permet de détecter les défauts de code directement à partir de l'APK d'une application. Pour ce faire, nous spécifions les défauts de code selon des évènements et des propriétés, Dynamics instrumente l'application selon la spécification des défauts de code, exécute l'application pour produire une trace et détecte ensuite les défauts de code de l'application en fonction de cette trace. En conséquence, Dynamics est un outil qui permet de prendre en compte le comportement de l'application afin de garantir une détection plus précise des défauts de code comportementaux.

Troisièmement, nous validons Dynamics en utilisant la précision et le rappel sur un jeu de données en accès libre de 538 applications provenant de F-Droid. De plus, nous le comparons aux deux outils d'analyse statique, aDoctor et Paprika, pour la détection des défauts de code comportementaux issus de la littérature. Nos résultats montrent l'efficacité de notre méthode pour la détection des défauts de code comportementaux, en soulignant que de nombreux cas de défauts de code qui ne peuvent pas être détectés avec des approches statiques sont effectivement détectés avec Dynamics.

1.6 Plan de la thèse

Cette thèse se divise en cinq chapitres. Le chapitre 2 présente l'état de l'art en le regroupant selon différents axes de recherche. Le chapitre 3 donne la description des défauts de code comportementaux qui seront étudiés dans cette thèse. Le chapitre 4 présente la première contribution de cette thèse, une étude empirique sur la détection des défauts de code comportementaux avec les résultats obtenus et

les limitations identifiées. Le chapitre 5 présente la deuxième contribution majeure de cette thèse, un outil, DYNAMICS, avec la méthode associée et la validation de l'outil. Enfin, dans le chapitre 6 nous présentons la conclusion de notre thèse ainsi que les perspectives de recherche.

CHAPITRE II

ÉTAT DE L'ART

Dans ce chapitre, nous discutons des différents travaux de la littérature concernant les sujets de notre thèse. Plus particulièrement, nous présentons l'état de l'art sur 1) la détection des défauts de code spécifiques à Android, 2) l'importance et l'impact des défauts de code spécifiques à Android, 3) l'analyse dynamique utilisée dans les applications mobiles, 4) la détection des défauts de code par de l'analyse dynamique, 5) les méthodes formelles appliquées aux problèmes de code, et enfin 6) les méthodes formelles appliquées à Android.

2.1 Détection des défauts de code spécifiques à Android

La notion de défauts de code spécifiques à Android a été introduite par Reimann *et al.* (Reimann *et al.*, 2014). Ils proposent un catalogue de 30 défauts de code de qualité spécifiques à Android (quality code smells). Ces défauts de code sont définis principalement en fonction des bonnes et mauvaises pratiques documentées en ligne dans la documentation Android ou par des développeurs rapportant leur expérience sur des blogs. Ces défauts de code concernent différents aspects comme l'implémentation, les interfaces utilisateurs ou l'utilisation de la base de données. Ces défauts de code sont signalés comme ayant un impact négatif sur des propriétés telles que l'efficacité, l'expérience utilisateur ou la sécurité. Deux (DW et NLMR)

de nos défauts de code étudiés proviennent de ce catalogue.

Les défauts de code de sécurité (security code smells) (Ghafari *et al.*, 2017) constituent une autre catégorie de défauts axés sur les vulnérabilités des applications mobiles. Ghafari *et al.* (Ghafari *et al.*, 2017) identifient 28 défauts dont la présence peut indiquer un problème de sécurité dans une application mobile. Ces auteurs ont également développé un outil d'analyse statique pour étudier la prévalence des défauts de sécurité. Cependant, ces défauts de code ne sont pas comportementaux, car ils concernent la simple présence d'un attribut dans le manifeste ou la simple présence d'un appel de méthode dans le code indépendamment de tout comportement inapproprié induit dans le code.

Plusieurs outils sont également disponibles pour détecter les défauts de code Android, et Rasool *et al.* (Rasool et Ali, 2020) donnent un bon aperçu des outils existants qui peuvent identifier les défauts de code spécifiques à Android. Par exemple, Rasool *et al.* (Rasool et Ali, 2020) proposent leur propre approche qui est capable de détecter 25 défauts de code Android par l'analyse du code source et le traitement des métriques du code source. EARMO (Morales *et al.*, 2018) rapporte en outre une approche capable de détecter et de corriger les défauts de code liés à la consommation d'énergie dans les applications mobiles (energy code smells). Cette approche, lorsqu'elle est utilisée pour corriger ces défauts, est capable d'étendre considérablement l'autonomie de la batterie. La programmation génétique multi-objectifs a également été utilisée pour détecter les défauts d'Android (Kessentini et Ouni, 2017). Cette approche génère des règles, qui consistent en une combinaison de métriques de qualité avec des valeurs seuils pour détecter les défauts de code. Cette méthode prend donc en entrée un ensemble d'exemples de défauts de code spécifiques à Android et trouve le meilleur ensemble de règles pour couvrir la plupart des défauts de code Android attendus.

Cependant, sur les 19 outils différents rapportés par Rasool *et al.* (Rasool et Ali, 2020), si l'on retire ceux qui ne sont pas disponibles (prototypes, outils commerciaux ou privés), et ceux ne détectant aucuns défauts de code non comportementaux, tous ceux qui restent sont des extensions d'aDoctor (Palomba *et al.*, 2017) ou de Paprika (Hecht *et al.*, 2015b). Ces deux outils sont donc d'une importance capitale dans la détection des défauts de code spécifiques à Android.

aDoctor (Palomba *et al.*, 2017) est un outil de détection léger capable d'identifier 15 défauts de code spécifiques à Android. Paprika (Hecht *et al.*, 2015b) est également un outil de détection capable d'identifier 17 défauts de code Android. Les deux outils aDoctor et Paprika sont des outils d'analyse statique et tandis que aDoctor opère sur le code source, Paprika traite le bytecode. Iannone *et al.* (Iannone *et al.*, 2020) a proposé une nouvelle version de aDoctor, qui aide les développeurs à réusiner les défauts automatiquement. Cette version étendue est libre d'accès et disponible dans Android Studio sous la forme d'un module d'extension publié dans la boutique officielle. Sniffer (Habchi *et al.*, 2019b) est une boîte à outils libre d'accès qui permet de suivre l'historique complet des défauts de code spécifiques à Android. Cependant, Sniffer n'est pas totalement un nouvel outil de détection car il s'appuie sur Paprika pour détecter les défauts de code Android. Dans Sniffer, Paprika a néanmoins été étendu pour pouvoir analyser directement le code source au lieu du bytecode.

2.2 Importance et impact des défauts de code spécifiques à Android

De nombreuses études dans la littérature se concentrent sur l'impact que peuvent avoir les défauts de code spécifiques à Android sur différents aspects, comme la consommation énergétique ou la performance. Ces études sont primordiales pour démontrer l'intérêt dans la détection et la correction de ces défauts de code.

Il a été montré que les défauts de code spécifiques à Android avaient un impact sur la consommation énergétique (Carette *et al.*, 2017). Cette étude montre que la correction des défauts de code peut réduire la consommation d'énergie d'une application de manière significative. L'impact est plus ou moins significatif selon l'application et le scénario, mais dans tous les cas, corriger les défauts de code ne provoque pas d'effets négatifs et est donc recommandé.

Les défauts de code spécifiques à Android ont également un impact sur la performance des applications Android (Hecht *et al.*, 2016). En effet, selon plusieurs métriques comme la performance de l'interface utilisateur ou la mémoire, il a été montré dans une étude empirique que la correction de ces défauts de code améliorait les performances des applications. Il a également été montré que l'utilisation de Linters corrigeant des défauts de code permettait d'améliorer la qualité des applications mobiles (Habchi *et al.*, 2018). Un Linter est un outil d'analyse statique du code utilisé pour repérer les erreurs de programmation, les bogues et les erreurs stylistiques.

La qualité d'une application varie positivement comme négativement lors de son évolution (Hecht *et al.*, 2015a). Dans cette étude, le nombre de défauts de code orientés objet et spécifiques à Android a été retracé à travers les versions de certaines grandes applications. Il a été montré que lors de grosses mises à jour ajoutant de nombreuses classes, la qualité, ici mesurée par le nombre de défauts de code présents, diminuait fortement. D'un autre côté, la qualité s'améliore lors des versions de réusinage suivant les implémentations de ces mises à jour. La présence des défauts de code spécifiques à Android pendant l'évolution des applications a également été étudiée sur des centaines d'applications et des centaines de milliers de commits (Habchi *et al.*, 2019b). Il a été démontré que certains défauts de code restaient des années avant d'être détectés et corrigés. Les plus gros projets avec beaucoup de développeurs et d'itérations sont les plus enclins à détecter et

corriger ces défauts. La fréquence élevée de publication dans l'écosystème mobile ne favorise pas nécessairement une longue durée de vie des codes. Cette étude encourage également l'utilisation de Linters ou d'outils afin d'éliminer ces défauts de code spécifiques à Android. Une autre étude a été réalisée sur le même jeu de données que l'étude précédente afin de déterminer qui introduisait les défauts de code spécifiques à Android (Habchi *et al.*, 2019a). Plusieurs constats intéressants ressortent de cette étude. Tout d'abord, n'importe quel développeur semble introduire des défauts de code et pas uniquement un groupe isolé de développeurs. Ensuite, il n'y a pas de groupes distincts entre les développeurs qui introduisent des défauts de code et les développeurs qui corrigent ces défauts. Les développeurs qui introduisent et corrigent les défauts de code sont généralement les mêmes. Enfin, les nouveaux développeurs dans un projet n'introduisent ou ne corrigent pas plus de défauts de code que les autres développeurs de ce projet.

2.3 Analyse dynamique utilisée dans les applications mobiles

L'analyse dynamique des applications mobiles a été utilisée à de nombreuses reprises dans des travaux connexes. Habituellement, l'analyse dynamique est utilisée pour inspecter le comportement des applications malveillantes afin de détecter les logiciels malveillants. ANDLANTIS (Bierma *et al.*, 2014), un système hautement évolutif capable d'analyser dynamiquement 3 000 applications par heure, montre qu'il est possible d'envisager une analyse dynamique à grande échelle pour évaluer le comportement d'exécution et le trafic réseau.

De nombreux outils existent pour la détection d'outils malveillants, par exemple DROIDTRACE (Zheng *et al.*, 2014) est un outil permettant d'étudier le comportement malveillant des logiciels. Il utilise PTRACE (Process Trace), un appel système pour observer et contrôler l'exécution d'un autre processus. DROIDTRACE est

notamment capable d'effectuer une exécution pour provoquer différents comportements de chargement dynamique. DroidTrace partage certaines similitudes avec l'approche utilisée dans Dynamics dans la mesure où les deux outils tracent les appels qui se produisent pendant l'exécution. Cependant, DroidTrace est limité au chargement dynamique des bibliothèques, alors que nous considérons tout appel pouvant présenter un intérêt pour le défaut de code considéré.

TwinDroid (Razagallah *et al.*, 2022) est un jeu de données récent de traces d'appels système d'applications Android. Ce jeu de données se base sur des applications Android bénignes et infectées. De plus, TwinDroid permet la génération de traces entièrement automatisée, ce qui permet aux utilisateurs de générer de nouvelles traces de manière standardisée. Ce jeu de données a été utilisé pour classifier les applications malveillantes (Razgallah et Khoury, 2021) selon un ensemble de techniques de détection de logiciels malveillants, principalement de la détection dynamique basée sur des modèles d'apprentissage automatique.

Il existe encore plein d'autres outils utilisant les techniques d'analyse dynamiques dans la détection d'applications Android malveillantes, par exemple AndroShield (Amin *et al.*, 2019) et DATDroid (Thangaveloo *et al.*, 2020). D'un côté, l'outil DATDroid se base sur de l'apprentissage automatique en effectuant de l'extraction de caractéristiques pour classifier les applications si elles sont infectées ou bénignes. De l'autre côté, AndroShield utilise de l'analyse statique et dynamique pour détecter des vulnérabilités et propose une infrastructure logicielle avec une interface web.

L'analyse dynamique nécessite d'exécuter l'application. Pour ce faire, des générateurs d'entrées ont vu le jour. Ces générateurs d'entrées permettent de générer des actions utilisateurs comme un clic, saisir un texte, appuyer sur un bouton afin de simuler le comportement d'un utilisateur. Android fournit de base dans son

SDK un générateur d'entrées purement aléatoire, MonkeyRunner [1]. De nombreux outils se basent sur des versions étendues de MonkeyRunner utilisant des techniques plus avancées, comme Droidbot (Li *et al.*, 2017) qui utilise une stratégie d'exploration basée sur un modèle dans le cadre d'une approche boite noire, et DroidbotX (Yasin *et al.*, 2021) qui utilise une stratégie d'exploration basée sur de l'intelligence artificielle, spécifiquement du Q-Learning. Dynodroid (Machiry *et al.*, 2013) est un autre générateur d'entrée qui utilise une stratégie dite "observation-sélection-exécution" qui va calculer à chaque étape de l'exécution quels sont les évènements pertinents, les sélectionner selon un ensemble de métriques puis va les exécuter selon l'évènement qui semble le plus pertinent. Enfin, Humanoid (Li *et al.*, 2019) est un générateur d'entrée utilisant une stratégie d'exploration basée sur de l'apprentissage automatique, spécifiquement de l'apprentissage en profondeur.

2.4 Détection des défauts de code par de l'analyse dynamique

Bien qu'il n'existe pas, à notre connaissance, de littérature sur la détection dynamique des défauts de code spécifiques à Android, il existe une certaine littérature sur la détection des défauts de code utilisant de l'analyse dynamique. Par exemple, JSNose (Fard et Mesbah, 2013) détecte les défauts de code JavaScript en utilisant l'analyse dynamique. Cependant, la détection des défauts de code se fait en fonction des valeurs des métriques logicielles et les aspects dynamiques ne concernent pas le comportement de l'application, mais plutôt le traitement des métriques et la couverture du code. Le défaut de code "Feature Envy", un défaut de code orienté objet, a également été détecté avec une approche dynamique (Kumar et Chhabra, 2014). Comme dans Dynamics, le code Java est instrumenté et

1. `https://developer.android.com/studio/test/monkeyrunner`

le comportement du programme est ensuite analysé pendant l'exécution. L'article mentionne également que l'analyse dynamique pourrait être avantageuse pour la détection d'autres types de défauts de code, cependant, la détection n'est effectuée que sur un seul défaut de code orienté objet. Il y a une autre différence avec notre approche, cet article utilise la programmation orientée aspect pour instrumenter le code source, alors que notre approche instrumente le bytecode (APK) et ne nécessite pas le code source de l'application mobile.

Une étude des outils de détection des défauts de code de test de la littérature (Aljedaani *et al.*, 2021) a identifié 22 outils. Les défauts de code de test sont définis comme des choix de conception sous-optimaux que les développeurs font lors de l'implémentation des cas de test. Parmi ces 22 outils, 10 utilisent des méthodes dynamiques dénommées le "dynamic tainting". Le "dynamic tainting" surveille le code source pendant son exécution. Il permet d'analyser les données réelles du code sur la base des informations d'exécution. Plus spécifiquement, il fonctionne en deux étapes : (1) exécuter le code source avec une valeur ou une marque "taint" prédéfinie (c'est-à-dire l'entrée de l'utilisateur), et (2) déterminer quelles exécutions sont affectées par cette valeur ou cette marque.

2.5 Méthodes formelles appliquées aux problèmes de code

Des vérificateurs de modèles (model checkers en anglais) comme BLAST (Beyer *et al.*, 2007) et MOPS (Chen et Wagner, 2002) visent également à analyser des problèmes de code. Ils vérifient si des propriétés temporelles de système C ont été violées en utilisant des techniques de vérification de modèles. Les propriétés temporelles sont en LTL et chaque vérificateur de modèles modélise le code à l'aide d'automates. Les vérificateurs de modèles vont soit vérifier formellement que le programme est correct, soit retourner un contre-exemple de comportement pos-

sible qui viole les propriétés. PATHFINDER (Havelund et Pressburger, 2000) est un outil permettant de traduire du Java vers du PROMELA, un langage de modélisation du vérificateur de modèle SPIN. PATHFINDER se concentre principalement sur les interblocages, mais peut également vérifier les violations de propriétés diverses. Comme PATHFINDER s'intéresse au Java, il est utilisé par la suite pour effectuer de la vérification de modèles liée à Android. Bandera (Corbett *et al.*, 2000) est une collection de programmes d'analyse et de transformation permettant d'extraire un modèle depuis du code source Java afin d'effectuer de la vérification de modèles pour le vérifier. L'objectif est de résoudre la difficulté de l'extraction de modèles de systèmes compliqués pour obtenir un système à états finis. JAVAMOP (Jin *et al.*, 2012) et JAVA PATHEXPLORER (JPAX) (Havelund et Rosu, 2001) sont des systèmes de surveillance utilisés pour effectuer de la vérification à la volée, paramétrique pour JAVAMOP. La vérification à la volée est une approche d'analyse formelle permettant de détecter et réagir à des comportements respectant ou violant certaines propriétés sur l'exécution d'un système. Ces deux systèmes permettent de vérifier des propriétés à tout instant du logiciel, du test au débogage, en surveillant aussi bien des objets que des propriétés globales. À noter que les deux outils se concentrent sur le respect d'une spécification formelle à l'aide de la LTL.

2.6 Méthodes formelles appliquées à Android

Les méthodes formelles dans l'écosystème Android s'intéressent principalement à la sécurité, et plus spécifiquement à trois aspects particulièrement importants qui ont été introduits avec l'apparition des applications mobiles : les permissions, la vie privée et les applications malveillantes. Ces trois aspects sont liés les uns aux autres : en général, les permissions sont utilisées pour divulguer la vie privée et les applications malveillantes utilisent quant à elles les permissions de manière mali-

cieuse. La vérification à la volée et la vérification de modèles prenant en compte le comportement de l'application, ils sont très adaptés à ce type d'application. Chaque application Android définit un ensemble de permissions dans son manifeste que l'utilisateur doit normalement valider afin d'utiliser l'application. Les méthodes formelles se concentrent donc sur l'utilisation de permissions sensibles non autorisées, comme l'envoi furtif de SMS, la suppression de contacts, le blocage de SMS, etc. La vie privée (privacy) est très importante dans les applications mobiles, puisque beaucoup de données sont présentes dans les téléphones : GPS, contacts, photos, IMEI (International Mobile Equipment Identity). La divulgation de ces données sans l'accord de l'utilisateur est un problème.

(Bai *et al.*, 2018) et (Chen *et al.*, 2013) utilisent des méthodes de vérification de modèles afin de détecter l'utilisation de permissions sensibles ainsi que l'utilisation de données privées sensibles et retourner une exécution possible menant à ces utilisations. (Armando *et al.*, 2012) fournissent une modélisation et vérification formelle pour les permissions dangereuses à haute sécurité. (Shin *et al.*, 2009) et (Shin *et al.*, 2010) soumettent une analyse formelle des permissions dans Android. Le premier procède au moyen d'une vérification de théorèmes de sécurité à l'aide d'assistant de preuve. Le deuxième propose un modèle formel du schéma des permissions en vérifiant les autorisations et les interactions entre ces permissions. (Gunadi et Tiu, 2013) utilisent des méthodes de vérification à la volée pour vérifier ces permissions. (Lu et Mukhopadhyay, 2012) se basent sur des solveurs SMT afin d'aider les développeurs à détecter les erreurs de programmations et les violations de permissions statiquement.

(Song et Touili, 2014) et (Battista *et al.*, 2016) traitent l'identification et la détection de logiciels malveillants à l'aide de la vérification de modèles, en modélisant le comportement de l'application à l'aide d'automates. Ils spécifient des propriétés propres aux applications malicieuses à l'aide de la logique temporelle et tentent

de les repérer en appliquant la vérification de modèles.

Les techniques formelles habituelles ont également été appliquées pour la consommation énergétique des applications, et vérifier qu'il n'y a pas de défauts de conception dans l'application qui entraînerait une surconsommation d'énergie. La vérification à la volée est utilisée dans (Espada *et al.*, 2015) afin de vérifier des propriétés énergétiques. Ils utilisent des modèles pour conduire de la génération et l'exécution de suite de tests : l'objectif est de décrire le profil énergétique des actions du téléphone (Réseau, GPS...). Ils apportent également trois méthodes importantes : une méthode pour décrire les propriétés de la trace d'exécution en termes d'énergie, une méthode pour surveiller la trace d'exécution dans le téléphone et une méthode pour vérifier les propriétés énergétiques. La vérification de modèles a également été utilisée dans (Nakajima, 2015), se basant sur le modèle de consommation d'énergie présentée dans (Nakajima, 2013). La vérification de modèles est utilisée afin de vérifier si des propriétés, spécifiées à base d'"ebugs" (Pathak *et al.*, 2011), ont été violées. Les "ebugs" sont des défauts de code qui provoquent de la consommation d'énergie superflue. De tels "ebugs" sont donc nommés différemment, mais se rapprochent dans le principe des défauts de code décrits dans ce document.

Les méthodes formelles sont majoritairement utilisées pour la sécurité et l'énergie, mais il y a également beaucoup d'utilisations diverses. (Jing *et al.*, 2012) génèrent des suites de tests afin de vérifier que l'application mobile est conforme à une spécification formelle, de manière comportementale, à partir de modèles. De même, (Bauer *et al.*, 2012) utilisent la vérification à la volée pour vérifier également si l'application est conforme à une spécification formelle. (Falcone *et al.*, 2012) s'intéressent à la vérification à la volée pour les applications mobiles ainsi qu'à l'application de règles à la volée (runtime enforcement en anglais), où l'exécution est modifiée en temps réel pour s'adapter aux propriétés désirées. Il est utilisé pour

vérifier la bonne utilisation de structures Java. TaintDroid (Enck *et al.*, 2014) est souvent utilisé dans les précédents articles. Il s'agit d'un système d'analyse et de suivi en temps réel d'applications sur Android afin d'être utilisé par d'autres méthodes afin de surveiller le comportement.

2.7 Conclusion

De nombreuses recherches ont porté sur la détection des défauts de code dans les applications mobiles et sur l'analyse dynamique des applications mobiles, ce qui démontre l'intérêt de la communauté pour ces sujets. En revanche, peu de travaux existent sur la détection des défauts de code utilisant de l'analyse dynamique, ce qui montre qu'il y a encore beaucoup de questions de recherche ouvertes. Les approches formelles, comme la vérification à la volée, sont également fortement utilisées dans le contexte Android, plus particulièrement pour prendre en compte le comportement de l'application. Cependant, à notre connaissance, il n'existe pas de travaux basés sur des approches formelles sur la détection de défauts de code. Il y a donc un manque d'approches et d'outils pour détecter les défauts de code, plus spécifiquement les défauts de code comportementaux, en utilisant l'analyse dynamique et les approches formelles dans les applications mobiles. Ceci démontre donc l'intérêt des contributions de notre thèse à l'état de l'art.

CHAPITRE III

DÉFAUTS DE CODE COMPORTEMENTAUX

Dans ce chapitre, nous donnons la description des défauts de code comportementaux qui seront concernés par l'étude empirique et par notre approche outillée DYNAMICS. Pour rappel, nous définissons un défaut de code comportemental comme des caractéristiques dans le code source induisant un comportement inapproprié du code pendant l'exécution qui peut avoir un impact négatif sur la qualité du logiciel en termes de performance, de consommation d'énergie ou de mémoire. Le terme "comportement" fait spécifiquement référence au comportement d'exécution du code, c'est-à-dire à une occurrence ou à une séquence d'événements ou d'actions observables dans le code pendant son exécution. Ces défauts de code comportementaux sont liés à diverses préoccupations importantes des applications mobiles, telles que la consommation d'énergie, la mémoire et les performances. Même si d'autres défauts de code répondent à ces mêmes préoccupations, ils ne sont pas nécessairement comportementaux. C'est le cas, par exemple, du défaut de code IDS (Inefficient Data Structure) d'Android (Palomba *et al.*, 2017). En effet, la définition du défaut de code IDS est la suivante : "L'association d'un entier vers un objet par l'utilisation d'une *HashMap<Integer,Object>* est lent et devrait être remplacé par d'autres structures de données efficaces, telles que le *SparseArray* (Reimann *et al.*, 2014)". Il ne s'agit pas spécifiquement d'une caractéristique du code *induisant* un comportement inapproprié du code pendant l'exécution, mais plutôt

d'une caractéristique du code (l'utilisation d'un *HashMap<Integer,Object>*) qui *a un impact négatif* sur la qualité logicielle de l'application (baisse des performances de l'application). Il ne s'agit donc pas d'un défaut de code comportemental.

De plus, nous relions les sept défauts de code comportementaux étudiés à trois catégories de défauts de code comportementaux différentes. Nous présentons maintenant ces sept défauts de code définis selon les définitions suivantes (Hecht, 2016), (Hecht *et al.*, 2015b), (Reimann *et al.*, 2014) et (Palomba *et al.*, 2017). Nous fournissons pour chaque défaut de code comportemental le comportement inapproprié associé et les caractéristiques dans le code source associées.

3.1 Défauts caractérisés par l'utilisation d'un appel de méthode ou d'une séquence d'appels de méthode pendant l'exécution

La première catégorie de défauts de code comportementaux est caractérisée par l'utilisation d'un appel de méthode ou d'une séquence d'appels de méthode pendant l'exécution. Plus précisément, cette catégorie concerne les défauts de code qui décrivent l'appel d'une méthode spécifique selon certaines conditions ou qui décrivent une séquence d'appels de méthode selon un ordre spécifique.

3.1.1 Durable Wakelock (DW)

Description : Un *Wakelock* est le mécanisme permettant à une application de garder l'appareil en marche afin d'accomplir une tâche. Toutefois, lorsque cette tâche est terminée, le verrou doit être libéré afin de réduire la consommation de la batterie (Reimann *et al.*, 2014). Dans Android, la classe *PowerManager.WakeLock* est chargée de définir les méthodes d'acquisition et de libération du verrou. Si une méthode utilisant une instance de la classe *WakeLock* acquiert le verrou sans appeler la libération, un défaut est donc identifié.

Comportement inapproprié : Un appel à la méthode *acquire* n'est pas suivi de l'appel de la méthode *release*.
Caractéristiques dans le code source : Utilisation des méthodes *acquire* et *release* de la classe *WakeLock*.

3.2 Défauts caractérisés par des problèmes pendant l'exécution d'une méthode

La deuxième catégorie contient les défauts de code comportementaux caractérisés par des problèmes pendant l'exécution d'une méthode, comme le temps d'exécution d'une méthode trop long ou une utilisation excessive de la mémoire. Plus précisément, les défauts de code appartenant à cette catégorie décrivent des comportements au sein des méthodes importantes dans les activités Android et dans le cycle de vie Android.

3.2.1 Heavy AsyncTask (HAS)

Description : Dans Android, l'API *AsyncTask* permet aux développeurs d'effectuer de courtes opérations en arrière-plan. Toutefois, trois des quatre étapes de *AsyncTask* sont exécutées sur le processus principal de l'interface utilisateur et non en arrière-plan. Ainsi, ces étapes ne doivent pas être des opérations longues ou bloquantes pour éviter : 1) que l'interface graphique ne réponde plus aux interactions de l'utilisateur ou 2) que la boîte de dialogue ANR (*Android ne répond plus*) ne s'affiche. Ainsi, une classe étendant *AsyncTask* ne doit jamais contenir de méthodes *onPostExecute*, *onPreExecute* ou *onProgressUpdate* chronophages ou bloquantes (Mariotti, 2013c).
Comportement inapproprié : Les méthodes *onPostExecute* / *onPreExecute* / *onProgressUpdate* sont chronophages ou bloquantes.

Caractéristiques dans le code source : La mise en œuvre des méthodes *onPostExecute* / *onPreExecute* / *onProgressUpdate* dans une classe *AsyncTask*.

3.2.2 Heavy BroadcastReceiver (HBR)

Description : Les applications Android peuvent utiliser un récepteur de diffusion pour gérer les communications de diffusion avec le système ou d'autres applications. Toutefois, la méthode *onReceive* de *BroadcastReceiver* s'exécute dans le fil d'exécution principal de l'interface utilisateur et peut provoquer le gel de l'application ou l'affichage d'une boîte de dialogue ANR (Mariotti, 2013a). Par conséquent, la méthode *onReceive* ne doit jamais contenir d'opérations chronophages ou bloquantes.

Comportement inapproprié : La méthode *onReceive* est chronophage ou bloquante.

Caractéristiques dans le code source : L'implémentation de la méthode *onReceive* dans une classe *BroadcastReceiver*.

3.2.3 Heavy Service Start (HSS)

Description : Les services sous Android peuvent effectuer des opérations lourdes en arrière-plan. Cependant, les services Android s'exécutent dans le fil d'exécution principal de leur processus hôte. Par défaut, l'exécution du service commence par un appel à la *OnStartCommand* du service, qui est exécuté dans le fil d'exécution principal de l'interface utilisateur. Lorsque le service exécute des opérations longues ou asynchrones, un nouveau fil d'exécution doit être créé par la méthode *OnStartCommand* pour traiter ces opérations en dehors du fil d'exécution princi-

pal de l'interface utilisateur, sans quoi l'application risque de se figer ou d'afficher une boîte de dialogue ANR (Mariotti, 2013b). Ainsi, la méthode *OnStartCommand* ne doit jamais contenir d'opérations chronophages ou bloquantes.

Comportement inapproprié : La méthode *onStartCommand* est chronophage ou bloquante.

Caractéristiques dans le code source : L'implémentation de la méthode *onStartCommand* dans une classe *Service*.

3.2.4 Init OnDraw (IOD)

Description : Les routines *OnDraw* sont responsables de la mise à jour de l'interface graphique d'une application Android. Ces routines sont invoquées chaque fois que l'interface graphique est rafraîchie (jusqu'à 60 fois par seconde). Ainsi, tout travail de calcul supplémentaire effectué dans *OnDraw* est amplifié par cette fréquence. De plus, un taux élevé d'allocations de mémoire peut entraîner une forte consommation de mémoire et de nombreux appels aux activités du ramasse-miettes (Ni-Lewis, 2015). Par conséquent, les routines *OnDraw* ne devraient jamais contenir d'instructions *init* pour allouer de la mémoire (soit par un *new* ou par l'appel à une fabrique/constructeur) ou être une méthode excessivement longue.

Nous divergeons légèrement des définitions (Hecht *et al.*, 2015b; Hecht, 2016) en ce qui concerne le défaut de code IOD. En effet, dans ces articles, le défaut de code est détecté en vérifiant que chaque méthode *onDraw* n'a pas d'allocations de mémoire. Cependant, la discussion autour du défaut de code IOD montre clairement que l'intention est de ne pas avoir de travail de calcul supplémentaire effectué dans *onDraw*, donc la méthode ne doit pas être chronophage comme nous

le faisons dans la définition ci-dessus.

Comportement inapproprié : La méthode *onDraw* est chronophage ou initialise des objets.

Caractéristiques dans le code source : L'implémentation de la méthode *onDraw* dans une classe *View*.

3.2.5 No Low Memory Resolver (NLMR)

Description : Lorsque la mémoire du système Android est insuffisante, le système appelle la méthode *onLowMemory* de chaque activité en cours. Cette méthode est chargée de réduire l'utilisation de la mémoire de l'activité. Si cette méthode n'est pas implémentée par l'activité, le système Android termine automatiquement le processus de l'activité pour libérer de la mémoire. Cela peut entraîner la fin inattendue du programme. En outre, lorsque la méthode *onLowMemory* est déclarée, elle doit contenir une action de récupération de la mémoire. Par conséquent, une classe *Activity* qui ne définit pas *onLowMemory* ou qui définit cette méthode, mais dans laquelle *onLowMemory* n'effectue aucune action de récupération de mémoire, est considérée comme un défaut de code.

Nous divergeons légèrement des définitions (Hecht *et al.*, 2015b; Hecht, 2016; Palomba *et al.*, 2017) en ce qui concerne le défaut de code NLMR. En effet, dans ces articles, le défaut de code est détecté en vérifiant que chaque activité implémente la méthode *onLowMemory*, que cette méthode contienne ou non une instruction. Cependant, la discussion autour du défaut de code NLMR montre clairement que l'intention est de récupérer de la mémoire comme nous le faisons dans la définition ci-dessus.

Le défaut de code NLMR fait référence à la méthode *onLowMemory*, qui est désor-

mais dépréciée et a été remplacée par la méthode *onTrimMemory*[1]. Nous l'avons néanmoins incluse dans cette étude, car même si la méthode *onLowMemory* est dépréciée, elle est toujours largement utilisée. En effet, nous montrerons dans la section 4.5.2 que de nombreuses instances NLMR apparaissent dans notre jeu de données. De plus, le remplacement d'une méthode par une autre ne change pas le principe de détection et notre méthode de détection sera toujours applicable lorsque la nouvelle méthode recommandée sera d'usage courant.

Comportement inapproprié : La méthode *onLowMemory* ne récupère pas de mémoire lors de son exécution.

Caractéristiques dans le code source : L'implémentation de la méthode *onLowMemory* dans une classe *Activity*.

3.3 Défauts caractérisés par des variations de données indésirables pendant l'exécution

La troisième catégorie contient les défauts de code comportementaux caractérisés par des variations de données indésirables pendant l'exécution. Plus précisément, les défauts de code appartenant à cette catégorie décrivent des attributs ou des variables qui ne doivent pas atteindre des tailles supérieures à une valeur spécifique ou bien être égaux à une valeur spécifique pendant l'exécution selon certaines conditions.

---

1. `https://developer.android.com/reference/android/content/ComponentCallbacks\#onLowMemory()`

3.3.1 HashMap Usage (HMU)

Description : Le framework Android fournit *ArrayMap* et *SimpleArrayMap* en remplacement de la traditionnelle *HashMap* de Java. Ces structures sont considérées comme étant plus efficaces en termes de mémoire et déclenchant moins le ramasse-miettes, sans différence significative sur les performances des opérations pour les *Map* contenant jusqu'à des centaines de valeurs [2] (Haase, 2015). Ainsi, à moins qu'une *Map* complexe pour un grand nombre d'objets soit nécessaire, l'utilisation de *ArrayMap* doit être préférée à celle de *HashMap* dans les applications Android. Par conséquent, la création de petites instances de *HashMap* ou de grandes instances de *SimpleArrayMap/ArrayMap* est considérée comme un défaut de code.

Comportement inapproprié : Une structure *HashMap* est utilisée pour un petit ensemble d'objets ou les structures *ArrayMap* / *SimpleArrayMap* sont utilisées pour un grand ensemble d'objets.

Caractéristiques dans le code source : L'utilisation des structures *HashMap* / *ArrayMap* / *SimpleArrayMap*.

---

2. `https://developer.android.com/reference/android/support/v4/util/ArrayMap.html`

CHAPITRE IV

ÉTUDE EMPIRIQUE

Dans ce chapitre, nous étudions l'efficacité de la détection des défauts de code comportementaux dans la pratique ainsi que l'identification des limites des techniques de détection actuelles des outils. À cette fin, nous menons une étude empirique comparant les définitions textuelles des défauts de code comportementaux telles que décrites dans la littérature avec leurs techniques de détection dans les outils étudiés. L'étude empirique consiste également à étudier les problèmes des règles de détection définies dans les outils afin de vérifier si les résultats de détection sont partiels ou erronés. Concrètement, nous avons analysé 676 instances de sept défauts de code comportementaux détectés dans 318 applications, manuellement, avec les outils concernés.

Il s'agit de la première étude empirique de ce type à étudier l'efficacité et la cohérence de la détection des défauts de code comportementaux dans la pratique, ainsi qu'à identifier les limites des techniques de détection actuelles.

En résumé, notre première contribution dans la section 4.5 consiste à fournir une étude empirique qui compare les résultats des outils de détection aux définitions textuelles de sept défauts de code comportementaux décrits dans la littérature. Nous déterminons également dans cette étude empirique les outils existants représentatifs de la détection des défauts de code spécifiques à Android. Enfin, notre

deuxième contribution dans la section 4.7 est une analyse approfondie des techniques de détection des outils afin d'identifier leurs limites et de partager les leçons tirées de notre étude empirique.

4.1 Questions de recherche

L'étude empirique vise à répondre aux deux questions de recherche suivantes :

- ***QR*$_1$: Est-ce que les outils étudiés sont efficaces pour détecter les défauts de code comportementaux ?** Nous tentons d'identifier dans quelle mesure les outils renvoient des *faux positifs* et des *faux négatifs*. C'est l'une des principales motivations de cette étude. Certaines instances définies comme ayant un défaut de code n'ont finalement pas de défaut de code et sont donc des *faux positifs*. Un trop grand nombre de faux positifs nuit à l'efficacité des outils de détection, puisque les résultats sont erronés. Parallèlement, certaines instances définies comme n'ayant pas de défauts de code ont finalement un défaut de code et sont donc des *faux négatifs*. Le fait d'avoir trop de faux négatifs a un impact sur l'efficacité des outils, puisque les résultats de la détection sont partiels.
- ***QR*$_2$: Les défauts de code comportementaux détectés par les outils étudiés sont-ils cohérents avec leur définition littérale originale ?** Cette question vise à vérifier si les règles de détection spécifient bien ou non la définition de la littérature des défauts de code comportementaux. Les défauts de code détectés sont incohérents par rapport à leur définition littérale si des éléments de la définition sont absents des règles de détection de l'outil. Cela suppose qu'il existe des définitions communes pour ces défauts de code. C'est en effet le cas des défauts de code comportementaux considérés dans cette étude, pour lesquels (Hecht *et al.*,

2015b; Hecht, 2016; Palomba *et al.*, 2017) donnent tous les mêmes définitions textuelles qui ne varient que dans l'utilisation de certains synonymes. Nous avons donc suivi ces définitions à la seule exception des défauts de code NLMR et IOD, comme nous l'expliquons dans le chapitre 3.

Pour répondre à nos questions de recherche, nous voulons rejeter les hypothèses nulles formulées comme suit :

- H_1^{posi} : Il n'y a pas de faux positifs parmi les défauts de code détectés renvoyés par les outils.
- H_2^{nega} : Il n'y a pas de faux négatifs parmi les défauts de code détectés renvoyés par les outils.
- H_3^{regle} : Il n'y a aucune différence entre les règles de détection et la définition littérale des défauts de code.

4.2 Sujets de l'étude

4.2.1 Outils

L'étude est basée sur deux outils, Paprika et aDoctor. Ces deux outils sont faciles à utiliser et libres d'accès. Les deux outils utilisent des techniques de détection statique et sont entièrement automatiques.

Nous avons utilisé une procédure systématique pour collecter et sélectionner les outils de détection des défauts de code spécifiques à Android à prendre en compte dans cette étude, et nous nous sommes limités à aDoctor (Palomba *et al.*, 2017) et Paprika. (Hecht *et al.*, 2015b). Nous avons commencé par la liste des outils de détection des défauts de code spécifiques à Android fournie par Rasool *et al.* (Rasool et Ali, 2020). À notre connaissance, cette liste est la plus complète et la plus à jour des outils de détection des défauts de code spécifiques à Android.

En effet, nous n'avons pas trouvé d'autres travaux qui rapportent de telles listes.

Rasool *et al.* rapportent 25 publications sur la détection des défauts de code spécifiques à Android et donne dans chaque cas la technique utilisée pour la détection (basée sur la recherche, basée sur la métrique et basée sur les symptômes), l'outil qui met en oeuvre la technique et la disponibilité de l'outil (oui ou non). Cette liste comprend 19 outils différents pour les 25 publications. Parmi ces 19 outils, 5/19 sont des prototypes et ne sont pas disponibles (Paternò *et al.*, 2017; Kessentini et Ouni, 2017; Morales *et al.*, 2018; Rubin *et al.*, 2019; Ibrahim *et al.*, 2020). Parmi les 14/19 outils restants, 4/14 sont commercialisés ou privés et ne sont pas disponibles (Banerjee *et al.*, 2014; Lin *et al.*, 2015; Mannan *et al.*, 2016; Elsayed *et al.*, 2019). Des 10/14 outils restants, 4/10 sont des extensions de aDoctor (Almalki, 2018; Iannone *et al.*, 2020) et Paprika (Habchi *et al.*, 2017; Habchi *et al.*, 2019b). Parmi les 6/10 outils restants, 4/6 traitent des catégories de défauts de code (défauts de code liés aux permissions (Dennis *et al.*, 2017), défauts de code liés à la sécurité (Gadient *et al.*, 2018), défauts de code liés aux tests unitaires (Peruma, 2018) et défauts de code orientés objets (Lim, 2018)) qui ne sont pas comportementaux, contrairement à l'objet de cette étude. Les 2/6 derniers outils restants sont aDoctor (Palomba *et al.*, 2017) et Paprika (Hecht *et al.*, 2015b), et nous avons donc choisi de nous concentrer sur ces deux outils.

aDoctor et Paprika sont donc des outils disponibles et accessibles, détectant certains défauts de code comportementaux, et ont plusieurs outils qui en dérivent. Ils sont donc représentatifs des outils de détection des défauts de code spécifiques à Android de l'état de l'art.

Une différence majeure entre ces outils est que l'outil Paprika est beaucoup plus adapté à l'analyse à grande échelle. Tout d'abord, en termes de temps d'analyse, Paprika est beaucoup plus rapide, et particulièrement utile lorsque nous

souhaitons analyser un grand nombre d'applications. Tandis que Paprika avec l'extension Sniffer n'attend qu'une liste de dossiers de sources de code, aDoctor doit se lancer pour chaque application et pour chaque défaut de code.

Bien que les deux outils soient statiques dans la détection, aDoctor fait simplement appel à des expressions régulières et à la recherche de chaînes de caractères. Paprika, en revanche, utilise des requêtes sur les bases de données orientées graphe, et donc sur la structure des applications et des classes, ce qui permet plus d'options dans les règles de détection. Cette différence se ressent sur les résultats de aDoctor, beaucoup plus importants en nombre d'applications et en nombre d'instances affectées par les défauts de code.

Paprika utilise des métriques logicielles, telles que le nombre d'instructions et la complexité cyclomatique, pour la détection des défauts de code. aDoctor, qui est décrit comme un outil léger, ne prend en compte aucun de ces aspects. Paprika utilise ces métriques logicielles principalement pour la détection des défauts orientés objets, mais aussi pour les défauts de code spécifiques à Android, tels que HAS, HBR et HSS, qui sont décrits plus loin dans le document.

aDoctor utilise des expressions régulières ou des chaînes de caractères spécifiques dans le code selon des règles de détection prédéfinies pour déterminer si un défaut de code est présent ou non. Il analyse le code source Java en entrée et produit un tableau de résultats pour chaque défaut de code. Paprika, quant à lui, analyse le bytecode de l'application et l'analyse pour remplir une base de données orientée graphe interne. Les nœuds de ce graphe représentent des méthodes, des classes et des attributs. Les défauts de code sont détectés grâce aux requêtes exécutées dans cette base de données.

4.2.2 Défauts de code

Les défauts de code considérés dans cette étude sont les défauts de code comportementaux détaillés dans le chapitre 3.

Dans cette étude, parmi les 11 défauts de code comportementaux détectés par Paprika et aDoctor (voir le tableau 4.1), nous considérons les sept défauts de code comportementaux suivants : DW, HAS, HBR, HMU, HSS, IOD, NLMR. Chacun d'entre eux est traité par un seul outil, à l'exception de NLMR, qui est traité à la fois par Paprika et aDoctor. Cependant, nous excluons les quatre défauts de code comportementaux : DTWC, LT, UC et UIO. Le défaut de code LT est déprécié en raison de la dépréciation de la méthode *Thread.stop()*[1], et sa dépréciation depuis trop d'années empêchait l'apparition de cette méthode dans nos jeux de données. Les trois autres défauts de code, DTWC, UC et UIO, ont très peu d'occurrences lors d'études des jeux de données préliminaires, puisqu'elles apparaissent dans des contextes plus rares et leurs quelques occurrences ne permettent pas d'étudier suffisamment de cas pertinents.

4.3 Objets de l'étude

Les applications utilisées dans l'étude proviennent du jeu de données publié par Habchi *et al.* (Habchi *et al.*, 2019b), où le jeu de données a été utilisé pour analyser et détecter les défauts de code à partir des commits de projets GitHub. Ce jeu de données est constitué de véritables applications libres d'accès de F-Droid publiées sur GitHub. F-Droid fournit un jeu de données d'applications réelles qui ne sont ni des applications factices, ni des modèles, ni des bibliothèques. Nous étudions

1. `https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`

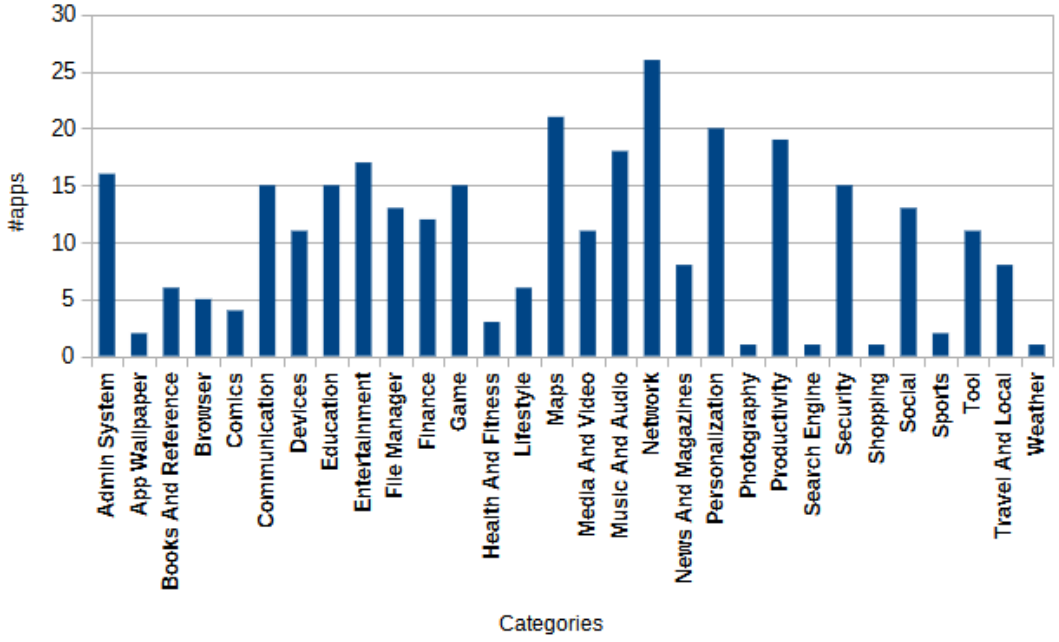
Tableau 4.1 Résumé des défauts de code comportementaux détectés par Paprika et aDoctor.

Défaut de code	Est comportemental	Étudié	Outil
DR			aDoctor
DTWC	✓		aDoctor
DW	✓	✓	aDoctor
HAS	✓	✓	Paprika
HBR	✓	✓	Paprika
HMU	✓	✓	Paprika
HSS	✓	✓	Paprika
IDFP			aDoctor
IDS			aDoctor
IGS			aDoctor & Paprika
IOD	✓	✓	Paprika
ISQLQ			aDoctor
IWR			Paprika
LIC			aDoctor & Paprika
LT	✓		aDoctor
MIM			aDoctor & Paprika
NLMR	✓	✓	aDoctor & Paprika
PD			aDoctor
RAM			aDoctor
SL			aDoctor
UC	✓		aDoctor
UCS			Paprika
UHA			Paprika
UIO	✓		Paprika

318 des 324 applications publiées à l'origine dans le jeu de données de Habchi *et al.* Pour les six cas restants, quatre applications ne sont plus disponibles, leurs dépôts sont devenus privés ou supprimés, et deux d'entre elles ne peuvent pas être traitées par PAPRIKA. Les raisons ne sont pas connues ou détaillées, aucune erreur n'étant affichées, cela peut être dû au fait que leurs dépôts ont évolué en utilisant des fichiers/classes non supportés par PAPRIKA.

Nous présentons le jeu de données des applications sous différents angles en termes de catégorie, de taille, d'intérêt et de contributions pour illustrer sa diversité et sa représentativité. La liste des applications et les mesures associées aux applications utilisées pour l'étude sont publiées [2] à des fins d'évaluation et de réplication.

FIGURE 4.1 Distribution des applications par rapport à leur catégorie.

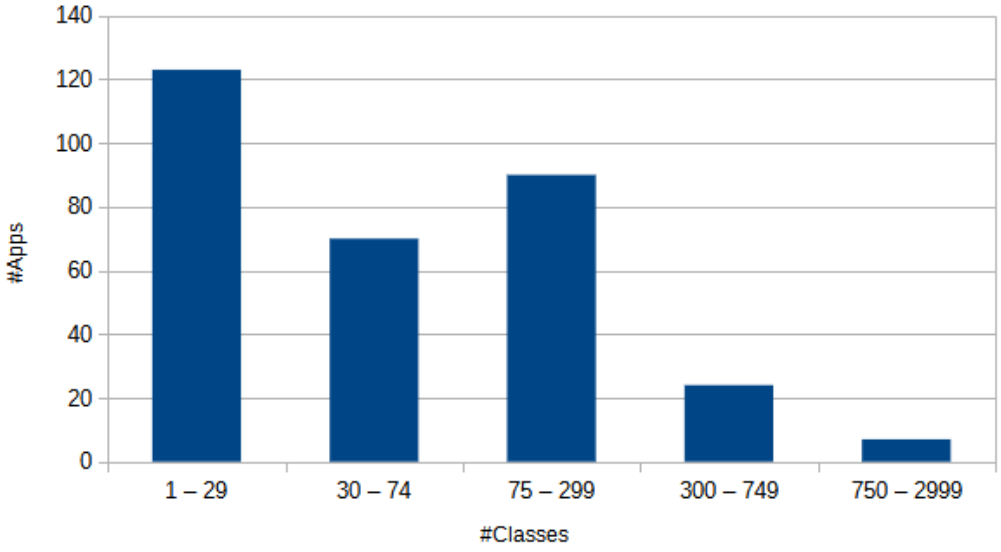


1) *Catégorie :* Nous classons les applications du jeu de données en fonction de la catégorie à laquelle elles appartiennent. Nous déterminons les catégories en ana-

2. `https://figshare.com/s/8235a0575ff4a88f0deb`

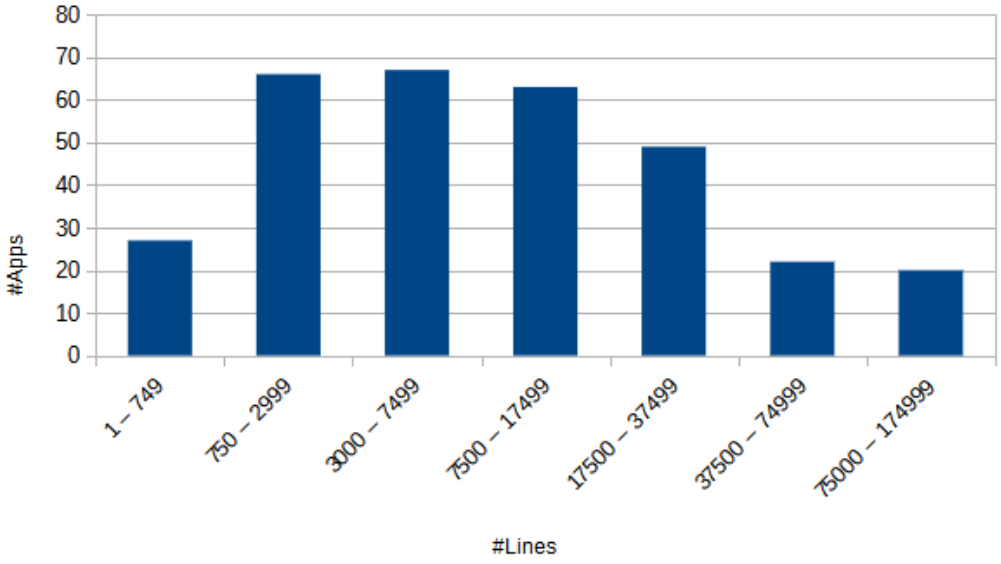
lysant manuellement les informations connexes dans le dépôt GitHub de chaque application. Nous avons trouvé 30 catégories dans le jeu de données. La figure 4.1 montre le nombre d'applications par catégorie. Par exemple, *opensudoku* appartient à la catégorie *Game* et *Wifi-Fixer* appartient à la catégorie *Network*. Deux applications n'ont pas de catégorie, puisque leur dépôt GitHub associé a disparu entre-temps. Les applications du jeu de données sont bien représentées de manière uniforme au sein de ces 30 catégories différentes.

FIGURE 4.2 Distribution des applications par rapport à leur nombre de classes.



2) *Taille :* Nous décrivons notre jeu de données en termes de taille d'application. Nous nous concentrons en particulier sur les métriques du nombre de classes et du nombre de lignes de code. La distribution du nombre de classes est présentée dans la figure 4.2 et la distribution du nombre de lignes est présentée dans la figure 4.3. Le nombre de classes varie de 1 à 1 326 et le nombre de lignes varie de 108 à 166 611. Des applications de toutes tailles sont présentes dans le jeu de données, tant en nombre de classes qu'en nombre de lignes. La majorité des applications ont

FIGURE 4.3 Distribution des applications par rapport à leur nombre de lignes.



entre 15 et 138 classes et entre 2 032 et 21 535 lignes de code.

3) *Intérêt :* Nous décrivons notre jeu de données du point de vue de l'intérêt porté par la communauté GitHub. Les "stars" et les "watchers" nous permettent de mesurer l'intérêt de la communauté envers les différents dépôts. Donner une "star" à un projet permet de montrer son intérêt pour un projet afin de pouvoir le retrouver facilement. Devenir un "watcher" d'un dépôt vous permet de montrer votre intérêt et d'être tenu au courant des activités du dépôt. La distribution des "stars" et des "watchers" est présentée dans la figure 4.4. Nous constatons que le jeu de données représente tous les types d'intérêt, certains dépôts ayant un intérêt très élevé avec des nombres de "stars" atteignant des milliers de "stars", jusqu'à 14 767. La majorité des applications ont entre 29 et 335 "stars" et entre 6 et 31 "watchers".

4) *Contribution* : Nous décrivons également le jeu de données en fonction de la

FIGURE 4.4 Distribution des applications par rapport à leur nombre de “stars” et de “watchers”.

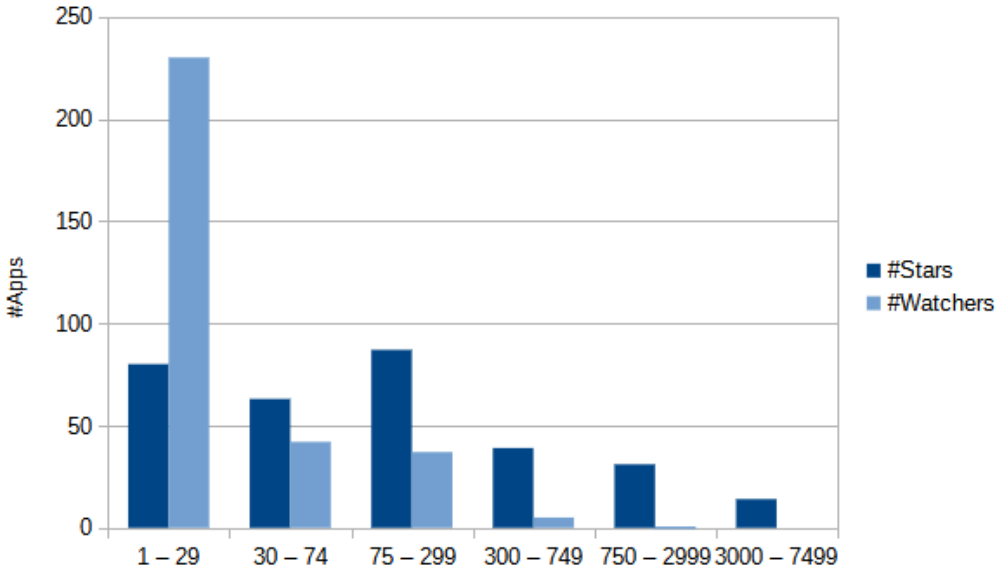


FIGURE 4.5 Distribution des applications par rapport à leur nombre de commits.

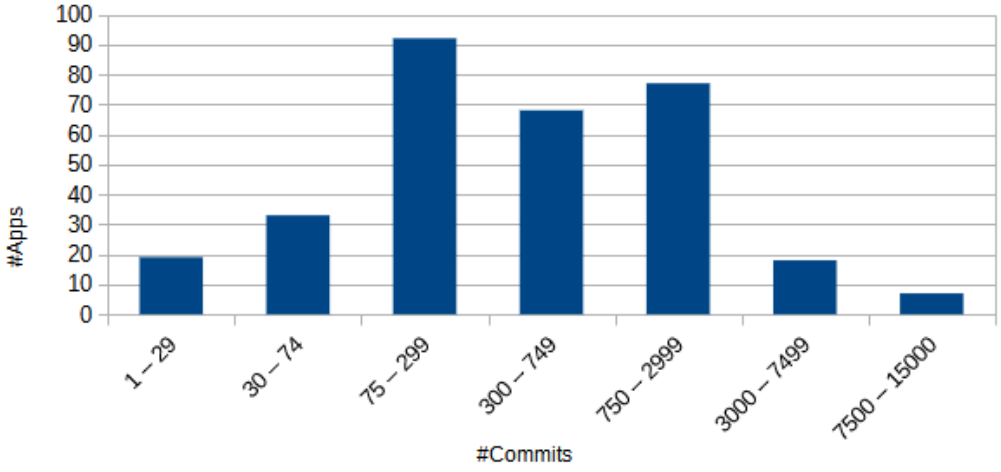
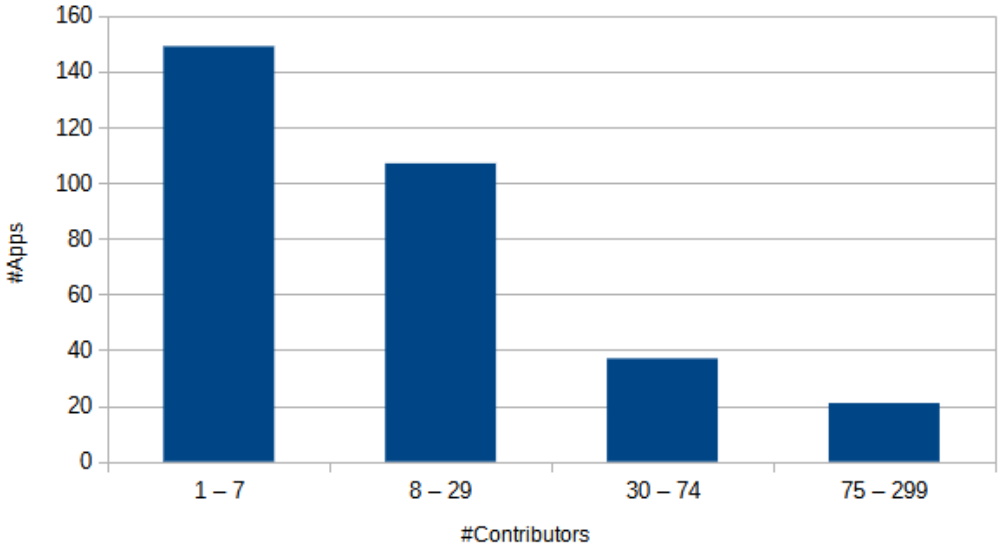


FIGURE 4.6 Distribution des applications par rapport à leur nombre de contributeurs.



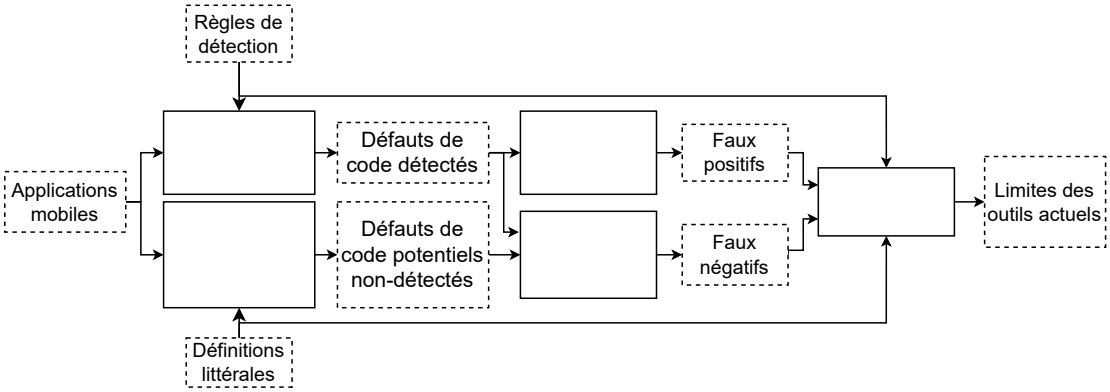
contribution des utilisateurs dans les différents dépôts. Pour ce faire, nous utilisons le nombre de contributeurs aux dépôts ainsi que le nombre de commits. La distribution des commits est présentée dans la figure 4.5 et la distribution des contributeurs est présentée dans la figure 4.6. La majorité des applications ont entre 127 et 1043 commits et entre 3 et 21 contributeurs.

4.4 Processus de l'étude

Le processus de l'étude empirique, tel qu'illustré dans la figure 4.7, consiste en cinq étapes principales décrites ci-dessous.

Étape 1. Nous utilisons PAPRIKA et ADOCTOR sur le jeu de données de 318 applications pour détecter les défauts de code, et donc récupérer les défauts de code détectés (c'est-à-dire les instances positives).

FIGURE 4.7 Vue d'ensemble du processus de validation de l'étude empirique.



Étape 2. En parallèle, à l'étape 2 nous identifions les classes candidates pour les défauts de code potentiels non détectés dans les 318 applications. Ces classes candidates sont toutes les classes susceptibles d'être affectées par un défaut de code conformément à sa définition. Les défauts de code potentiels non détectés (c'est-à-dire les instances négatives) sont les classes candidates qui n'ont pas été détectées comme défauts de code par PAPRIKA ou ADOCTOR. Nous avons choisi les classes candidates en nous basant sur la présence ou non d'une méthode ou d'une instruction mentionnée dans la définition du défaut de code. Il est clair que toute autre classe ne peut pas être une instance du défaut de code. Par exemple, la définition du défaut de code IOD fait référence à la méthode héritée *onDraw* contenant des allocations de mémoire ou ayant un temps d'exécution trop long. Les instances candidates sont donc toutes les classes implémentant la méthode *onDraw*. Nous obtenons ensuite les défauts de code potentiels non détectés de ces classes candidates en retirant toutes les classes déjà détectées par PAPRIKA ou ADOCTOR.

En outre, il est important de noter qu'à l'étape 3 et à l'étape 4, en fonction du défaut de code, il peut y avoir un grand nombre d'instances à analyser manuel-

TABLEAU 4.2 Instances de type Vrai/Faux Positif/Négatif.

	Instances Validées	
	Positives (V)	**Négatives (V^c)**
Défauts de code Détectés (D) (instances positives)	Vrai Positif (VP) $D \cap V$	Faux Positif (FP) $D \cap V^c$
Défauts de code Potentiels non Détectés (D^c) (instances négatives)	Faux Négatif (FN) $D^c \cap V$	Vrai Négatif (VN) $D^c \cap V^c$

lement. Par exemple, comme nous le verrons plus loin dans le tableau 4.3 de la section 4.5, dans notre jeu de données, certains défauts de code peuvent contenir peu d'instances, telles que 19 instances pour le défaut de code IOD, tandis que d'autres ont plus de 1000 instances, comme le défaut de code NLMR.

L'inspection manuelle de chaque instance est une tâche complexe, surtout compte tenu du nombre d'instances. Par conséquent, pour faire face à l'analyse manuelle des étapes 3 et 4, tout en considérant un échantillon statistiquement significatif pour chaque défaut de code, nous nous appuyons sur un échantillonnage stratifié. Cet échantillonnage stratifié garantit que la proportion de chaque défaut de code est préservée dans l'échantillonnage. Plus précisément, nous avons sélectionné de manière aléatoire un ensemble de 312 instances positives et 244 instances négatives de défauts de code. L'échantillonnage pour chaque défaut de code est composé des défauts de code détectés par l'étape 1 et des défauts de code potentiels non détectés par l'étape 2. Cela représente un échantillonnage stratifié statistiquement significatif de 95% avec un intervalle de confiance de 10% des instances obtenues par l'étape 1 et l'étape 2. L'intervalle de confiance est fixé à 10% pour utiliser le même intervalle que le jeu de données de référence de Habchi et al. (Habchi *et al.*, 2019b).

A partir de là, nous pouvons maintenant distinguer les quatre types d'instances présentés dans le tableau 4.4 qui sont déterminés dans les étapes suivantes.

Le processus de la figure 4.7 distingue deux types d'instances de défaut de code. Comme le montre le tableau 4.4, il considère d'abord les *Défauts de code détectés* (D), qui sont les instances retournées par les outils à l'étape 1 de la figure 4.7. Inversement, le complément de cet ensemble, les *Défauts de code potentiels non détectés* D^c est constitué des instances non détectées par les outils et donc renvoyées à l'étape 2 de la figure 4.7. Deuxièmement, les *Instances validées* (V) sont les instances validées lors de l'analyse manuelle de l'étape 3 et de l'étape 4 de la figure 4.7 et sont conformes à la définition des défauts de code. Dans ce cas, le complément V^c est l'ensemble des instances validées lors de l'analyse manuelle qui sont non conformes à la définition des défauts de code.

Nous pouvons maintenant distinguer les quatre types d'instances suivants, comme le montre le tableau 4.4. Les vrais positifs sont les instances détectées par les outils et validées manuellement en conformité avec leur définition de défauts de code : $VP = D \cap V$. Les faux positifs sont les instances détectées par les outils, mais validées manuellement comme étant non conformes à la définition des défauts de code : $FP = D \cap V^c$. Les faux négatifs sont les instances non détectées par les outils, mais validées manuellement comme étant conformes à la définition des défauts de code : $FN = D^c \cap V$. Enfin, les vrais négatifs sont les instances non détectées par les outils et validées manuellement comme étant non conformes à la définition : $VN = D^c \cap V^c$.

Étape 3. Nous analysons manuellement les défauts de code détectés qui sont des faux positifs.

Étape 4. Nous analysons manuellement les défauts de code potentiels non détectées qui sont des faux négatifs.

L'analyse manuelle a été réalisée comme suit. Chaque classe contenue dans l'échantillon des défauts de code a été étudiée manuellement par trois personnes : l'auteur de cette thèse et deux autres doctorants spécialisés dans les applications mobiles et leurs défauts de code. Pour chaque classe à vérifier, chaque participant a examiné la classe de manière approfondie afin de déterminer si le défaut de code se produit réellement. Nous ne partageons les résultats qu'à la fin pour éviter de nous influencer mutuellement. Dans les quelques cas où il y avait des divergences, nous avons réexaminé le cas pour parvenir à un consensus. Ce processus a pris deux semaines. Tous les résultats du consensus peuvent être trouvés dans les artefacts [3]. Les trois participants ont reçu des critères spécifiques pour chaque défaut de code afin de déterminer si une instance est un vrai positif, un faux positif, un vrai négatif ou un faux négatif.

Par exemple, pour le défaut de code HMU qui se réfère à la taille des structures de données, nous considérons comment ces structures sont remplies. Si elles sont remplies un nombre fini de fois ou dans des boucles itérant sur une petite variable, nous les considérons comme petites. Si elles sont remplies par des boucles itérant sur une variable de valeur éventuellement très grande, nous supposons qu'elles sont éventuellement grandes. Pour les défauts de code comme HSS, où le temps d'exécution est crucial, les participants ont recherché les boucles itérant sur une variable de valeur éventuellement très grande, les nombreux appels à des méthodes complexes ou les nombreux objets utilisés ou alloués. Les règles permettant de déterminer si un défaut de code s'est réellement produit sont définies dans l'analyse qualitative de la section 4.5.2, dans les parties consacrées à l'analyse manuelle.

Ensuite, nous calculons la précision dans cette étape. La précision est la proportion de vrais positifs parmi les positifs (ici, défauts de code détectés) $\frac{|VP|}{|D|}$. Ensuite, nous

3. `https://figshare.com/s/8235a0575ff4a88f0deb`

calculons le rappel. Le rappel est la proportion de vrais positifs parmi les vrais positifs et les faux négatifs $\frac{|VP|}{|VP \cup FN|}$.

Étape 5. Enfin, nous identifions les incohérences entre les règles de détection de PAPRIKA et ADOCTOR avec les définitions littérales en nous appuyant sur les faux positifs/négatifs et nous en déduisons les limites de ces outils. Cette validation a été effectuée par un développeur qui a une connaissance approfondie des défauts de code spécifiques à Android. Cette étape de validation n'est pas sujette à interprétation et ne nécessite pas l'implication de plusieurs développeurs, puisqu'il s'agit d'un processus simple comme indiqué dans ce qui suit. L'étape de validation comprend les trois sous-étapes suivantes. La première sous-étape consiste à comparer les règles de détection des outils avec les définitions littérales et à déduire les incohérences entre elles. La deuxième sous-étape consiste à confirmer ces incohérences avec les faux positifs/négatifs qui illustrent des cas concrets de défauts de code détectés et de défauts de code potentiels non détectés. Dans la troisième sous-étape, sur la base des incohérences confirmées, les limites des outils sont identifiées. Par exemple, la définition littérale du défaut de code HMU indique qu'un défaut de code HMU se manifeste lorsque des *HashMaps* sont utilisées pour de petites structures ou des *ArrayMaps* sont utilisées pour de grandes structures. Cependant, la règle de détection du défaut de code HMU de PAPRIKA indique seulement qu'il y a un défaut de code lorsqu'une structure *HashMap* est utilisée. En termes d'incohérences, la règle de détection ne prend pas en compte la taille des structures et ne se réfère pas spécifiquement à la structure *ArrayMap*.

4.5 Résultats

Dans cette section, nous répondons aux deux questions de recherche en utilisant les résultats de l'étude empirique. Nous répondons à la première question de re-

Tableau 4.3 Résultats rapportés par les outils Paprika et aDoctor.

Défaut de code	**#Applications**	**#Instances**	**Outil**
DW	81	199	aDoctor
NLMR	292	2826	aDoctor
NLMR	259	1132	Paprika
HMU	136	729	Paprika
HAS	53	144	Paprika
HSS	38	48	Paprika
HBR	132	513	Paprika
IOD	16	19	Paprika

cherche par une analyse quantitative. En ce qui concerne la deuxième question de recherche, nous y répondons par une analyse qualitative.

4.5.1 QR1 : Est-ce que les outils étudiés sont efficaces pour détecter les défauts de code comportementaux ?

Pour répondre à cette question de recherche, nous nous concentrerons sur les résultats quantitatifs de notre étude. Les différents types d'instances traités dans l'étude quantitative sont détaillés et expliqués dans le tableau 4.4 de la section 4.4.

Nous allons maintenant présenter nos résultats. Le tableau 4.3 montre le nombre d'applications et le nombre de défauts de code détectés par Paprika et aDoctor à l'étape 1 de la figure 4.7. Il convient de noter qu'un seul défaut de code est commun aux deux outils, le défaut de code NLMR. Il est également intéressant de mentionner que même si, dans ce cas, les deux outils renvoient des classes *Activity* ne définissant pas la méthode *onLowMemory*, le nombre de classes et le nombre d'instances diffèrent. Cela est dû au fait que Paprika vérifie si l'une des

Tableau 4.4 Nombre de défauts de code potentiels non détectés.

Défaut de code	**#Applications**	**#Instances**	**Outil**
DW	46	64	aDoctor
NLMR	18	33	aDoctor
NLMR	18	33	Paprika
HMU	13	24	Paprika
HAS	159	846	Paprika
HSS	72	113	Paprika
HBR	112	376	Paprika
IOD	102	274	Paprika

superclasses de la classe candidate est la classe *Activity*, tandis que aDoctor vérifie si la classe étend l'une des 95 classes Android d'une liste prédéfinie.

Le tableau 4.4 présente le nombre de défauts de code potentiels non détectés tels que retournés par l'étape 2 de la figure 4.7. Comme décrit dans le processus, les défauts de code potentiels non détectés sont les classes candidates affectées par un défaut de code, mais non détectées par les outils. Ici, le nombre de défauts de code potentiels non détectés pour NLMR est le même pour les deux outils. En effet, l'identification des classes candidates à l'étape 2 de la figure 4.7 est indépendante des outils et le filtrage des instances retournées par ces outils donne, dans ce cas, le même résultat. Même si les définitions dans la littérature sont similaires, les règles de détection peuvent varier d'un outil à l'autre. Les deux outils détectent un défaut de code NLMR si une classe *Activity* n'implémente pas la méthode *onLowMemory*. Alors que l'outil Paprika se concentre uniquement sur les classes héritant de *Activity*, l'outil aDoctor s'intéresse également aux autres classes héritant de *ComponentCallBacks*, tel que *Fragments*. Ceci est raisonnable puisque

TABLEAU 4.5 Résultats de l'échantillonnage des défauts de code étudiés.

Défaut de code	#Défauts de code détectés	# Défauts de code potentiels non détectés	#faux positifs	#faux negatifs	Précision	Rappel	F_1-Mesure	Outil
DW	66	64	0 (0%)	25 (39%)	100%	73%	84%	aDoctor
NLMR	93	33	0 (0%)	6 (18%)	100%	94%	97%	aDoctor
NLMR	89	33	0 (0%)	6 (18%)	100%	94%	97%	Paprika
HMU	85	24	42 (49%)	10 (42%)	50%	81%	62%	Paprika
HSS	48	52	35 (73%)	10 (19%)	27%	57%	37%	Paprika
IOD	19	71	0 (0%)	12 (17%)	100%	61%	76%	Paprika
Total	400	277	88 (28%)	62 (25%)	N/A	N/A	N/A	N/A
Moyenne	N/A	N/A	N/A	N/A	64%	69%	64%	N/A

la méthode *onLowMemory* est définie dans *ComponentCallBacks*. Nous avons suivi la définition et ne nous sommes intéressés qu'aux classes *Activity* pour les classes candidates, que les deux outils détectent à un niveau similaire. Par conséquent, les défauts de code détectés diffèrent selon les outils, tandis que les défauts de code potentiels non détectés sont les mêmes.

Le tableau 4.5 présente le nombre de faux positifs et de faux négatifs tel que déterminé aux étapes 3 et 4 de la figure 4.7. Il y a une certaine variation d'un défaut de code à l'autre, mais chaque défaut de code a son lot de faux positifs et de faux négatifs. Pour les faux positifs, on observe que l'on peut passer de 0% pour les défauts de code dont la détection assure que le défaut de code se produit effectivement, à 72,92% pour les résultats pour lesquels une vérification manuelle est nécessaire et indispensable. D'autre part, pour les faux négatifs, nous observons que nous pouvons passer de 17% pour les défauts de code comportementaux

pour lesquelles les outils couvrent presque toutes les instances, à 42% pour les défauts de code où les outils ratent beaucoup d'instances. Étant donné que les défauts de code HBR, HAS et HSS sont très similaires et ne diffèrent que par le nom de la méthode, et qu'il existe un grand nombre d'instances pour chacune d'entre elles (voir le tableau 4.3), l'étude se concentre uniquement sur HSS. Les moyens de détection pour ces trois défauts de code sont exactement les mêmes en détectant selon le nombre d'instructions et la complexité cyclomatique selon les mêmes seuils. Dans certains cas, le nombre d'instances indiqué dans le tableau 4.5 coïncide avec le nombre d'instances du tableau 4.3 ou du tableau 4.4. Il s'agit d'une simple conséquence de la stratification. En effet, les défauts de code avec peu d'instances auront souvent le même ou presque le même nombre d'instances après stratification.

Les faux positifs et les faux négatifs proviennent des règles de détection utilisées dans Paprika et aDoctor. On peut donc conclure que ces outils capturent beaucoup plus de défauts de code que nécessaire. Ceci est une conséquence d'une analyse statique basée sur des patrons de recherche et le calcul de métriques qui ne permet pas une précision parfaite conduisant à des faux positifs. Pour des raisons similaires, les techniques de détection ne capturent pas tous les défauts de code qui devraient être capturés, ce qui conduit à des faux négatifs. Ceci est principalement dû aux limitations techniques apportées par une détection purement statique des défauts de code comportementaux, comme nous le verrons plus loin dans cette section. Les défauts de code comportant de nombreuses instances peuvent entraîner un surplus de travail afin de vérifier si ces instances sont pertinentes. En effet, Paprika (Hecht, 2016) indique que pour certains défauts de code étudiés, comme NLMR et HMU, la responsabilité est laissée au développeur de vérifier si une instance identifiée par Paprika mérite ou non une correction. Les faux positifs et les faux négatifs sont analysés et discutés plus en détail à la section 4.5.2. Les

règles à l'origine des faux négatifs et des faux positifs sont définies dans l'analyse qualitative de la section 4.5.2.

On peut également considérer la précision, le rappel et la mesure F_1. La précision est la proportion de vrais positifs parmi tous les positifs, le rappel est la proportion de vrais positifs parmi les vrais positifs et les faux négatifs, enfin la mesure F_1 est la moyenne harmonique de la précision et du rappel. La précision est généralement élevée, mais dans certains cas, comme HMU et HSS, plus de la moitié des instances détectées ne sont pas pertinentes. Dans ces cas, il est donc laissé aux développeurs de déterminer si les défauts de code détectés sont pertinents. Le rappel a tendance à être plus faible à la précision, ce qui signifie que de nombreuses instances pertinentes sont absentes des instances détectées. C'est donc aux développeurs de trouver ces instances manquantes. Sur la base des résultats du tableau 4.5, nous rejetons donc H_1^{posi} et H_2^{nega}.

Réponse $\boldsymbol{QR_1}$: Les outils renvoient des faux positifs jusqu'à environ 73% pour le défaut de code HSS et des faux négatifs jusqu'à environ 42% pour le défaut de code HMU. Ils ont tous deux un impact sur l'efficacité. Les faux positifs détectés par les outils concernés affectent leur efficacité, puisqu'ils renvoient aux développeurs des résultats de détection erronés. Les faux négatifs affectent l'efficacité en renvoyant aux développeurs des résultats de détection partiels. La précision peut être faible pour des défauts de code spécifiques comme HMU avec 27% ou HSS avec 50%. Le rappel descend jusqu'à 57%. Dans l'ensemble, la précision des défauts de code étudiés est de 74%, le rappel de 69% et la mesure F_1 de 64%, ce qui est assez faible.

4.5.2 QR2 : Les défauts de code comportementaux détectés par les outils étudiés sont-ils cohérents avec leur définition littérale originale ?

Alors que la section précédente se concentrait principalement sur les résultats quantitatifs, cette section aborde la nature des résultats décrits dans le tableau 4.5 avec une analyse qualitative. Nous étudions donc en détail les défauts de code sélectionnés pour vérifier dans quelle mesure les résultats de la détection vont à l'encontre de leur définition littérale.

Pour chaque défaut de code, nous suivons un modèle de description commun, qui se réfère aux termes de la figure 4.7 : a) un extrait de la définition littérale du défaut de code de la section 4.2 comme rappel. b) les règles de détection concrètes mises en œuvre dans les outils aDoctor et Paprika pour détecter le défaut de code indépendamment de ce qui a été décrit dans les articles connexes ; c) la nature des défauts de code détectés, c'est-à-dire la manière dont les classes qui contiennent ces défauts de code sont caractérisées ; d) la nature des défauts de code potentiels non détectés ; e) les critères utilisés pendant l'analyse manuelle pour les défauts de code détectés ; f) les critères utilisés pendant l'analyse manuelle pour les défauts de code potentiels non détectés ; g) la nature des faux positifs avec un exemple illustratif, si applicable ; h) la nature des faux négatifs avec un exemple illustratif, si applicable ; i) une discussion sur les résultats ; j) les limitations techniques identifiées des outils aDoctor et Paprika avec les problèmes associés aux règles de détection actuelles. Chaque modèle de description associé à un défaut de code nous permet de répondre à la question de recherche QR_2, à savoir si le défaut de code détecté par aDoctor et Paprika est cohérent avec sa définition littérale originale.

Durable Wakelock (DW)

FIGURE 4.8 Exemple d'un faux négatif du défaut de code DW dans l'application *Conversations* et la classe *AudioPlayer*.

```
private void acquireProximityWakeLock() {
    synchronized (AudioPlayer.LOCK) {
        if (wakeLock != null) {
            wakeLock.acquire();
        }
    }
}

private void releaseProximityWakeLock() {
    synchronized (AudioPlayer.LOCK) {
        if (wakeLock != null && wakeLock.isHeld()) {
            wakeLock.release();
        }
    }
    messageAdapter.setVolumeControl(AudioManager.STREAM_MUSIC);
}
```

a) Définition littérale : Un défaut de code DW se manifeste lorsqu'une instance de la classe *WakeLock* acquiert le verrou (en utilisant la méthode *acquire*) sans le libérer (en utilisant la méthode *release*).

b) Règles de détection : (ADOCTOR) Un défaut de code DW se manifeste s'il y a un appel d'une méthode dont le nom contient la chaîne de caractères *"acquire"*. Cette règle de détection ne vérifie pas spécifiquement la méthode *acquire* mais une chaîne de caractères *"acquire"*. De même, elle ne vérifie pas s'il existe une instance de la classe *WakeLock* et si la méthode *release* est absente. Cependant, la règle de détection présentée dans l'article de ADOCTOR (Palomba *et al.*, 2017) indique de vérifier également l'absence de *release* et que les deux méthodes appartiennent à la classe *WakeLock*. Nous avons donc vérifié manuellement dans les défauts de code détectés l'instance *WakeLock* et l'absence de *release*. Nous ne rapportons les résultats que sur cette version améliorée de la règle.

c) Défauts de code détectés : Avec cette règle améliorée, les défauts de code

DW détectés dans aDoctor sont des classes qui appellent la méthode *acquire* de la classe *WakeLock*, mais qui n'implémentent pas la méthode *release*.

d) Défauts de code potentiels non détectés Il s'agit des classes contenant les deux méthodes de la classe *WakeLock*.

e) Analyse manuelle des défauts de code détectés : Aucune analyse manuelle n'est nécessaire, puisque tous les défauts de code détectés sont des vrais positifs. En effet, la méthode *release* n'est pas implémentée dans les classes concernées.

f) Analyse manuelle des défauts de code potentiels non détectés : Si les deux méthodes sont présentes, plusieurs critères sont appliqués. Si les deux appels de méthode sont séparés par tout un tas d'appels de fonction et de tests de conditions dont nous ne sommes pas sûrs qu'ils soient exécutés/satisfaits, alors nous ne pouvons pas être sûrs que les deux méthodes sont correctement appelées.

g) Faux positifs : Chaque défaut de code détecté est irrévocablement un vrai positif, puisque la méthode *release* n'est pas implémentée dans les défauts de code détectés.

h) Faux négatifs : Les vrais négatifs sont les défauts de code potentiels non détectés où les deux méthodes seront appelées, tandis que les faux négatifs sont les défauts de code potentiels non détectés où ce n'est pas le cas. Près de 39% des défauts de code potentiels non détectés sont des faux négatifs, ce qui signifie que nous ne pouvons pas être suffisamment sûrs que les deux méthodes sont appelées correctement pour minimiser la consommation de la batterie. Cela peut être dû au fait que les deux méthodes sont très éloignées l'une de l'autre et que de nombreuses conditions ne sont pas toujours respectées, comme le montre la figure 4.8. Comme les appels se font dans deux méthodes dissociées, ici *acquireProximityWakeLock()* et *releaseProximityWakeLock()*, il est également nécessaire de vérifier si elles sont appelées correctement. Cependant, nous n'avons aucun indice nous permettant de savoir si ces fonctions sont correctement appelées dans le bon ordre. Il se peut

également que le *release* ne soit effectué que lorsque l'application est détruite, ce qui n'est pas conforme à la définition du défaut de code. Ce dernier scénario est le plus fréquent parmi les faux négatifs rencontrés.

i) Discussion sur les résultats : Ce défaut de code est l'un des plus difficiles à déterminer manuellement. Il est difficile de déterminer si deux méthodes distantes seront appelées sur des applications dont la structure n'est pas entièrement connue.

j) Limites des outils actuels : L'implémentation actuelle d'ADOCTOR sur la détection du défaut de code DW identifie simplement la présence de la méthode *acquire* sans vérifier la présence de la méthode *release*, et ce n'est pas parce que les deux méthodes sont présentes qu'elles sont nécessairement appelées. Nous avons pris en compte la version "améliorée" décrite dans l'article afin d'obtenir des résultats à analyser, autrement tous les résultats ou presque seraient invalides.

FIGURE 4.9 Exemple de faux positif du défaut de code HMU dans l'application *Anki-Android* et la classe *Media*.

```
private Pair<List<String>, List<String>> _changes() {
    Map<String, Object[]> cache = new HashMap<>();
    try (Cursor cur = mDb.getDatabase().query("select fname,
       csum, mtime from media where csum is not null", null)) {
        while (cur.moveToNext()) {
            String name = cur.getString(0);
            String csum = cur.getString(1);
            Long mod = cur.getLong(2);
            cache.put(name, new Object[] { csum, mod, false });
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    (...)
}
```

FIGURE 4.10 Exemple de faux négatif du défaut de code HMU dans l'application *client-android* et la classe *PBMediaStore*.

```
private void getBucketData(final ArrayMap<String, String>
    bucketNamesList, final String buckedId, final String
       buckedName) {
 final String[] projection = { buckedId, buckedName, "count(*)
    as media_count" };
 final String groupBy = "1) GROUP BY (2";
 final Cursor cursor =
    PBApplication.getApp().getContentResolver().query(imagesUri,
    projection, groupBy, null, "media_count desc");

 if (cursor != null && cursor.moveToFirst()) {
        (...)
  do {
   (...)
   bucketNamesList.put(id, name + " (" + count + ")");
  } while (cursor.moveToNext());
 }
 (...)
}
```

HashMap Usage (HMU)

a) Définition littérale : Un défaut de code HMU se manifeste lorsqu'une structure *HashMap* est utilisée pour un petit ensemble d'objets et *SimpleArrayMap*/*ArrayMap* pour un grand ensemble d'objets.

b) Règles de détection : (PAPRIKA) Une classe utilise une structure *HashMap*.

c) Défauts de code détectés : Il s'agit des classes qui utilisent une structure de type *HashMap*.

d) Défauts de code potentiels non détectés : Ce sont les classes qui utilisent une structure *SimpleArrayMap* ou *ArrayMap*.

e) Analyse manuelle des défauts de code détectés : Nous examinons comment ces structures sont remplies. Si elles sont remplies un nombre fini de fois ou dans des boucles itérant sur une petite variable, nous les considérons comme

petites. Si elles sont remplies par des boucles itérant sur une variable de valeur éventuellement très grande, nous supposons qu'elles sont éventuellement grandes.

f) Analyse manuelle des défauts de code potentiels non détectés : Les mêmes critères sont utilisés, puisqu'il est également nécessaire d'examiner la taille des structures.

g) Faux positifs : Conformément à la définition du défaut de code, un vrai positif est une instance qui utilise une *HashMap* pour une petite structure, et un faux positif est une instance qui utilise une *HashMap* pour une grande structure. Un exemple du défaut de code HMU détecté par PAPRIKA est présenté dans la figure 4.9. Ici, PAPRIKA détecte un défaut de code, puisqu'il identifie une *HashMap* et laisse à l'utilisateur le soin de vérifier si la détection est efficace. Cependant, dans cet exemple, la structure *cache* n'a aucune limitation sur le *.put()* qui lui est associé. On peut s'attendre à ce que la *HashMap cache* soit volumineuse, puisque la requête peut renvoyer beaucoup de résultats. Par conséquent, il ne peut pas être considéré comme un défaut.

g) Faux négatifs : Inversement, un vrai négatif est une instance qui utilise une *SimpleArrayMap/ArrayMap* pour une structure courte, et un faux négatif est une instance qui utilise une *SimpleArrayMap/ArrayMap* pour une grande structure. Les faux négatifs devraient également inclure *HashMap* utilisé pour un petit nombre d'éléments. Cependant, il n'y a pas de tels cas dans notre échantillon. Comme PAPRIKA signale toutes les *HashMaps* comme des défauts de code, toutes les instances contenant des *HashMaps* sont des défauts de code détectés. Un exemple de défaut de code potentiel non détecté est illustré dans la figure 4.10. Ici, PAPRIKA ne détecte pas de défaut de code, puisqu'il se concentre uniquement sur les *HashMap*. Cependant, dans cet exemple, l'*ArrayMap bucketNamesList* n'a aucune limitation sur le *.put()* qui lui est associé. On peut s'attendre à ce que ce *ArrayMap* soit de grande taille, puisqu'elle est remplie à l'aide d'une requête provenant d'une base de données de médias. Par conséquent, comme cette structure

devrait rester une petite structure, ce n'est pas conforme à la définition et peut être considéré comme un défaut.

i) Discussion sur les résultats : Les résultats sont assez similaires, que ce soit dans le cas de défauts de code détectés avec 49% de faux positifs ou de défauts de code potentiels non détectés avec 42% de faux négatifs, ce qui signifie que dans les deux cas, la détection semble être insuffisante pour identifier correctement le défaut de code. Il est important de noter que les développeurs ne semblent pas utiliser spécifiquement une structure ou une autre en fonction de la taille envisagée, ce qui suggère qu'une détection en regardant l'une des structures n'est pas assez efficace. Cependant, *HashMap* est la structure la plus utilisée et de très loin.

j) Limites des outils actuels : Il n'y a pas de notion de taille de structure, les outils détectent seulement la présence d'un *HashMap*.

Heavy Service Start (HSS)

a) Définition littérale : Un défaut de code HSS se manifeste lorsque la méthode *OnStartCommand* contient des opérations chronophages ou bloquantes.

b) Règles de détection : (Paprika) La méthode *onStartCommand* a plus de 17 instructions ou une complexité cyclomatique supérieure à 3,5 (lorsque le seuil est réglé sur faible).

c) Défauts de code détectés : Ce sont les classes définissant la méthode *onStartCommand* ayant une complexité cyclomatique supérieure à 3,5 ou un nombre d'instructions supérieur à 17.

d) Défauts de code potentiels non détectés : Ce sont les classes définissant la méthode *onStartCommand* ayant une complexité cyclomatique inférieure à 3,5 et un nombre d'instructions inférieur à 17.

e) Analyse manuelle des défauts de code détectés : Nous déterminons si une méthode n'est pas chronophage en examinant le code source selon plu-

FIGURE 4.11 Exemple de faux négatif HSS dans l'application *QuickLyrics* et la classe *BatchDownloaderService*.

```
@Override
public int onStartCommand(Intent intent, int flags, int
    startId) {
 onHandleIntent(intent);
 return START_NOT_STICKY;
}

@Override
protected void onHandleIntent(Intent intent) {
 (...)
 Cursor cursor = getContentResolver().query(content,
     projection, selection, null, null);
 (...)
 newSongsMetadata = new ArrayList<>();
 while (cursor.moveToNext()) {
  (...)
  newSongsMetadata.add(new String[]{artist, title});
 }
 (...)
 for (String[] track : newSongsMetadata) {
    (...)
     File musicFile =
         Id3Reader.getFile(BatchDownloaderService.this, artist,
         title, false);
     (...)
 }
}
```

sieurs indicateurs. Voici une liste non exhaustive d'indicateurs : 1) La méthode ne comporte pas de boucles ou des boucles itérant sur une petite variable ; 2) Elle ne comporte que des appels de méthode triviaux dont le contenu est connu et reconnu ; 3) Il n'y a pas d'opérations potentiellement coûteuses, comme des requêtes sur des éléments volumineux dans une base de données ; 4) Il y a peu d'allocations de mémoire, nous pouvons supposer qu'elle n'est pas chronophage. Si nous ne pouvons pas déterminer l'aspect chronophage à cause de bouts de code

FIGURE 4.12 Exemple de faux positif HSS dans l'application *OpenManga* et la classe *SaveService*.

```
@Override
public int onStartCommand(Intent intent, int flags, int
    startId) {
 int action = intent != null ? intent.getIntExtra("action", 0)
     : 0;
 switch (action) {
  case ACTION_ADD:
   (...)
   break;
         (...)
  case ACTION_PAUSE:
   if (id == 0) break;
   saveTask = mTasks.get(id);
   if (saveTask != null) {
    saveTask.pause();
   }
   break;
  case ACTION_RESUME:
   (...)
   break;
         (...)
 }
 return START_REDELIVER_INTENT;
}
```

inconnus, nous les considérons comme potentiellement chronophages pour les défauts de code détectés et potentiellement non chronophages pour les défauts de code potentiels non détectés. En effet, les défauts de code détectés ont déjà été identifiés avec des méthodes longues et complexes, tandis que les défauts de code potentiels non détectés ont des méthodes courtes et non complexes.

f) Analyse manuelle des défauts de code potentiels non détectés : Les mêmes critères sont utilisés, puisque nous devons également examiner si la méthode prend du temps.

g) Faux positifs : Un défaut de code détecté est un faux positif si la méthode ne

prend pas beaucoup de temps. Très souvent, ces fonctions consistent en un grand "switch" ou une séquence de "if-else", ce qui donne une grande fonction, mais sans beaucoup d'opérations exécutées. La figure 4.12 est un extrait d'un défaut de code détecté. Bien que la méthode *onStartCommand* soit longue de 57 lignes, elle est subdivisée en petits blocs d'instructions simples. La méthode consiste en un grand "switch" et chaque branche du "switch" est composée d'opérations mineures. Une telle instance ne peut pas être considérée comme un défaut de code, puisqu'on peut s'attendre à ce que la méthode ait un temps d'exécution court.

g) Faux négatifs : Les faux négatifs sont les potentiels défauts de code non détectés qui prennent beaucoup de temps. Dans certains cas, malgré la brièveté d'une méthode, celle-ci peut tout de même prendre du temps ou bloquer des opérations. Ces faux négatifs sont généralement composés de boucles coûteuses, de l'imbrication de nombreuses sous-méthodes. La figure 4.11 est un extrait d'un défaut potentiel de code non détecté. Bien que la méthode *onStartCommand* soit très brève, elle se compose d'une longue sous-méthode *onHandleIntent* de 70 lignes. Cette dernière est composée de nombreuses opérations complexes, telles que des requêtes sur un grand jeu de données et des boucles effectuant des opérations sur ces données. Une telle instance peut être considérée comme un défaut de code, puisqu'on peut s'attendre à ce qu'une telle méthode ait long temps d'exécution.

i) Discussion sur les résultats : Malgré des critères similaires, il y a beaucoup plus de faux positifs (73%) que de faux négatifs (19%). La différence entre les faux positifs et les faux négatifs montre que la métrique est suffisamment efficace pour détecter les opérations courtes, mais inefficace pour déterminer les opérations longues.

j) Limites des outils actuels : La détection est basée sur des heuristiques arbitraires. Plutôt que de vérifier réellement si les méthodes concernées impliquent des opérations longues ou bloquantes, des heuristiques sont utilisées pour signaler comme défectueuses les méthodes dont le nombre d'instructions et les mesures de

complexité cyclomatique sont supérieurs à une valeur arbitraire.

FIGURE 4.13 Exemple de faux négatif du défaut de code IOD dans l'application *osmeditor4android* et la classe *MapTilesLayer*.

```
@Override
protected void onDraw(Canvas c, IMapView osmv) {
    (...)
    for (int y = tileNeededTop; y <= tileNeededBottom; y++) {
        for (int x = tileNeededLeft; x <= tileNeededRight;
            x++) {
            (...)
            while ((tileBitmap == null) && (zoomLevel -
                tile.zoomLevel) <= maxOverZoom &&
                tile.zoomLevel > minZoom) {
                (...)
                tileBitmap = mTileProvider.getMapTile(tile,
                    owner);
            }

            if (tileBitmap != null) {
                c.drawBitmap(tileBitmap, new Rect(tx, ty, tx +
                    sw, ty + sh),
                        new Rect(destRect.left + xPos,
                            destRect.top + yPos,
                                destRect.right +
                            xPos, destRect.bottom + yPos),
                            mPaint);
            } (...)
    }
}
```

Init OnDraw (IOD)

a) Définition littérale : Un défaut de code IOD se manifeste lorsque la méthode *onDraw* contient des instructions *init* pour allouer de la mémoire ou contient des opérations qui prennent du temps.

b) Règles de détection : (PAPRIKA) Une méthode *onDraw* appelle des constructeurs.

c) Défauts de code détectés : Les défauts de code détectés dans PAPRIKA sont les classes utilisant la méthode *onDraw* contenant des appels de constructeur.

d) Défauts de code potentiels non détectés : Ce sont les classes utilisant la méthode *onDraw* non détectée par PAPRIKA, et donc excluant celles qui contiennent directement des appels de constructeur.

e) Analyse manuelle des défauts de code détectés : Comme il n'y a pas de faux positifs comme expliqué dans la suite (voir g)), il n'y a pas d'analyse manuelle des défauts de code détectés.

f) Analyse manuelle des défauts de code potentiels non détectés : Les critères sont les mêmes que pour le défaut de code HSS, puisque les critères concernent la mémoire et les opérations qui prennent du temps.

g) Faux positifs : Un défaut de code détecté est un faux positif si, bien qu'il soit signalé comme un défaut de code, la méthode n'est pas chronophage et ne comporte pas d'allocations de mémoire. Comme chaque défaut de code détecté comporte des allocations de mémoire, il n'y a pas de faux positifs.

g) Faux négatifs : Un faux négatif est un défaut de code potentiel non détecté où la méthode *onDraw* présente des opérations qui prennent du temps et une allocation de mémoire incorrecte. La figure 4.13 est un extrait d'un défaut de code potentiel non détecté. Il n'y a pas d'initialisations directes, mais il s'agit tout de même d'une longue méthode de 150 lignes qui effectue de nombreux calculs qui auraient pu être effectués en dehors de cette méthode. Par exemple, cette méthode contient toute une série de calculs liés au zoom qui pourraient être effectués ailleurs, ainsi que des boucles imbriquées. Une telle méthode peut avoir de fortes conséquences sur les performances de l'application en étant trop longue à l'exécution, comme l'indique la définition du défaut de code IOD. On peut s'attendre à ce que cette méthode soit coûteuse et peut donc être considérée comme un défaut de code.

i) Discussion sur les résultats : Seule une petite partie des défauts potentiels

du code non détectés semble prendre du temps avec 17% de faux négatifs. Comme pour HSS, la détection semble assez efficace pour ne pas négliger les instances qui doivent être détectées. Cela signifie que les méthodes sans initialisation ne comportent généralement pas d'opérations chronophages, mais qu'il existe encore des instances comportant de nombreuses boucles de calcul chronophages à chaque itération de *onDraw*.

j) Limites des outils actuels : La présence de l'initialisation de nouveaux objets ne suffit pas à gérer le temps d'exécution.

FIGURE 4.14 Exemple de la méthode *onLowMemory* dans l'application *osmeditor4android* et la classe *MapViewLayer*.

```
public void onLowMemory() {}
```

FIGURE 4.15 Exemple de la méthode *onLowMemory* dans l'application *trackworktime* et la classe *WorkTimeTrackerApplication*.

```
public void onLowMemory() {
  Logger.info("low memory for application");
  super.onLowMemory();
 }
```

No Low Memory Resolver (NLMR)

a) Définition littérale : Une classe *Activity* qui ne définit pas *onLowMemory*, ou qui définit cette méthode, mais dans laquelle *onLowMemory* n'effectue aucune action de récupération de mémoire est considérée comme un défaut de code.

b) Règles de détection : (PAPRIKA & ADOCTOR) Une classe *Activity* ne possède pas de méthode *onLowMemory*. Les règles de détection sont les mêmes pour ADOCTOR et PAPRIKA, mais leurs implémentations sont légèrement différentes. PAPRIKA vérifie si l'une des superclasses de la classe candidate est la classe *Acti-*

vity, tandis que aDoctor vérifie si la classe étend l'une des 95 classes Android d'une liste prédéfinie. Cette différence d'implémentation a un impact sur le nombre de défauts de code détectés, comme le montre le tableau 4.3.

c) Défauts de code détectés : Il s'agit des classes *Activity* qui ne définissent pas la méthode *onLowMemory*.

d) Défauts de code potentiels non détectés : Il s'agit des classes *Activity* qui définissent la méthode *onLowMemory* et n'effectuent aucune action de récupération de mémoire.

e) Analyse manuelle des défauts de code détectés : Aucune analyse manuelle n'est nécessaire, puisque tous les défauts de code détectés sont des vrais positifs.

f) Analyse manuelle des défauts de code potentiels non détectés : Nous vérifions que les méthodes *onLowMemory* n'effectuent aucune action de récupération de mémoire.

g) Faux positifs : La méthode *onLowMemory* n'est pas implémentée dans tous les défauts de code détectés, ce sont donc tous de vrais positifs. Il n'y a donc pas de faux positifs.

h) Faux négatifs : Lors de l'inspection des méthodes *onLowMemory* des défauts de code potentiels non détectés, certaines méthodes n'effectuent aucune action de récupération de mémoire. Plus particulièrement, cela se produit pour les raisons suivantes : (1) le corps de la méthode est vide, comme dans la figure 4.14 ; (2) elle sert uniquement à la journalisation, comme dans la figure 4.15, (3) elle n'a aucun effet comme l'appel de la méthode *super.onLowMemory*, qui est également vide.

i) Discussion sur les résultats : Ce défaut de code n'est pas fréquent. Seulement 18% des méthodes étudiées (soit 6 instances) n'ont aucune action sur la mémoire. En effet, lorsque la méthode *onLowMemory* est déclarée, il y a généralement une action de récupération de mémoire.

j) Limites des outils actuels : Les outils ne prennent en compte que l'absence

de la méthode *onLowMemory* mais ils ne prennent pas en compte sa présence sans action de récupération de mémoire.

Résumé des résultats qualitatifs. Les défauts de code sont incohérents avec leur définition littérale si des éléments de la définition sont manquants dans les règles de détection de l'outil. Nous avons listé les incohérences dans la partie "Limites des outils actuels" pour chaque défaut de code. Par exemple, la définition littérale du défaut de code HMU indique qu'une *HashMap* doit être utilisée uniquement pour les grandes structures, tandis qu'une *SimpleArrayMap* est préférable pour les petites structures. Mais, dans ce cas, les instances détectées par l'outil contiennent la structure *HashMap*, alors que les grandes *HashMap* sont cohérentes avec la définition littérale. Il n'y a également aucune considération pour *SimpleArrayMap* dans la règle de détection de l'outil malgré son importance dans la définition littérale. Un autre exemple : pour le défaut de code DW, lorsqu'un *WakeLock* est acquis par la méthode *acquire*, la méthode *release* doit être appelée. Cependant, l'outil ne considère que l'existence des méthodes *acquire* et *release*. Toutefois, cela ne signifie pas nécessairement que ces méthodes sont appelées, comme l'exige la définition littérale. Sur la base des limitations des outils actuels pour chaque défaut de code, nous rejetons donc H_3^{regle}.

Réponse QR_2 : En analysant les règles de détection, des problèmes apparaissent pour chaque défaut de code étudié. Ces problèmes ont un impact sur les résultats de détection qui renvoient des faux positifs et des faux négatifs. L'observation des résultats montre que les défauts de code détectés ne sont pas toujours cohérents avec leur définition littérale. Pour chaque défaut de code, nous avons identifié des limitations des outils actuels dues à l'analyse statique.

4.6 Menaces à la validité

Validité interne. La principale menace pour la validité interne de cette étude pourrait être une détection imprécise des défauts comportementaux du code. Les résultats du tableau 4.5 sont également sujets à l'imprécision, même s'ils sont forcément très proches de la réalité, étant vérifiés manuellement par deux autres développeurs. La disparité dans la propagation des défauts de code comportementaux peut affecter les résultats, puisque certains défauts de code sont largement répandus, comme le défaut de code NLMR ou le défaut de code HMU, tandis que d'autres ne le sont pas, comme le défaut de code IOD. En effet, il se peut que nous ne trouvions pas les meilleurs cas d'étude dans notre échantillon de résultats qui ne sont pas répandus. De plus, le fait que l'on soit obligé de réduire le nombre d'échantillons à analyser peut également avoir un impact pour les mêmes raisons.

Validité externe. La principale menace pour la validité externe de cette étude est que notre étude ne concerne que sept défauts de code comportementaux spécifiques à Android. Sans une enquête plus approfondie, ces résultats ne devraient pas être généralisés à d'autres défauts de code ou cadres de développement. Nous encourageons donc les études futures à reproduire notre travail sur d'autres ensembles de données et avec différents défauts de code comportementaux et plateformes mobiles. Cependant, il s'agit de la première étude de ce type sur cette question. Une autre menace principale pour la validité externe est le jeu de données utilisé dans l'étude, qui est le même que celui de Habchi *et al.* (Habchi *et al.*, 2019b) d'où provient le jeu de données. Nous avons utilisé un ensemble de 318 applications Android libres d'accès provenant de F-Droid, ce qui est relativement peu. Il aurait été préférable de prendre également en compte les applications à code source fermé afin de constituer un jeu de données plus diversifié et plus représentatif. Cependant, nous n'avons pas eu accès à des logiciels propriétaires pouvant servir à

cette étude. Nous encourageons également les études futures à considérer d'autres ensembles de données d'applications libres d'accès pour étendre cette étude. Cependant, le nombre d'applications est raisonnable et cohérent par rapport à la validation des outils dans la littérature. En comparaison, aDoctor a été évalué avec 18 applications (Palomba *et al.*, 2017) et Paprika avec 106 applications (Hecht *et al.*, 2015b).

Validité de répétitivité/fiabilité. Les résultats de la validation sont répétables et fiables, puisque nous utilisons des programmes libres qui peuvent être téléchargés librement sur Internet. Les résultats sont disponibles dans nos artefacts [4].

Généralisabilité. La principale menace à la généralisabilité est que seuls deux outils sont considérés, Paprika et aDoctor. Cependant, ces deux outils sont populaires, disponibles, représentatifs des techniques utilisées dans la littérature et ont été largement étudiés par la communauté scientifique. Ils sont donc représentatifs des outils concernés par la détection des défauts du code Android à l'aide de l'analyse statique. L'utilisation d'autres outils pour une validation plus poussée devrait être envisagée.

4.7 Leçons tirées de l'étude empirique

Nous avons vu les problèmes avec les règles de détection des défauts de code comportementaux dans aDoctor et Paprika. Cependant, comme nous l'expliquons en détail dans la section 4.6, nous pouvons généraliser les résultats à d'autres outils de détection statique. Les défis rencontrés sont liés aux trois catégories de défauts de code comportementaux identifiés dans le chapitre 3. Les défis et les défauts de code comportementaux étudiés qui leur sont associés sont rattachés aux catégories

4. `https://figshare.com/s/8235a0575ff4a88f0deb`

de défauts de code comportementaux décrits dans le chapitre 3. Par une analyse statique, nous pouvons difficilement détecter les défauts de code caractérisés par l'utilisation d'un appel de méthode ou d'une séquence d'appels de méthode, la première catégorie, tels que le défaut de code DW où la méthode *acquire* doit être suivie de la méthode *release*. Il est également difficile de détecter les défauts de code caractérisés par des informations d'exécution, la deuxième catégorie. Par exemple, les défauts de code HAS, HSS et HBR où l'exécution des méthodes ne doit pas être trop longue ou bloquante. C'est également le cas pour le défaut de code IOD où le temps d'exécution doit être court et le défaut de code NLMR où la mémoire doit être libérée. Enfin, il est difficile de détecter les défauts du code caractérisés par des variations indésirables des données pendant l'exécution, la troisième catégorie. Par exemple, pour HMU, la taille d'une *ArrayMap* ne doit pas devenir excessivement grande et *HashMap* ne doit pas être utilisé pour les structures courtes. Les résultats peuvent donc être généralisés à tous les défauts comportementaux du code qui entrent dans l'une de ces trois catégories.

Pour l'instant, nous avons vu les limites de la détection statique pour les défauts du code qui présentent un fort aspect comportemental. Il reste que plusieurs autres approches peuvent être utilisées pour prendre en compte le comportement de l'application, comme la modélisation du comportement de l'application, la vérification de modèles ou la mise en place d'une analyse dynamique. Nous allons explorer les solutions par l'analyse dynamique.

Tout d'abord, pour chaque défaut de code comportemental abordé dans ce document, nous montrons comment une telle détection dynamique peut être effectuée. Ensuite, nous discutons et examinons plus en détail les défauts de code de l'état de l'art qui sont comportementaux et qui pourraient utiliser une analyse dynamique.

4.7.1 Recommandations sur les défauts de code étudiés

À partir de l'analyse et des différents retours des exemples d'applications réelles, nous montrons comment les différents défauts de code comportementaux étudiés pourraient bénéficier d'une analyse dynamique.

DW : Avec une analyse statique, il est assez difficile de savoir si les deux méthodes, *acquire* et *release*, sont appelées pendant l'exécution. Observer la trace d'exécution par une analyse dynamique permettrait de voir si ces deux méthodes sont correctement appelées.

NLMR : Ici, l'objectif est de déterminer, lorsque l'application est en arrière-plan ou lorsqu'elle commence à manquer de mémoire, si elle libère les ressources inutiles. Une analyse dynamique permettrait de s'assurer que la mémoire est réellement libérée pendant l'exécution de l'application. Une telle analyse dynamique viendrait compléter une analyse statique, qui permet déjà de détecter les classes qui n'ont pas défini la méthode.

HMU : Conformément à la définition initiale, il est préférable d'utiliser une *SimpleArrayMap* ou une *ArrayMap* pour les petits ensembles jusqu'à des centaines d'éléments, et d'utiliser une *HashMap* pour les ensembles plus grands. Une analyse dynamique consiste à vérifier deux assertions en suivant l'évolution du contenu des instances de ces structures. Premièrement, si une *SimpleArrayMap* à un moment donné dépasse la limite donnée par les recommandations d'Android de centaines d'éléments, alors nous signalons un défaut de code. Deuxièmement, si pendant une exécution entière de l'application une *HashMap* est inférieure à cette limite,

alors un défaut de code est relevé.

HBR, HAS and HSS Ces défauts de code, qui sont très similaires, indiquent que nous ne nous attendons pas à ce que ces méthodes soient trop longues ou bloquantes, puisqu'elles s'exécutent sur le processus principal et non sur un processus dédié. Plutôt que d'examiner statiquement si elles semblent trop longues en utilisant des métriques telles que le nombre d'instructions, une analyse dynamique serait plus précise pour déterminer si une méthode est effectivement trop longue pendant l'exécution de l'application.

IOD : Que la méthode *onDraw* soit chronophage ou soit trop gourmande en mémoire, il est complexe de le déterminer sans prendre en compte l'exécution de l'application. Une analyse statique peut simplement donner des indications en examinant des initialisations ou des opérations spécifiques. Une analyse dynamique pourrait indiquer, en observant l'exécution de l'application, si la méthode *onDraw* est trop gourmande en temps ou en mémoire. La recherche d'allocations de mémoire par initialisation est cependant possible par une analyse statique en recherchant les instructions impliquant une allocation de mémoire.

4.7.2 Défauts de code de l'état de l'art

Les défauts du code détectés par Paprika offrent généralement une plus grande opportunité de détection dynamique. Cela est dû à la nature des défauts traités par Paprika, avec des défauts ayant un impact sur les performances, l'affichage graphique, la mémoire et le blocage des processus. Ces types de défauts ont un aspect comportemental clair, ce qui suggère que l'inspection de l'exécution des applications conduira à une meilleure détection.

TABLEAU 4.6 Défauts de code qui pourraient utiliser ou non une analyse dynamique pour la détection.

Outil	Pourrait utiliser	N'a pas besoin
PAPRIKA (13)	**7**(**55**%) NLMR, HMU, IOD, HAS, HSS, HBR, UIO	**6**(**45**%) MIM, LIC, UCS, UHA, IGS, IWR
ADOCTOR (15)	**5**(**33**%) DW, NLMR, DTWC, UC, LT	**10**(**67**%) DR, IDFP, IDS, ISQLQ, LIC, MIM, PD, RAM, SL, IGS
Total (28)	12(43%)	16(57%)

Quant à l'outil ADOCTOR, il se concentre sur différents types de défauts de code qui sont généralement liés aux bonnes utilisations des méthodes spécifiques à Android. Pourtant, certains défauts de code pourraient néanmoins bénéficier d'une détection dynamique, mais la plupart d'entre eux n'en ont pas besoin, puisqu'il s'agit souvent de vérifier la présence de méthodes ou de remplacer certaines méthodes par une autre. Nous rappelons que PAPRIKA et ADOCTOR ont quatre défauts de code en commun : NLMR, MIM, LIC et IGS. L'analyse statique et l'analyse dynamique ne sont pas exclusives, les deux méthodes peuvent être combinées pour détecter les défauts du code.

Le tableau 4.7.2 donne le nombre de défauts de code détectés par les deux outils qui pourraient utiliser ou non des techniques dynamiques pour une détection plus proche de la réalité. Ces résultats sont obtenus par une étude de la définition des défauts de code, parmi ceux pour lesquels le comportement de l'application est à prendre en considération pour savoir si le défaut de code est bien respecté. Surtout, la nature des résultats retournés est prise en considération, ainsi que la mesure dans laquelle les instances détectées doivent être validées. De nombreux défauts

de code (57%) ne sont pas adaptés aux techniques dynamiques en raison de leur définition consistant à vérifier la présence d'une méthode ou d'un attribut. Par exemple, Unsupported Hardware Acceleration (UHA) est un défaut de code qui suggère d'éviter la méthode *drawPath* de la classe Android *Canvas*, en conseillant de la remplacer par de multiples appels à la méthode *drawLine*. Une telle description ne nécessite qu'une analyse statique. Les défauts du code qui ne nécessitent pas d'analyse dynamique sont MIM, LIC, UCS, IGS, IWR et UHA pour Paprika et DR, IDFP, IDS, IGS, ISQLQ, LIC, MIM, PD, RAM et SL pour aDoctor. Par ailleurs, près de 43% des défauts de code semblent avoir une détection insuffisante avec les méthodes actuelles en raison de leur nature dynamique, comme les sept défauts du code étudiés dans ce document. En plus des défauts de code étudiés dans cet article, les défauts de code qui pourraient utiliser une analyse dynamique sont UIO pour Paprika, et DTWC, UC et LT pour aDoctor. Les entrées de ce tableau sont corrélées avec celles du tableau 4.1.

Nous décidons de mettre de côté les défauts de code décrits comme étant des défauts de code de sécurité (Ghafari *et al.*, 2017). Même s'il y a de nombreux défauts de code de sécurité définis, nous nous concentrons sur les défauts impactant la qualité de l'application plutôt que la sécurité. En effet, les défauts de code de sécurité du code consistent généralement en l'utilisation d'une méthode/structure/classe plus sûre, ainsi qu'en la vérification de la présence d'un paramètre ou d'une variable. La trivialité de ces défauts de code ne semble pas convenir aux techniques dynamiques.

L'analyse dynamique permet plusieurs choses sur la détection de défauts de code comportementaux. Tout d'abord, prendre en compte l'aspect dynamique permet d'extraire de nouveaux défauts de code comportementaux qui n'étaient pas catégorisés auparavant. En particulier, les défauts de code concernant la mémoire, le temps d'exécution, le blocage de processus. En effet, les autres classifications des

défauts de code spécifiques à Android dans l'état de l'art ne décrivaient pas les défauts de code qu'ils ne pouvaient pas du tout détecter.

Bien que l'analyse dynamique soit plus précise que l'analyse statique, elle nécessite l'exécution de l'application. Cependant, nous devons tenir compte du fait que, dans de nombreux cas, les développeurs ne disposent pas de scénarios d'exécutions prêts à l'emploi. L'utilisation de l'analyse dynamique peut prendre plus de temps et nécessite également des artefacts qui ne sont pas toujours très courants dans les applications mobiles, comme les scénarios de test.

Enfin, certains défauts de code, tels que HMU et HSS, définissent certains seuils dans leurs règles de détection. Par exemple, PAPRIKA détecte le défaut de code HSS en calculant deux métriques, le nombre d'instructions et la complexité cyclomatique. Il définit un seuil pour chacune d'entre elles. Chaque méthode ayant un nombre d'instructions supérieur à 17 ou une complexité cyclomatique supérieure à 3,5 est détectée comme un défaut de code. Grâce à l'analyse dynamique, des métriques plus appropriées peuvent être appliquées. Pour le défaut de code HSS, le temps d'exécution peut être pris en considération. Les détecter dynamiquement serait encore difficile, puisque nous devons toujours définir un seuil, comme le seuil du temps d'exécution pour le défaut de code HSS. Cependant, une analyse dynamique permettrait de définir des métriques et des seuils plus appropriés, ce qui conduirait à une meilleure détection des défauts de code comportementaux.

4.8 Conclusion de l'étude empirique

Nous avons présenté une étude empirique sur la détection des défauts de code comportementaux dans les applications mobiles. Comme cette étude est la première du genre sur la détection des défauts de code comportementaux, elle est encore menée à petite échelle, puisqu'elle nécessite un travail manuel important.

Tout d'abord, nous avons analysé et comparé l'approche de PAPRIKA et de ADOCTOR, tous deux basés sur l'analyse statique. Nous nous sommes concentrés uniquement sur les défauts spécifiques du code concernant le comportement de l'application. Nous avons montré que les outils retournent des faux négatifs et des faux positifs pour ces défauts de code comportementaux, ce qui affecte l'efficacité des outils.

Nous avons par la suite montré par le biais d'une étude qualitative sur les défauts de code comportementaux que certaines applications qui n'étaient pas signalées comme ayant des défauts ne répondaient pourtant pas à la définition littérale des défauts. Une partie non négligeable des défauts de code détectés sont des faux positifs, c'est-à-dire des défauts de code qui n'auraient pas dû être détectés.

Ensuite, nous avons étudié les règles de détection afin de découvrir les problèmes à l'origine de ces erreurs de détection. De plus, nous avons discuté des impacts de ces règles de détection sur les résultats. Il s'agit de la première étude préliminaire de ce type sur ce sujet, afin de sensibiliser la communauté à fournir des outils de détection plus adaptés pour traiter les défauts de code comportementaux. Il est essentiel d'éviter les faux positifs et les faux négatifs pour aider au maximum le développeur dans le développement de l'application.

Enfin, nous mettons en avant la nécessité de prendre en compte le comportement dans le domaine de la détection des défauts du code Android, où de nombreuses préoccupations sont essentiellement comportementales.

CHAPITRE V

DYNAMICS : UNE APPROCHE OUTILLÉE POUR LA DÉTECTION DE DÉFAUTS DE CODE COMPORTEMENTAUX

Dans ce chapitre, nous présentons notre approche outillée qui est la contribution principale de cette thèse.

Nous présentons cette approche outillée par trois contributions visant à combler le manque d'approches dynamiques pour la détection des défauts de code comportementaux. Premièrement, nous proposons une méthode qui couvre toutes les étapes nécessaires à la spécification et à la détection des défauts de code comportementaux. Il s'agit d'une contribution majeure, puisque tous les travaux de pointe sur la détection des défauts du code Android sont uniquement statiques.

Deuxièmement, nous mettons en œuvre notre méthode à travers un outil, nommé DYNAMICS (DYNamic Analysis of Mobile app by Instrumentation for Code Smells, qui pourrait se traduire par analyse dynamique des défauts de code des applications mobiles par l'instrumentation). DYNAMICS permet de réaliser l'ensemble des étapes de manière automatique. DYNAMICS permet de détecter les défauts de code directement à partir de l'APK d'une application. Pour cela, à partir de la spécification des défauts de code, notre outil instrumente l'application, l'exécute pour produire une trace et détecte ensuite les défauts de code en fonction de cette trace. En conséquence, DYNAMICS est un outil qui permet de prendre en compte le

comportement de l'application pour assurer une détection plus précise des défauts de code comportementaux.

Troisièmement, nous validons DYNAMICS en utilisant la précision et le rappel sur un jeu de données libre d'accès de 538 applications provenant de F-DROID. De plus, nous le comparons à deux outils d'analyse statique, ADOCTOR et PAPRIKA, pour la détection de défauts de code comportementaux issus de la littérature. Nos résultats montrent l'efficacité de notre méthode pour la détection des défauts de code comportementaux, en soulignant que de nombreux cas de défauts de code qui ne peuvent pas être détectés avec des approches statiques sont effectivement détectés avec notre outil.

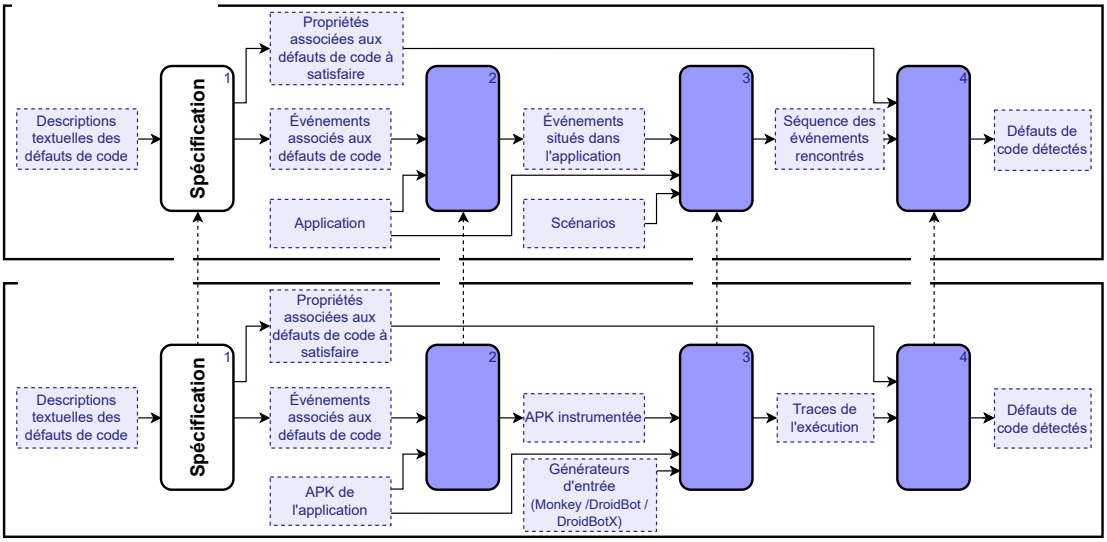
Ce chapitre est organisé de la façon suivante. La section 1 présente notre méthode de détection des défauts de code comportementaux. DYNAMICS est présenté étape par étape dans la section 2. La validation et les résultats de DYNAMICS sont présentés dans la section 3.

5.1 Méthode DYNAMICS et l'outil DYNAMICS

Plusieurs travaux antérieurs montrent que de nombreux défauts de code peuvent être détectés par l'analyse statique, y compris les défauts de code comportementaux. Cependant, nous allons montrer qu'en considérant le comportement de l'application à travers une analyse dynamique, ces défauts de code comportementaux peuvent être détectés avec plus de précision.

Dans un premier temps, nous proposons une méthode, la méthode DYNAMICS, couvrant toutes les étapes essentielles pour la détection des défauts de code comportementaux en considérant le comportement de l'application. Notre méthode est basée sur les quatre étapes séquentielles décrites dans la figure 5.1 : *Spécification*, *Traitement*, *Exécution* et *Détection*. Les étapes sont résumées comme suit :

FIGURE 5.1 Processus de la méthode DYNAMICS et de l'outil DYNAMICS. Les boites représentent les étapes, les flèches relient les entrées et les sorties de chaque étape décrite par les boites en pointillés. Les boites blanches représentent les étapes manuelles et les boites remplies les étapes entièrement automatisées.



— **Étape 1. Spécification :** Pour chaque défaut de code, nous déterminons, à partir de sa description textuelle, les événements et les propriétés associés. Dans ce contexte, un *événement* est une instruction ou un appel de méthode spécifique associé au comportement du défaut de code. Une *propriété* est une condition sur les événements qui détermine, lorsqu'elle est satisfaite, la présence du défaut de code. Les défauts de code comportementaux sont donc représentés par une série de propriétés sur des événements qui décrivent des comportements inappropriés de l'application mobile. Nous associons donc pour chaque défaut de code une propriété à vérifier de telle sorte que dès que cette propriété est vérifiée, le défaut de code est détecté.

— **Étape 2. Traitement :** Les événements associés aux défauts de code spécifiés lors de la première étape sont localisés dans l'application pour

permettre leur traçage et leur utilisation dans les étapes suivantes.

— **Étape 3. Exécution :** L'application contenant les événements localisés est exécutée selon des scénarios d'exécution et tous les événements associés aux défauts de code rencontrés sont enregistrés. Il est également possible de simuler le comportement de l'application au lieu de l'exécuter concrètement.

— **Étape 4. Détection :** La détection des défauts de code comportementaux est effectuée en analysant la séquence des événements rencontrés lors de l'exécution. Les défauts de code, dont la propriété associée est satisfaite, seront détectés.

La première étape est effectuée une fois pour tous les défauts du code, tandis que les autres étapes doivent être répétées pour chaque nouvelle application considérée. La troisième étape peut être répétée plusieurs fois si on désire plusieurs exécutions.

En deuxième lieu, nous avons implémenté concrètement la méthode DYNAMICS dans un outil, appelé l'outil DYNAMICS. La figure 5.1 présente un aperçu des quatre étapes de l'outil DYNAMICS : *Spécification*, *Instrumentation*, *Exécution*, *Détection*. Ces étapes sont des instances des étapes de la méthode DYNAMICS. Elle met également l'accent sur les étapes, les entrées et les sorties spécifiques à l'outil DYNAMICS. Les éléments suivants résument les étapes de l'outil DYNAMICS :

— **Étape 1. Spécification :** Comme détaillé dans la méthode DYNAMICS, nous définissons pour chaque défaut de code un ensemble d'événements associés et une propriété. Les événements associés à un défaut de code représentent les caractéristiques du code source qui permettent d'identifier ce défaut de code dans l'APK. De plus, nous exprimons les propriétés sous forme de propriétés formelles de la *Logique Temporelle Linéaire* (LTL) dont

la satisfaction permet la détection du défaut de code.

— **Étape 2. Instrumentation :** Pour chaque défaut de code, nous identifions les caractéristiques du code source représentées par les événements associés présents dans l'APK. Pour chacun de ces événements, nous instrumentons l'application pour insérer des instructions dans l'APK afin de générer des instructions de journalisation spécifiques. Cette étape a été développée à l'aide de la structure logicielle SOOT (Vallée-Rai *et al.*, 1999).

— **Étape 3. Exécution :** Nous exécutons l'application instrumentée automatiquement par émulation en utilisant des générateurs d'entrées pour surveiller ses événements. Au fur et à mesure que les événements sont rencontrés, plusieurs sorties de journal sont produites, et collectivement ces sorties de journal forment une trace. Cette étape a été développée en utilisant trois générateurs d'entrée : MONKEYRUNNER [1], DROIDBOT (Li *et al.*, 2017) et DROIDBOTX (Yasin *et al.*, 2021).

— **Étape 4. Détection :** À partir de la trace complète représentant tous les événements rencontrés dans leur ordre précis, nous vérifions si les propriétés LTL sont satisfaites ou non. Chaque propriété satisfaite conduit à la détection d'un défaut du code. Cette étape a été développée à l'aide de BEEPBEEP3 (Hallé et Khoury, 2017).

L'outil DYNAMICS effectue donc une analyse hors ligne sur les traces générées, puisque l'analyse est effectuée une fois l'exécution terminée. On aurait pu envisager une analyse à la volée où la détection se fait pendant l'exécution. Cependant, ce choix d'une analyse hors ligne a été fait pour interférer le moins possible avec l'exécution de l'application et pour réduire au maximum l'impact sur la performance et l'empreinte mémoire. Ceci est d'une importance capitale puisque nous considérons certains défauts de code liés à l'exécution et à l'usage de la mémoire.

1. `https://developer.android.com/studio/test/monkeyrunner`

5.2 L'outil DYNAMICS en détail

Dans ce qui suit, les quatre étapes de l'outil DYNAMICS sont décrites en utilisant un modèle commun : entrée, sortie, description et mise en oeuvre. Les étapes sont, en outre, illustrées par un exemple concret en utilisant le défaut de code DW.

5.2.1 Étape 1. Spécification

Entrée : Description textuelle des défauts de code spécifiques à Android à partir de la littérature.
Sortie : Les événements et la propriété LTL associés à chaque défaut de code. Un *événement* est une instruction ou un appel de méthode spécifique associé au comportement du défaut de code. La *propriété* est une condition sur les événements qui détermine, lorsqu'elle est satisfaite, la présence du défaut du code.

Description : Nous identifions les événements associés aux défauts de code directement à partir de la définition du défaut de code. Sur la base de ces événements, nous spécifions les propriétés associées aux défauts de code.

Chaque événement est décrit par des caractéristiques du code source (appels/-déclarations de méthodes spécifiques ou utilisation de structures de code spécifiques, voir le tableau 5.1). Des valeurs associées sont attachées à un événement en fonction des caractéristiques du code source. Ces valeurs peuvent être des horodatages, des mesures de mémoire ou des ID Java d'objets et de structures, voir le tableau 5.1.

Nous définissons les propriétés en utilisant la logique temporelle linéaire LTL, une

TABLEAU 5.1 Liste des événements associés aux défauts de code.

Défaut de code	**Événement**	**Caractéristiques du code source**	**Valeurs**
DW	Acquire	WakeLock.acquire()	ID WakeLock
	Release	WakeLock.release()	
HMU	Instantiation	map = new HashMap map = new ArrayMap map = new SimpleArrayMap	ID Structure Taille Type
	Addition	map.put(...) map.putAll(...)	
	Deletion	map.remove(...)	
	Clear	map.clear()	
HAS **HBR** **HSS**	Begin	Début de la méthode onPreExecute / onPostExecute / onProgressUpdate / onReceive / onStartCommand	ID Appel de méthode Horodatage
	End	Fin de la méthode onPreExecute / onPostExecute / onProgressUpdate / onReceive / onStartCommand	
IOD	Begin	Début de la méthode onDraw	ID Appel de méthode Horodatage
	End	Fin de la méthode onDraw	
	New	Instantiation (<init>) dans la méthode onDraw	ID Appel de méthode
NLMR	Begin	Début de la méthode onLowMemory / onTrimMemory	ID Appel de méthode Quantité de mémoire
	End	Fin de la méthode onLowMemory / onTrimMemory	

logique conçue pour exprimer des conditions sur des séquences. Dans notre cas, les séquences sont des séquences d'événements associés au défaut de code qui apparaissent dans la trace d'exécution. Brièvement, LTL est une logique temporelle propositionnelle et modale développée à l'origine pour la vérification des systèmes réactifs (Pnueli, 1977). Elle augmente la logique propositionnelle avec les modalités F(Éventuellement), G(Toujours), X(Suivant) et U(Jusqu'à) afin d'exprimer des énoncés tels que "Une structure sera *toujours* petite" ou "Une méthode sera *éventuellement* appelée". Ces déclarations peuvent être combinées au moyen de connecteurs logiques et de l'imbrication d'opérateurs modaux pour fournir des propriétés plus complexes. La syntaxe est naturelle et simple et, en tant que langage formel, elle possède une sémantique bien définie, elle est donc interprétable sans ambiguïté.

Chaque propriété est, en outre, paramétrée par une variable x. Cette variable x fait référence soit à l'instance d'un objet, soit à l'appel d'une méthode. Par exemple, pour le défaut de code DW, le *WakeLock* d'identifiant x aura la propriété φ_x.

Dans les propriétés suivantes, on utilise fréquemment l'opérateur jusqu'à (U). Lorsque deux événements, disons *acquire* et *release*, se produisent alternativement, nous sommes généralement intéressés, pour un événement *acquire*, par son événement *release* correspondant. Cet événement *release* se produit évidemment après le *acquire*, mais, en outre, il n'y a pas d'autres *acquire* avant que ce *release* ne se produise. Ceci est formellement exprimé par la condition $X(\neg acquire\ U\ release)$ qui exprime que, à partir de l'événement suivant de la séquence, il n'y a pas d'événement *acquire* jusqu'à ce qu'un événement *release* se produise. Nous utiliserons cette formulation pour les événements *acquire*/*release*, mais aussi pour les événements *begin*/*end*.

Il est important de préciser que pour que certaines propriétés LTL puissent donner

un résultat, les exécutions ont une fin. La fin peut être provoquée par une durée précisée, parce que l'application a cessé de fonctionner ou parce que l'utilisateur a fermé complètement l'application.

Exemple concret :

DW :

Événements : Les événements sont provoqués par des appels aux méthodes *acquire* et *release* de la classe *WakeLock*. La valeur associée est un nombre entier indiquant l'identifiant de l'instance *WakeLock*. Les événements *Acquire* et *Release* sont associés à une instance *WakeLock* unique.

Propriété : Un défaut de code est détecté chaque fois qu'un *acquire* sur un *WakeLock* est rencontré, mais qu'il n'y a pas de *release* associé.
"Éventuellement (c'est-à-dire à un moment donné de la séquence), le *WakeLock* x est acquis et ce n'est pas le cas que sa libération correspondante se produit "
Formule LTL pour un *WakeLock* d'identifiant x :

$$\varphi_x = \boldsymbol{F}(acquire_x \wedge \neg\ (X\ (\neg acquire_x\ U\ release_x))$$

Spécification des autres défauts de code

Nous détaillons maintenant, pour chaque autre défaut de code détecté par l'outil DYNAMICS, leurs événements et leur propriété.

HMU :

Événements : Les événements sont provoqués par des appels aux méthodes des classes *HashMap*, *SimpleArrayMap* et *ArrayMap* influençant la taille des structures : *new*, *put*, *putAll*, *remove*, *clear*. Les valeurs associées à chaque événement sont le type de la structure, c'est-à-dire sa classe, sa taille réelle et son identifiant.

Propriété : Un défaut de code est détecté si une *SimpleArrayMap*/*ArrayMap* atteint une grande taille ou si une *HashMap* n'atteint jamais une grande taille. Les structures contenant jusqu'à plusieurs centaines d'éléments sont considérées comme de grandes tailles [2]. Chaque ensemble de valeurs dont la taille est supérieure ou inférieure à 500 éléments selon le type identifie ainsi un défaut.
"Éventuellement (c'est-à-dire à un moment donné de la séquence), la structure est soit une *SimpleArrayMap* ou une *ArrayMap* et sa taille est supérieure ou égale à 500."

$$\varphi_{1x} = \boldsymbol{F}((type(x) = SimpleArrayMap \vee type(x) = ArrayMap) \wedge size(x) \geq 500)$$

"Globalement (c'est-à-dire pour tous les événements de la séquence), si la structure est une *HashMap*, sa taille est inférieure ou égale à 500."

$type(x)$ retourne le type de la structure d'ID x et $size(x)$ retourne la taille de la structure d'ID x.

$$\varphi_{2x} = \boldsymbol{G}(type(x) = HashMap \rightarrow size(x) \leq 500)$$

Comme il y a deux cas à vérifier, la formule LTL φ_x pour la structure x est la disjonction ("ou" logique) des deux formules LTL précédentes

$$\varphi_x = \varphi_{1x} \ \vee \ \varphi_{2x}$$

HAS/HBR/HSS :

Événements : Les événements sont provoqués par des appels à des méthodes

2. `https://developer.android.com/reference/android/support/v4/util/ArrayMap.html`

spécifiques au défaut de code : *onPostExecute*, *onPreExecute* et *onProgressUpdate* pour HAS, *onReceive* pour HBR, *onStartCommand* pour HSS. Cependant, pour toutes les méthodes, il existe des événements *Begin* et *End* ajoutés par l'instrumentation, respectivement au début et à la fin de l'appel de la méthode. Chaque événement est associé à un horodatage et à l'identifiant de l'appel de la méthode.
Propriété : Un défaut de code est détecté chaque fois qu'une méthode associée à HAS, HSS ou HBR est rencontrée avec un temps d'exécution supérieur à 100 ms, ce qui correspond à la différence entre les horodatages des événements *Begin* et *End*.

"Éventuellement, le début de l'appel de la méthode x est atteint et à la fin de cet appel de méthode, le temps écoulé est supérieur à 100ms."
Formule LTL pour un appel de méthode d'identifiant x :

$$\varphi_x = \boldsymbol{F}(begin_x \wedge (X\ (\neg begin_x\ U\ (end_x \wedge \\ (endTime(x) - beginTime(x) > 100ms)))))$$

$beginTime(x)$ retourne l'horodatage du début de l'exécution de la méthode d'ID x et $endTime(x)$ retourne l'horodatage de la fin de l'exécution de la méthode d'ID x.

IOD :

Événements : Les événements sont provoqués par des appels à la méthode *onDraw*. Là encore, les événements *Begin* et *End* sont présents au début et à la fin de l'appel de la méthode. Il existe en outre un événement *New* supplémentaire provoqué par l'instanciation de nouveaux objets. Chaque événement est associé à l'identifiant de l'appel de méthode. Les événements *Begin* et *End* sont associés à une valeur d'horodatage.

Propriété : Un défaut de code est détecté chaque fois qu'une méthode *onDraw* est rencontrée avec un temps d'exécution supérieur à 1/60 de seconde, ou si nous rencontrons une instanciation. La différence entre les valeurs d'horodatage des événements *End* et *Begin* est utilisée pour calculer le temps d'exécution de la méthode. "Éventuellement, le début de l'appel de méthode x est atteint et à la fin de cet appel de méthode, le temps écoulé est supérieur à 1/60s, ou le début de l'appel de méthode x est atteint et il n'y a pas d'instanciation avant la fin de l'appel"

Formule LTL pour un appel de méthode d'identifiant x :

$$\begin{aligned}\varphi_x = &\boldsymbol{F}(begin_x \wedge (X(\ \neg begin_x\ U\ (end_x \\ &\wedge (endTime(x) - beginTime(x) > 1/60s))))) \vee \\ &\boldsymbol{F}(begin_x \wedge \neg((\neg new_x)\ U\ end_x)\end{aligned}$$

NLMR :

Événements : Les événements sont provoqués par l'exécution des méthodes *onLowMemory* et *onTrimMemory*. Comme précédemment, il existe des événements *Begin* et *End* au début et à la fin de la méthode. Les événements *Begin* et *End* sont associés à la quantité de mémoire utilisée à ce moment-là et à l'identifiant de l'appel de la méthode.

Propriété : Un défaut de code est détecté lorsqu'une méthode *onLowMemory* ou *onTrimMemory* est rencontrée et que la mémoire libérée lors de cet appel de méthode est inférieure à 1024KB. La différence entre les valeurs de mémoire des événements *Begin* et *End* est utilisée pour calculer la quantité de mémoire libérée pendant l'exécution de la méthode. Il est également détecté si une *Activité* est présente et ne définit pas une méthode *onLowMemory* ou *onTrimMemory*. Cette deuxième partie de la règle de détection est détectée de manière statique et n'est pas traitée par la formule LTL.

"Éventuellement, le début de l'appel de la méthode x est atteint et à la fin correspondante, la mémoire libérée pendant l'appel est inférieure à 1024KB"
Formule LTL pour un appel de méthode d'identifiant x :

$$\varphi_x = \boldsymbol{F}(begin_x \wedge (X\ \neg begin_x\ U\ (end_x \wedge \\ (memoryEnd(x) - memoryBegin(x) < 1024))))$$

$memoryBegin(x)$ retourne la mémoire utilisée par l'application au début de l'exécution de la méthode d'ID x et $memoryEnd(x)$ retourne la mémoire utilisée par l'application à la fin de l'exécution de la méthode d'ID x.

Implémentation :

Il n'y a pas d'implémentation pour cette étape. La spécification, c'est-à-dire les définitions des événements et des propriétés, est définie manuellement en fonction de la description du défaut de code.

5.2.2 Étape 2. Instrumentation

Entrée : L'APK à instrumenter et les événements associés aux défauts de code.
Sortie : L'APK instrumentée avec les instructions de journalisation.

Description : Dans cette étape, l'objectif est d'instrumenter l'APK d'une application mobile en ajoutant des instructions de journalisation. Concrètement, l'instrumentation consiste à identifier les événements pertinents situés dans l'APK et à insérer une instruction qui produira une sortie de journal spécifique pour chaque événement. Ainsi, lorsque l'application instrumentée est exécutée, une trace composée d'une séquence de sorties de journal sera générée. Dans notre contexte, une sortie de journal est un quadruplet :

(*localisation*, *identifiant*, *événement*, *valeurs*)

Ce quadruplet correspond à :

— *localisation* est le nom du package, de la classe et de la méthode où l'événement se produit ;
— *identifiant* est un identifiant séquentiel qui distingue plusieurs occurrences du même événement dans la même méthode de la même classe ;
— *événement* est le mot-clé associé à un événement. Les événements sont listés dans le tableau 5.1 ;
— *valeurs* contient les valeurs utilisées pour les propriétés LTL de la détection des défauts du code. Les valeurs peuvent être trouvées dans le tableau 5.1 ;

De plus, l'instrumentation diffère en fonction de la catégorie (les catégories sont détaillées au chapitre 3) à laquelle appartient le défaut de code :

— Les défauts de code concernant la mauvaise utilisation d'un appel de méthode ou d'une séquence d'appels de méthode. L'instrumentation se fait en insérant une instruction pour chaque appel des méthodes concernées. Un seul défaut de code détecté par DYNAMICS appartient à cette catégorie : *DW*.
— Les défauts de code concernant les problèmes d'exécution d'une méthode. Pour ces défauts de code, l'instrumentation se fait en insérant une instruction au début et à la fin de la déclaration de la méthode concernée. Cinq défauts de code détectés par DYNAMICS appartiennent à cette catégorie : *HAS*, *HSS*, *HBR*, *IOD* et *NLMR*. Pour ces défauts de code, les événements associés contiennent un identifiant d'appel de méthode. Cet identifiant d'appel de méthode est déterminé à partir de la localisation du code et détermine l'identifiant dans le quadruplet de la journalisation.
— Les défauts de code concernant les variations non souhaitées de données d'un objet au cours de son exécution. L'instrumentation se fait en insérant une instruction pour chaque appel d'une méthode qui a un impact sur les

données des objets concernés. Un seul défaut de code dans cette thèse appartient à cette catégorie : *HMU*.

Implémentation : Cette étape nécessite une analyse statique afin d'instrumenter le code. Pour ce faire, nous avons développé un module Java, à l'étape 2 de l'outil DYNAMICS, en utilisant la structure logicielle SOOT (Vallée-Rai *et al.*, 1999) et son module DEXPLER (Bartel *et al.*, 2012) pour analyser les artefacts APK. Les fichiers APK sont des archives ZIP contenant diverses informations, comme le manifeste AndroidManifest.xml définissant toutes les métadonnées de l'application et un fichier .dex contenant toutes les classes de l'application compilées dans un fichier au format dex [3]. Le bytecode dex est basé sur les registres, ce qui signifie que sa traduction en Java ou en langage intermédiaire implique une importante perte d'informations. Cela est dû au fait que le type et le nom des variables locales peuvent être plus difficiles à retrouver, et que les branches (for, while, if, ...) sont remplacées par des goto. SOOT convertit le bytecode des APK en une représentation interne SOOT, qui est similaire au langage Java. DYNAMICS parcourt la représentation interne de SOOT pour chaque classe, chaque méthode et chaque instruction du corps de ces méthodes. Ces instructions sont converties en Jimple, une version simplifiée du code source Java. Ces instructions sont analysées pour détecter si un événement est présent. Si un tel événement est présent, les instructions permettant la journalisation sont insérées directement après l'événement dans l'APK. Enfin, une petite tâche indispensable, l'APK généré doit être signé afin d'être exécuté à l'étape suivante. Nous utilisons JARSIGNER [4] pour effectuer cette opération.

---

3. `https://source.android.com/devices/tech/dalvik/`

4. `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html`

Analyser directement le bytecode plutôt que le code source présente à la fois des avantages et des inconvénients. Par exemple, nous n'avons pas besoin d'avoir accès au code source ni d'effectuer d'étapes de compilation. D'autre part, nous ne pouvons pas connaître le numéro de ligne original associé à l'instruction Java.

FIGURE 5.2 Exemple d'instruction Java

```
PowerManager.WakeLock completeWakeLock;
(...)
completeWakeLock.acquire();
```

FIGURE 5.3 Sortie de journal associée à l'instruction Java.

```
lambdaapp.LockManager.java$lock:0:dwacq:145
```

Exemple concret : La figure 5.2 montre un appel à la méthode *acquire* du *WakeLock* qui verrouille le *WakeLock*. Il s'agit d'un événement *Acquire* pour le défaut de code DW. Dans ce cas, l'application instrumentée produira la sortie de journal décrite dans la figure 5.3. Cette sortie indique le nom de la méthode *lock*, le nom de la classe *LockManager* et le nom du package *lambdaapp* où cet appel a lieu. Elle montre également qu'il s'agit du premier événement de ce type dans la méthode grâce à l'identifiant 0. Enfin, elle indique qu'il s'agit d'un événement *Acquire* grâce au mot-clé *dwacq* et qu'il a opéré sur la structure d'identifiant 145.

5.2.3 Étape 3. Exécution

Entrée : L'APK instrumentée et les générateurs d'entrée.
Sortie : Les traces d'exécution obtenues après avoir exécuté l'application instrumentée avec les outils de génération d'entrées.

Description : Cette étape consiste à exécuter l'application instrumentée installée dans un appareil réel ou virtuel en utilisant différents générateurs d'entrée avec différentes configurations d'entrée (comme le temps d'exécution autorisé) pour produire des traces d'exécution. Un générateur d'entrées est un programme qui simule l'interaction de l'utilisateur en générant les entrées de l'application, par exemple en cliquant sur un bouton spécifique, en saisissant un texte dans un champ de saisie ou en revenant en arrière. Il s'agit d'une étape importante de l'outil DYNAMICS puisque ces traces permettront de détecter les défauts de code. Chaque fois que l'exécution atteint une instruction de journalisation, une sortie de journal spécifique sera produite. La trace d'exécution est composée de toutes ces sorties de journalisation. Ainsi, la trace est composée de tous les événements associés aux défauts de code rencontrés lors de l'exécution.

Le défi est naturellement d'atteindre une bonne couverture du code instrumenté pendant l'exécution pour capturer autant d'événements que possible afin de détecter autant de défauts de code que possible.

Implémentation : L'exécution peut être faite soit manuellement par un scénario d'exécution déjà fourni, soit automatiquement en utilisant un générateur d'entrée. Nous avons choisi d'utiliser des outils de génération d'entrées pour automatiser cette partie. Nous avons comparé les outils suivants : DYNODROID (Machiry *et al.*, 2013), DROIDUTAN [5], DROIDBOT (Li *et al.*, 2017), DROIDBOTX (Yasin *et al.*, 2021), HUMANOID (Li *et al.*, 2019) et MONKEYRUNNER [6]. Après quelques expériences préliminaires avec ces outils sur le même ensemble d'applications mobiles, en comparant leur capacité à provoquer les événements détectés par notre outil DYNAMICS, nous avons identifié les trois générateurs d'entrée suivants comme

5. `https://github.com/aleisalem/Droidutan`

6. `https://developer.android.com/studio/test/monkeyrunner`

étant les plus prometteurs :

— **MonkeyRunner** : Il s'agit d'un outil qui réalise des événements aléatoires sur l'interface utilisateur de l'application. Il s'agit d'un outil de test aléatoire gratuit inclus dans le SDK Android. Cet outil émule un utilisateur interagissant avec une application, générant et injectant des actions pseudo aléatoires, par exemple des clics, des balayages ou des événements système dans le flux d'entrée des événements de l'application.

— **Droidbot** : Cet outil est basé sur MonkeyRunner. Droidbot est un outil de test libre d'accès qui utilise une stratégie d'exploration basée sur un modèle dans le cadre d'une approche boîte noire. Il permet également aux utilisateurs de personnaliser leurs scripts de test en utilisant le modèle de transition d'état généré.

— **DroidbotX** : Cet outil est une extension de Droidbot qui génère des actions aléatoires basées sur la technique du Q-Learning (Kaelbling *et al.*, 1996). Cette approche sélectionne systématiquement les événements d'entrée et guide l'exploration pour exposer la fonctionnalité d'une application en cours de test afin de maximiser la couverture des instructions, méthodes et activités en minimisant les événements d'entrée redondants.

Nous utilisons ces trois générateurs d'entrées sur un émulateur pour générer les traces qui seront utilisées pour la détection. Nous avons utilisé l'outil de ligne de commande *adb*[7] pour installer l'APK. Nous avons également utilisé l'outil *logcat*[8] pour récupérer le journal de l'appareil, qui contient les traces d'exécution générées.

Exemple concret : La figure 5.4 est un extrait d'une trace générée lors d'une

7. `https://developer.android.com/studio/command-line/adb`

8. `https://developer.android.com/studio/command-line/logcat`

FIGURE 5.4 Extrait d'une trace d'exécution.

```
lambdaapp.LockManager.java$lock:0:dwacq:145
lambdaapp.LockManager.java$lock:1:dwacq:191
lambdaapp.LockManager.java$lock:2:dwacq:143
lambdaapp.LockManager.java$lock:0:dwrel:145
```

exécution. L'extrait provient des exemples donnés dans la deuxième étape. Il y a trois *WakeLocks* d'identités différentes comme le montre la trace. Un seul des *WakeLocks* est acquis puis libéré, les deux autres sont acquis, mais jamais libérés. Par conséquent, le premier, d'identifiant 145, ne signale pas la présence d'un défaut de code alors que les deux autres, d'identifiants 191 et 143, le font.

5.2.4 Étape 4. Détection

Entrée : Un ensemble de traces d'exécution et les propriétés LTL associées aux défauts du code.
Sortie : Les défauts de code détectés.

Description : Cette étape consiste à analyser les traces d'exécution résultantes en utilisant la technique de vérification à la volée (runtime verification) afin d'identifier les défauts de code qui se sont produits. Concrètement, la vérification à la volée consiste à analyser les séquences d'événements au sein des traces d'exécution et à vérifier les propriétés LTL sur ces séquences.

Comme vu dans l'étape de spécification, les propriétés sont paramétrées avec une variable x. Cette variable x fait référence soit à l'instance d'un objet, soit à l'appel d'une méthode. Seuls les événements associés à cette instance ou à cet appel de méthode doivent être utilisés pour vérifier la propriété. Cela est nécessaire pour que les propriétés ne soient pas vérifiées à l'aide d'événements provenant de différents

objets ou appels de méthode. Par exemple, pour vérifier la propriété du défaut de code DW, il ne faut pas utiliser l'événement *Acquire* d'un *WakeLock* x_1 avec l'événement *Release* d'un *WakeLock* x_2 différent. Seuls les événements concernant le *WakeLock* d'identifiant x doivent être utilisés pour vérifier la propriété φ_x.

Implémentation : Pour cette étape, nous avons implémenté un module Java qui surveille l'exécution dans notre outil en utilisant la structure logicielle BeepBeep 3 (Hallé et Khoury, 2017). BeepBeep 3 est un moteur de traitement de flux qui permet le traitement des données de journalisation. Il permet des traitements multiples sur les traces, notamment la vérification de formules LTL. Nous définissons donc une chaîne de processus BeepBeep 3 permettant de vérifier s'il s'agit bien d'une trace d'événements liée à un défaut de code. Nous spécifions une branche par défaut de code pour vérifier la propriété LTL associée. Lorsque toute la trace est traitée, nous obtenons l'ensemble des défauts de code détectés.

BeepBeep 3 offre également la possibilité de découper la trace en fonction d'un paramètre. Cela nous permet de vérifier une formule LTL pour chaque élément d'un défaut de code. Par exemple, ceci nous permet de vérifier une propriété LTL pour chaque *HashMap* de différents identifiants pour le défaut de code HMU. Ceci nous permet également de vérifier une propriété LTL pour chaque appel de méthode *onDraw* différent rencontré pour le défaut de code IOD.

Exemple concret :

La figure 5.5 montre la chaîne de processeurs nécessaire pour détecter le défaut de code DW à partir d'une trace d'exécution. Tous les défauts de code détectés par Dynamics ont été détectés par une chaîne de processeur, mais cela est si volumineux à représenter que nous avons décidé de ne représenter que DW à titre d'exemple. BeepBeep 3 est constitué de processeurs produisant une sortie en fonction d'une entrée. Les processeurs sont reliés entre eux par des tuyaux .

FIGURE 5.5 Extrait de la chaîne de processeurs BeepBeep pour la détection du défaut de code DW

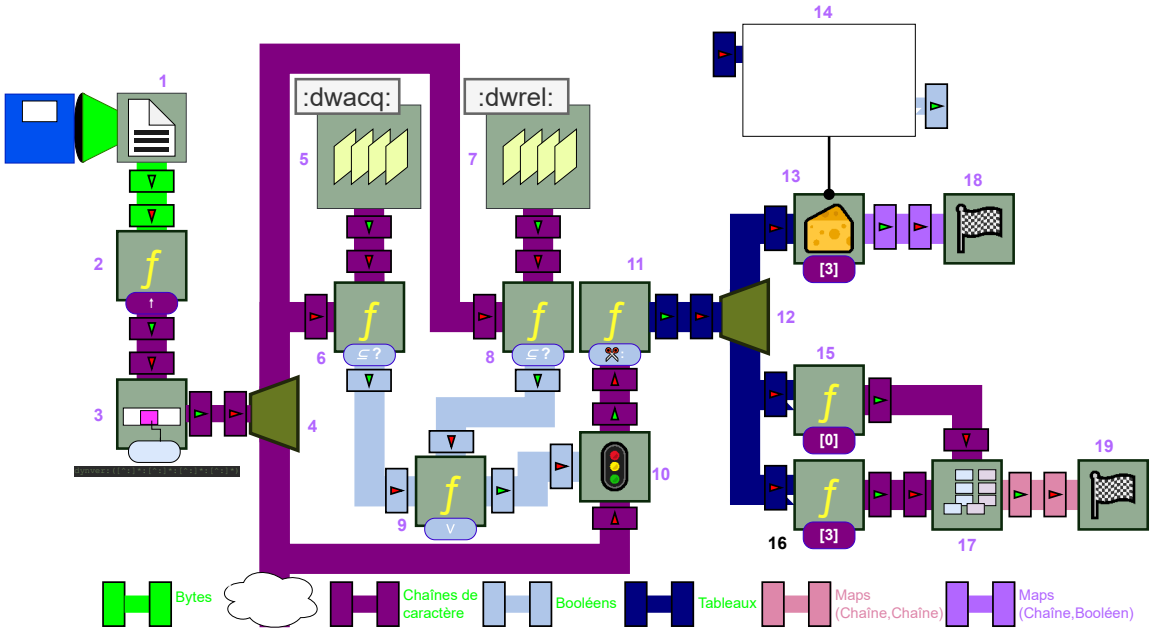
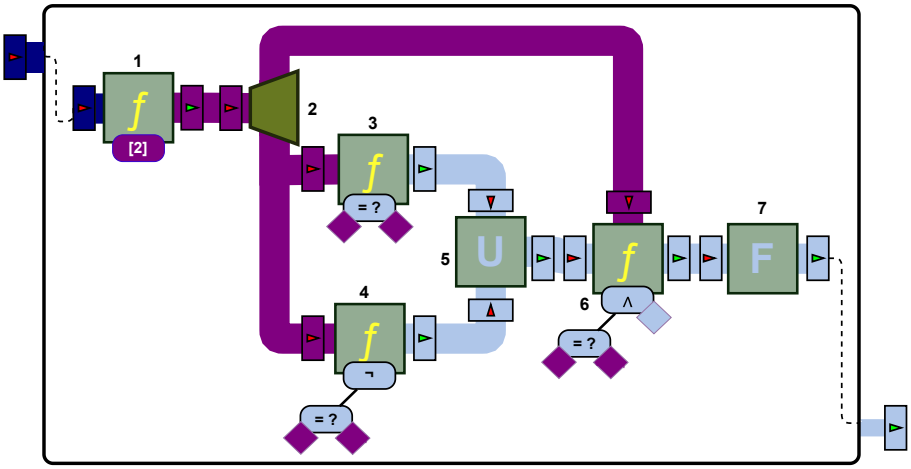


FIGURE 5.6 La chaîne de processeurs BeepBeep pour la propriété LTL du défaut de code DW



Dans la figure, un processeur est représenté par une boîte carrée, avec un pictogramme représentant le type de traitement qu'il exécute sur les événements. Sur les côtés de cette boîte se trouvent un ou plusieurs "tuyaux" représentant ses entrées et sorties. Les tuyaux d'entrée sont indiqués par un triangle rouge dirigé vers l'intérieur, tandis que les tuyaux de sortie sont représentés par un triangle vert dirigé vers l'extérieur. La couleur du tuyau indique la nature des données qui circulent dans la chaîne. Une explication détaillée sur la représentation graphique des chaînes de processeurs BeepBeep 3 se trouve sur ce site [9]. Un glossaire des différents processeurs et fonctions associées à BeepBeep 3 se trouve sur ce site [10].

Tout d'abord, le processeur 1 (*ReadLines*) lit les données ligne par ligne à partir d'un fichier, dans notre cas une trace d'exécution d'une application. Ce sont donc les lignes du fichier qui vont passer dans les tuyaux une par une. Le processeur 2 (*ToString*) le convertit ensuite en String tandis que le processeur 3 (*FindPattern*) utilise une expression régulière pour ne garder que les lignes de la forme *location :id :event :values*. Le processeur 4 est un *Fork* qui duplique le flux vers plusieurs processeurs. Les processeurs 5 (QueueSource) et 6 (fonction Contains Strings) vérifient si le flux contient l'événement *Acquire* tandis que, en parallèle, les processeurs 7 et 8 vérifient si le flux contient l'événement *Release*. Le processeur 9 (Function Or) vérifie ensuite si le flux contient un événement d'acquisition ou un événement de libération. Le processeur 10 (Filter) continuera à transmettre le flux uniquement si le booléen d'entrée est vrai, ce qui, dans notre cas, signifie que le flux continue dans cette branche uniquement s'il contient un événement *Acquire* ou *Release*, les deux événements liés au défaut de code DW. Le processeur

---

9. `https://liflab.gitbook.io/event-stream-processing-with-beepbeep-3/drawing`

10. `https://liflab.gitbook.io/event-stream-processing-with-beepbeep-3/dictionary`

11 (Function split string) divise la chaîne dans un tableau en utilisant le double point comme séparateur, et le processeur 13 (Slice) divise les événements du flux en plusieurs sous-flux selon un paramètre, dans notre cas la valeur de la sortie de journal, l'identifiant du WakeLock. Nous obtenons donc autant de sous-flux que d'identifiants de *WakeLock* rencontrés.

Le sous-flux est finalement traité par la formule LTL décrite dans la figure 5.6. La sortie est un tableau associatif contenant l'identifiant du *Wakelock* comme clé, et un booléen (vrai ou faux) indiquant si le défaut de code est présent ou non comme valeur. Les processeurs 15 et 16 (Function NthElement) sélectionnent les premier et quatrième éléments du tableau, c'est-à-dire la localisation et l'identifiant du *WakeLock* qui sont ensuite rassemblés dans un tableau associatif par le processeur 17 (PutInto (Maps)).

Les processeurs 18 et 19 (KeepLast) sont des processeurs particuliers qui renvoient simplement les derniers événements, dans notre cas les tableaux associatifs remplis. Ainsi, à la fin de ce flux, nous obtenons deux cartes, l'une reliant les localisations aux identifiants, et l'autre reliant les identifiants à la présence d'un défaut de code.

5.2.5 Discussion

Lorsque des seuils spécifiques sont utilisés, les valeurs seuils que nous utilisons proviennent soit des publications d'où proviennent les défauts de code où elles sont déjà définies, comme pour HMU ou IOD, soit d'une interprétation des publications lorsqu'elles ne fournissent qu'une description qualitative, comme "chronophage" ou "réclamant de la mémoire" pour HAS, HSS, HBR et NLMR. Pour ce dernier cas, nous utilisons une valeur qui, à notre connaissance, correspond à cette description en tenant compte du temps d'exécution et de l'empreinte mémoire rencontrés lors

de l'exécution de notre outil. Les valeurs des seuils peuvent être subjectives et sujettes à discussion, mais notre méthode est généralement applicable et notre outil peut facilement être modifié pour s'adapter à d'autres valeurs.

L'outil DYNAMICS est également extensible pour de nouveaux défauts de code. Pour détecter un nouveau défaut de code, il suffit de donner les événements associés et de créer une formule LTL pour la propriété de ce défaut de code, comme décrit dans l'étape 1 Spécification. Les autres étapes peuvent être facilement appliquées à ces nouveaux défauts de code.

5.3 Validation de DYNAMICS

Dans cette section, nous présentons l'étude qui vise à valider DYNAMICS. Nous suivons une méthodologie mixte à travers la collecte et l'analyse de données quantitatives et qualitatives.

5.3.1 Hypothèses

Nous voulons valider les trois hypothèses suivantes :

- ***H*$_1$** DYNAMICS *permet la détection des défauts de code comportementaux.* Cette hypothèse soutient l'utilité et l'efficacité de DYNAMICS pour la détection des défauts de code comportementaux.
- ***H*$_2$** DYNAMICS *permet la détection* ***précise et exacte*** *des défauts de code comportementaux.* Cette hypothèse soutient la bonne précision et le bon rappel de DYNAMICS sur les défauts de code comportementaux grâce à des méthodes de détection plus adaptées.
- ***H*$_3$** DYNAMICS *permet la détection* ***précise et exacte*** *des défauts de code comportementaux* ***grâce à sa nature dynamique****.* Grâce aux règles de détection de DYNAMICS basées sur l'analyse dynamique, nous pouvons détec-

ter des cas de défauts de code qui ne se produisent que pendant l'exécution et qui sont donc peu susceptibles d'être détectés par l'analyse statique.

5.3.2 Sujets

La validation est menée principalement sur notre outil, DYNAMICS. Ensuite, nous détectons les défauts de code sur les mêmes applications à l'aide de deux autres outils, aDOCTOR et PAPRIKA, afin de comparer les résultats. Ces deux outils, déjà présentés dans le l'étude empirique du chapitre 4, sont sélectionnés pour les mêmes raisons.

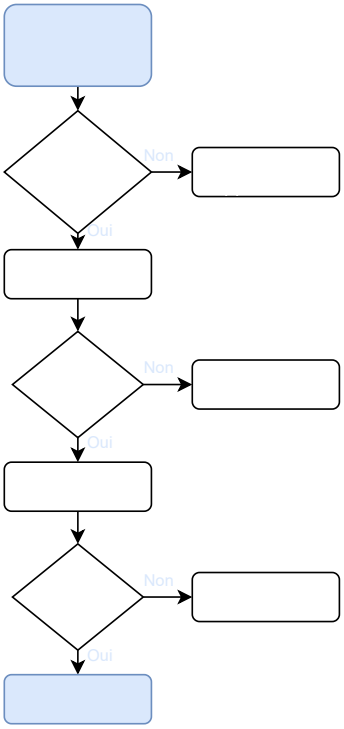
Nous utilisons DYNAMICS pour détecter sept défauts de code comportementaux spécifiques à Android : Durable WakeLock (DW), HashMapUsage (HMU), Heavy AsyncTask (HAS), Heavy BroadcastReceiver (HBR), Heavy Service Start (HSS), Init OnDraw (IOD), No Low Memory Resolver (NLMR). La définition de ces défauts de code est fournie dans le chapitre 3.

5.3.3 Objets

La validation se fonde sur 538 applications recueillies auprès de F-DROID [11]. Ce jeu de données est constitué de véritables applications libres d'accès provenant de F-DROID et publiées sur GITHUB. F-DROID fournit un jeu de données de véritables applications qui ne sont ni des applications factices, ni des modèles, ni des bibliothèques. Pour garantir la récupération des apps et de leur code source associé, nous nous assurons que les applications proviennent de GITHUB, qu'elles sont toujours disponibles, qu'elles peuvent être construites et instrumentées. Les applications sont sélectionnées de la manière suivante, comme le montre la figure 5.7.

11. https ://f-droid.org/

FIGURE 5.7 Diagramme de la sélection des applications.



F-DROID contenait au moment de l'expérimentation 4008 applications. Sur ces 4008 applications, 3034 proviennent de GITHUB, mais seulement 2974 sont encore disponibles sur GITHUB. Les APK des applications sont construites à l'aide des scripts de construction automatique fournis par les développeurs sur F-DROID. 565 applications (sur les 2974) ont pu être construites, et le reste des applications (2409 = 2974 - 565) n'ont pas pu l'être, principalement en raison de scripts non fonctionnels ou d'éléments obsolètes. Enfin, 538 (sur les 565 apps construites) ont pu être instrumentées alors que le reste des apps (27 = 565 - 538) n'ont pas pu l'être principalement en raison d'une API trop ancienne et de bibliothèques spécifiques manquantes.

Nous fournissons dans la liste complète [12] de nos artefacts, qui comprend la liste

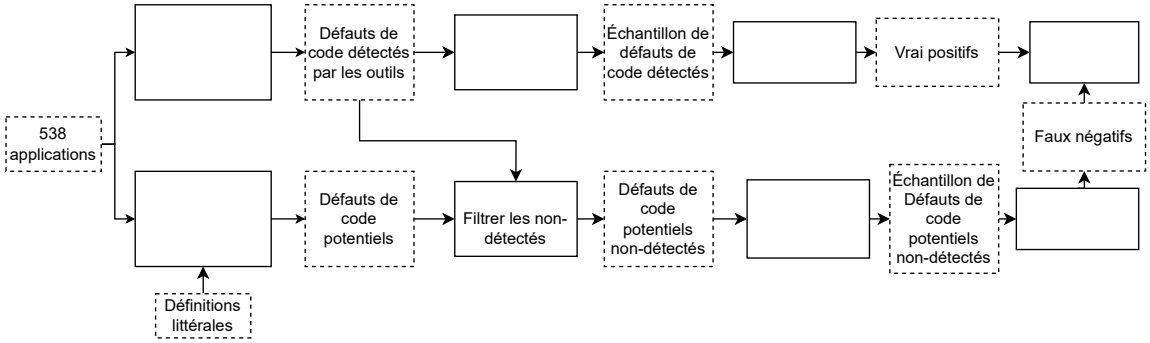
12. https://doi.org/10.21227/49vr-wr08

des APK, des APK instrumentés, des traces d'exécution, des défauts de code détectés pour chaque outil, des résultats compilés et de notre outil.

Le jeu de données diffère de l'étude empirique. Nous avons changé de jeu de données pour deux raisons, tout d'abord dans le but d'avoir un nombre d'applications plus conséquent. Enfin, le jeu de données de l'étude empirique comportait du code source qui ne fournissait pas nécessairement de scripts permettant de construire l'apk de l'application de façon automatique, contrairement à F-Droid qui fournit un script permettant la construction des apk de façon automatique pour beaucoup d'applications.

5.3.4 Processus

Figure 5.8 Vue d'ensemble du processus de validation de Dynamics.



Le processus de l'étude, tel qu'illustré dans la figure 5.8, consiste en cinq étapes principales décrites ci-dessous.

Étape 1. Nous utilisons aDoctor, Dynamics et Paprika sur le jeu de données de 538 applications pour détecter les sept défauts de code comportementaux, ce qui nous permet de récupérer les défauts de code détectés (c'est-à-dire les instances positives). Le tableau 5.3 indique le nombre de défauts de code détectés par Dynamics, Paprika et aDoctor.

Étape 2. En parallèle, nous identifions les défauts de code potentiels sur les 538 applications. Ces défauts de code potentiels sont toutes les classes pouvant être affectées par un défaut de code conforme à sa définition. Les classes de défauts de code potentiels sont identifiées lors de l'étape d'instrumentation du DYNAMICS (voir figure 5.1), au cours de laquelle les événements associés aux défauts de code potentiels sont localisés dans le code source.

Dans l'identification des défauts de code potentiels, nous faisons référence à des classes uniques. Les classes uniques signifient que l'on ne peut pas avoir deux défauts de code similaires dans une même classe. Par exemple, si on identifie deux défauts de code potentiels DW, un défaut dans la méthode *foo1* de la classe *Foo* et un défaut dans la méthode *foo2* de la classe *Foo*, ils comptabiliseront pour un seul défaut de code. Ce choix a été réalisé afin d'uniformiser le résultat des différents outils, certains retournant des méthodes et d'autres des classes. Les classes des défauts de code potentiels sont donc identifiées comme ci-dessous :

- **DW :** Les classes uniques contenant l'événement *Acquire*. Il y a 45 *Acquires* dans 40 classes uniques.
- **HMU :** Les classes uniques contenant l'événement HMU *Implementation*. Il y a 1559 HMU *Implementation* dans 836 classes uniques.
- **HBR :** Les classes uniques contenant la méthode *onReceive* (événements *Begin*/*End*). Il existe 719 classes uniques.
- **HSS :** Les classes uniques contenant la méthode *onStartCommand* (événements *Begin*/*End*). Il y a 134 classes uniques.
- **HAS :** Les classes uniques contenant le *onPreExecute* / *onPostExecute* / *onProgressUpdate* (événements *Begin*/*End*). Il existe 703 méthodes dans 509 classes uniques.
- **IOD :** Les classes uniques contenant la méthode *onDraw* (événements *Begin*/*End*). Il y a 130 classes uniques.

— **NLMR :** Les classes *Activity* uniques. Il existe 18 classes *Activity* contenant les méthodes *onLowMemory*/*onTrimMemory* (événements *Begin*/*End*) et 2002 classes *Activity* sans ces méthodes, soit un total de 2020 classes uniques.

Les défauts de code potentiels sont également indiqués dans les artefacts [13].

Étape 3. Pour chaque outil, nous filtrons les défauts de code potentiels pour ne récupérer que les défauts de code potentiels non détectés. Les défauts de code potentiels non détectés sont les défauts de code potentiels qui n'ont pas été détectés par les outils, et ne sont donc pas des défauts de code détectés. Ils sont indiqués dans les artefacts.

Étape 4 et 5. Afin de considérer des échantillons statistiquement significatifs pour les défauts de code détectés lors de l'étape 4 et les défauts de code potentiels non détectés lors de l'étape 5, nous nous appuyons sur un échantillonnage stratifié. Cet échantillonnage stratifié garantit que la proportion de chaque défaut de code est préservée dans l'échantillon. L'échantillon, pour chaque défaut de code, est constitué des défauts de code détectés par l'étape 1 et cette méthode représente un échantillon stratifié statistiquement significatif de 95% avec un intervalle de confiance de 10%. L'échantillonnage stratifié est en outre toujours effectué sur les défauts de code accessibles, c'est-à-dire ceux qui peuvent être trouvés dans le code source. Cette stratification est nécessaire, puisque pour valider les résultats des outils, aux étapes 6 et 7, le code source doit être inspecté manuellement. Le code source n'est cependant pas requis par notre outil, mais utilisé uniquement lors de ces deux étapes de validation manuelle. Certaines instances de défauts de code ne sont cependant pas accessibles en raison de l'obfuscation, puisque Paprika et Dynamics analysent directement l'APK. Ceci est aussi potentiellement une

13. https://doi.org/10.21227/49vr-wr08

conséquence des applications Kotlin, qui génère des fonctions au moment de la compilation. Par exemple, certaines des fonctions générées contiennent des *HashMap*, ce qui entraîne des défauts de code HMU non accessibles à PAPRIKA ni à DYNAMICS. Cela est également dû aux bibliothèques externes qui sont retournées dans les résultats, comme c'est le cas pour certains des défauts de code détectés par PAPRIKA. Le résultat de ces échantillonnages est présenté dans le tableau 5.4 dont le contenu va maintenant être détaillé.

À partir de là, les quatre types d'instances déterminés dans les étapes suivantes (vrais positifs, faux positifs, vrais négatifs, faux négatifs) sont détaillés et expliqués dans le tableau 4.4 de la section 4.4.

Étape 6. Nous analysons l'échantillon positif manuellement pour déterminer quels sont les vrais positifs et les faux positifs. Les vrais positifs sont les instances détectées par les outils et validées manuellement conformément avec leur définition des défauts de code : $VP = D \cap V$. Les faux positifs sont les instances détectées par les outils, mais validées manuellement comme étant non conformes à la définition des défauts du code : $FP = D \cap V^c$. L'échantillon est validé manuellement par quatre doctorants ayant une expérience des défauts de code et des applications mobiles. Elles sont présentées dans le tableau 5.4.

Étape 7. De même, nous analysons manuellement les défauts de code potentiels non détectés afin de déterminer quels sont les vrais négatifs et les faux négatifs. Les vrais négatifs sont les instances non détectées par les outils et validées manuellement comme étant non conformes à la définition : $VN = D^c \cap V^c$. Les faux négatifs sont les instances non détectées par les outils mais validées manuellement comme étant conformes à la définition des défauts de code : $FN = D^c \cap V$. L'échantillon est validé manuellement par quatre doctorants ayant une expérience des défauts de code et des applications mobiles. Elles sont présentées dans le tableau 5.4.

TABLEAU 5.2 Nombre de défauts de code détectés avec DYNAMICS en fonction du générateur d'entrée avec une exécution de cinq minutes et une exécution de dix minutes.

Défaut de code	DROIDBOTX		DROIDBOT		MONKEYRUNNER		DROIDBOTX DROIDBOT MONKEYRUNNER		
	5min	**10min**	**5min**	**10min**	**5min**	**10min**	**5min**	**10min**	**5min & 10min**
DW	3	3	3	1	1	3	3	3	3
HMU	221	266	235	268	185	227	294	319	324
HAS	6	2	5	2	9	2	14	5	15
HBR	2	3	4	2	1	2	5	5	8
HSS	2	1	0	2	2	1	3	2	4
IOD	10	11	5	8	9	7	14	13	16
NLMR	2004	2004	2004	2004	2004	2004	2004	2004	2004

Étape 8**.** Nous calculons la précision en utilisant l'échantillon positif. La précision est la proportion de vrais positifs parmi les positifs (ici, défauts de code détectés) $\frac{|VP|}{|D|}$. Ensuite, nous calculons le rappel avec l'échantillon positif et l'échantillon négatif. Le rappel est la proportion de vrais positifs parmi les vrais positifs et les faux négatifs $\frac{|VP|}{|VP \cup FN|}$. Ces valeurs sont présentées dans le tableau 5.4.

5.3.5 Résultats

Dans cette section, nous allons étudier et répondre aux hypothèses au cas par cas.

H_1 DYNAMICS permet la détection des défauts de code comportementaux.

Nous examinons cette hypothèse par le biais d'une étude quantitative. Nous présentons d'abord le nombre de défauts de code détectés par l'outil DYNAMICS, et les comparons à ceux détectés par PAPRIKA et ADOCTOR.

Tout d'abord, nous examinons le nombre de défauts de code détectés par l'outil DYNAMICS décrit dans le tableau 5.2. Le nombre de défauts de code détectés varie considérablement en fonction du type de défaut de code. Nous avons effectué la détection pour chaque générateur d'entrée, ainsi que pour la combinaison des trois générateurs d'entrée. Souvent, on constate que l'usage de DROIDBOTX pour générer les traces permet de détecter un peu plus de défauts de code que l'utilisation de DROIDBOT, et que l'utilisation de DROIDBOT permet de détecter plus de défauts de code que l'utilisation de MONKEYRUNNER, mais c'est loin d'être toujours vrai à cause du non-déterminisme des exécutions. La combinaison des trois outils permet d'obtenir plus de défauts de code, ce qui correspond à l'union des défauts de code détectés par chaque générateur d'entrée. Le nombre de défauts de code détectés dépend du nombre d'événements rencontrés, et donc de la qualité de l'exécution. La limite supérieure des défauts de code détectables est définie par les défauts de code potentiels. Ainsi, on ne peut pas détecter plus de défauts de code qu'il n'y a de classes contenant des événements rencontrés, par exemple, on ne peut pas détecter plus de défauts de code DW qu'il n'y a de classes contenant des appels *acquire* rencontrés. Des facteurs tels que le générateur d'entrée choisi et le nombre de traces augmentent le nombre de défauts de code détectés. Le temps d'exécution a également un impact sur le nombre de défauts de code détectés, un temps d'exécution plus long peut permettre d'atteindre plus d'événements, et donc potentiellement plus de défauts de code à détecter.

Ce n'est cependant pas toujours le cas, comme le montre le tableau 5.2 qui présente le nombre de défauts de code détectés par DYNAMICS avec des traces d'exécution de 5 minutes et de 10 minutes. Certains défauts de code ont effectivement plus d'occurrences lorsque nous doublons le temps d'exécution. Mais dans d'autres cas, la nature aléatoire du générateur d'entrée produit un nombre réduit d'instances. Dans tous les cas, la combinaison des traces d'une exécution de 5 minutes avec

Tableau 5.3 Nombre de défauts de code détectés par les outils Dynamics, Paprika et aDoctor.

Défaut de code	**#Défauts de code Dynamics**	**#Défauts de code Paprika**	**#Défauts de code aDoctor**
DW	3	N/A	95
HMU	324	573	N/A
HAS	15	109	N/A
HBR	8	305	N/A
HSS	4	63	N/A
IOD	16	18	N/A
NLMR	2004	1017	2999

celles d'une exécution de 10 minutes augmente nécessairement le nombre total de défauts de code détectés, ou au moins ne diminue pas le nombre total, comme présenté dans la dernière colonne "5min & 10min" du tableau 5.2.

À titre de comparaison, nous examinons également les défauts de code détectés par Paprika et aDoctor présentés dans le tableau 5.3. Dynamics renvoie moins de cas de défauts de code qu'aDoctor ou Paprika. Par exemple, la différence est significative pour DW, où Dynamics détecte 3 défauts de code par rapport à aDoctor qui détecte 95 défauts de code, ou pour HAS, où Dynamics détecte 15 défauts de code par rapport à Paprika qui détecte 109 défauts de code. La différence est parfois plus faible, comme pour IOD, où Dynamics détecte 16 défauts de code par rapport à Paprika qui détecte 18 défauts de code. Comme discuté dans notre étude empirique précédente, cela peut être dû à l'analyse dynamique qui permet une meilleure précision et donc moins de faux positifs. Mais cela peut aussi être dû au fait que la couverture des événements pendant les exécutions n'est pas parfaite et que certains faux négatifs sont présents dans le résultat de Dynamics.

> ***H_1* : DYNAMICS permet la détection des défauts de code comportementaux**. Cependant, en ce qui concerne le nombre d'instances, DYNAMICS détecte moins d'instances (comme présenté dans le tableau 5.3) qu'aDOCTOR et PAPRIKA. Comme nous l'avons vu dans l'étude empirique du chapitre 4, PAPRIKA et aDOCTOR ont signalé de nombreux faux positifs (environ 30%). Nous étudions maintenant la précision et le rappel de DYNAMICS pour mieux comprendre ces résultats.

H_2 DYNAMICS permet la détection **précise et exacte** des défauts de code comportementaux

Nous examinons cette hypothèse par le biais d'une étude quantitative. Nous présentons d'abord la précision et le rappel de DYNAMICS, et nous les comparons à ceux d'aDOCTOR et PAPRIKA. Ensuite, nous discutons du rappel de DYNAMICS en présentant la couverture des événements pendant les exécutions.

Selon l'analyse manuelle présentée dans le processus de la validation de la figure 5.8, nous calculons la précision et le rappel de DYNAMICS. Le tableau 5.4 donne la précision et le rappel de DYNAMICS en comparaison avec aDOCTOR et PAPRIKA. Notez que la précision et le rappel de DYNAMICS sont traités sur la base des défauts de code détectés en utilisant les traces obtenues à partir de la combinaison des exécutions de 5 et 10 minutes (dernière colonne du tableau 5.2). Le tableau 5.2 présente les résultats pour tous les défauts de code, sauf HBR et HSS. Nous avons choisi de présenter uniquement les résultats du défaut de code HAS, puisque les règles de détection de HAS, HBR et HSS sont très similaires et ne diffèrent que par le nom de la méthode. De plus, le nombre de défauts de code détectés pour HBR et HSS, en particulier dans PAPRIKA, est très élevé (voir le tableau 5.3), ce qui rend l'analyse manuelle fastidieuse. Nous nous concentrons

TABLEAU 5.4 Précision et rappel de DYNAMICS en comparaison avec ADOCTOR et à PAPRIKA.

Défaut de code	#Défauts détectés #Défauts échantillonnés détectés #Vrai Positifs Précision : $\frac{\|VP\|}{\|D\|}$			#Défauts potentiels non-detectés #Défauts potentiels échantillonnés non-detectés #Faux Négatifs Rappel : $\frac{\|VP\|}{\|VP \cup FN\|}$		
	DYNAMICS	PAPRIKA	ADOCTOR	DYNAMICS	PAPRIKA	ADOCTOR
DW	3 3 3 **100%**	N/A	95 46 9 19.6%	37 27 7 30%	N/A	8 8 5 **64.3%**
HMU	324 69 44 **63.8%**	573 30 10 33.3%	N/A	512 76 43 **50.6%**	800 82 45 18.2%	N/A
HAS	15 11 11 **100%**	109 46 16 34.8%	N/A	494 79 17 39.3%	424 77 10 **61.5%**	N/A
IOD	16 16 16 **100%**	18 17 17 **100%**	N/A	114 41 12 57.1%	121 53 11 **60.7%**	N/A
NLMR	2004 92 92 **100%**	1017 87 87 **100%**	2999 93 93 **100%**	16 16 10 **90.2%**	1370 90 89 49.4%	1074 88 85 52.2%
	Précision moyenne de DYNAMICS : **92.8%**			Rappel moyen de DYNAMICS : **53.4%**		

donc spécifiquement sur HAS.

Il est important de noter que la précision et le rappel d'aDoctor sont moins bons pour le défaut de code DW par rapport à l'étude empirique du chapitre 4. En effet, dans l'étude empirique, nous avons pris en compte la règle définie dans l'article d'aDoctor (Palomba *et al.*, 2017) afin d'avoir des échantillons à étudier, la règle implémentée dans l'outil ne correspond pas à la règle indiquée dans le papier et est suffisamment erronée pour ne pas avoir d'échantillons pertinents. En effet, dans l'implémentation les règles de détection détectent le défaut de code DW en recherchant le terme "*acquire*" peu importe que le terme soit associé à un *WakeLock* ou non. Cependant, dans cette validation nous comparons directement les outils entre eux, nous avons donc gardé les résultats bruts de l'outil.

Comme le montre le tableau 5.4, la précision globale de Dynamics est élevée avec une moyenne de 88.1%. En outre, Dynamics fournit une meilleure précision que les deux autres outils pour chaque défaut de code comportemental détecté. En ce qui concerne le rappel, Dynamics fournit une moyenne de 55.4%, mais le rappel varie fortement en fonction des défauts de code. Contrairement à la précision, le rappel du Dynamics n'est pas toujours meilleur que celui des deux autres outils. Cela peut être dû au fait que la couverture des événements pendant les exécutions n'est pas bonne. En effet, certains événements associés à des défauts de code peuvent ne pas avoir été rencontrés au cours des exécutions et peuvent donc entraîner l'apparition de faux négatifs, c'est-à-dire de défauts de code non détectés qui auraient dû l'être. Ainsi, pour vérifier cette supposition, nous étudions la couverture des événements lors des exécutions en fonction des générateurs d'entrée, du nombre d'exécutions et du temps d'exécution.

Le tableau 5.5 indique le nombre d'événements (tels que spécifiés dans le tableau 5.1) rencontrés dans le code (jeu de données constitué de 538 APK) et

TABLEAU 5.5 Nombre d'événements rencontrés dans le code et dans les traces d'exécution en utilisant les outils de génération d'entrées sur DYNAMICS pendant 5 minutes et 10 minutes

Défaut de code	Événement	Dans le code	DroidBotX		DroidBot		MonkeyRunner		DroidBotX DroidBot MonkeyRunner		
			5min	10min	5min	10min	5min	10min	5min	10min	5min & 10min
DW	**Acquire**	45	4	4	3	2	4	4	5	4	5/45 (11%)
	Release	50	2	2	1	1	3	2	3	2	3/50 (6%)
HMU	**Implementation**	1559	308	371	324	367	272	323	406	439	449/1559 (29%)
	Addition	2654	158	187	141	211	134	164	205	254	265/2654 (10%)
	Deletion	47	3	4	4	4	1	3	4	5	5/47 (11%)
	Clean	438	18	33	15	28	19	18	29	42	43/438 (10%)
HAS	**Begin/End**	703	90	113	89	103	68	91	128	132	143/703 (20%)
HBR	**Begin/End**	719	43	58	42	58	44	40	66	84	94/719 (13%)
HSS	**Begin/End**	134	25	31	31	37	27	28	40	43	48/134 (36%)
IOD	**Begin/End**	130	41	46	37	50	31	41	49	54	56/130 (43%)
	New	487	9	9	4	10	5	6	9	11	11/487 (2%)
NLMR	**Begin/End**	18	1	1	1	2	0	2	1	2	2/18 (11%)

dans les traces d'exécution générées par DROIDBOTX, DROIDBOT et MONKEYRUNNER. Les événements dans le code sont des caractéristiques uniques du code source situées statiquement dans l'application, tandis que les événements rencontrés dans les traces d'exécution sont des événements uniques rencontrés au cours des exécutions à la suite d'une analyse dynamique. Le nombre d'événements varie légèrement en fonction du générateur d'entrée utilisé et fortement en fonction du défaut de code. Par exemple, pour une exécution de 5 minutes, nous rencontrons 272 événements *Implementation* HMU avec MONKEYRUNNER alors que nous rencontrons 308 événements *Implementation* pour le défaut de code HMU avec DROIDBOTX. De même, avec DROIDBOTX, nous rencontrons 4 événements *Acquire* pour le défaut de code DW alors que nous rencontrons 308 événements *Implementation* pour le défaut de code HMU.

La couverture des événements est plus élevée en utilisant DROIDBOT ou DROID-

BotX que MonkeyRunner. Nous définissons la couverture des événements par le nombre d'événements rencontrés pendant l'exécution par rapport au nombre d'événements présents dans le code. Toutefois, il est possible d'obtenir une meilleure couverture des événements avec MonkeyRunner en augmentant le nombre d'actions par seconde comme paramètre de l'outil. En revanche, DroidBot et DroidBotX ne permettent qu'une action par seconde. De même, la combinaison des générateurs d'entrée, l'augmentation du nombre d'exécutions et/ou l'augmentation du temps d'exécution entraînent une légère amélioration de la couverture des événements, tout comme elle augmente le nombre de défauts de code détectés (voir le tableau 5.2). Cependant, il peut arriver que la couverture ne s'améliore pas. Par exemple, pour le code HBR, avec MonkeyRunner, nous avons obtenu 44 événements *Begin*/*End* pour l'exécution de 5 minutes et 40 événements *Begin*/*End* pour l'exécution de 10 minutes. Ceci est dû au fait que les actions générées par un générateur d'entrée donné sont faites de manière aléatoire. Comme exemple de l'augmentation de la couverture des événements, avec la combinaison des trois générateurs d'entrée et des différents nombres et temps d'exécution, nous atteignons 94 événements rencontrés pour l'événement *Begin*/*End* HBR, soit une augmentation significative de 62% (36 événements) par rapport aux 58 événements de DroidBotX, qui était le plus élevé. En revanche, pour IOD, nous atteignons 56 événements rencontrés pour l'événement Début/Fin, soit une légère augmentation de 12% (6 événements) par rapport aux 50 événements de DroidBot, qui était le plus élevé. Malgré le fait que nous ayons augmenté le temps d'exécution, le nombre d'exécutions et le nombre de générateurs d'entrée, la couverture reste faible. Elle ne varie que de 2% à 43% au maximum selon l'événement. Il pourrait être intéressant d'explorer d'autres générateurs d'entrée plus adaptés afin de permettre une meilleure couverture.

L'étude de la couverture des événements confirme l'hypothèse énoncée précédem-

ment, à savoir que le rappel n'est pas très bon en raison de la faible couverture des événements. En effet, les défauts de code existants sont limités par les événements dans le code, par exemple 45 événements *Acquire* pour le défaut de code DW. D'autre part, les défauts de code détectables sont limités par les événements rencontrés, par exemple 5 événements *Acquire* pour le défaut de code DW. Il y aura donc des défauts de code potentiels non détectés qui auraient dû être détectés, c'est-à-dire des faux négatifs, tant que la couverture est inférieure à 100%. Par exemple, il y a 40 événements *Acquire* pour le défaut de code DW qui ne sont pas rencontrés par notre outil et qui sont potentiellement liés à un défaut de code. Comme ces faux négatifs sont limités par les événements non rencontrés dans le code, plus la couverture est faible, plus les faux négatifs sont fréquents et plus le rappel est faible. Cependant, on pourrait s'attendre à ce que la couverture des événements soit directement liée au rappel, mais ce n'est pas aussi simple que cela. Alors qu'une faible couverture entraîne généralement un faible rappel, une plus faible couverture n'implique pas nécessairement un plus faible rappel. Par exemple, le défaut de code HMU, qui a une couverture de 29% de l'événement *Implementation*, a un rappel de 68,4%, alors que le défaut de code IOD, qui a une couverture de 43% des méthodes *onDraw*, a un rappel de 57,1%. Une étude détaillée des approches du générateur d'entrée et de leur impact sur le rappel sort du cadre de cette étude, mais mérite certainement d'être approfondie.

***H*$_2$: DYNAMICS permet la détection précise et exacte des défauts de code comportementaux**. Le rappel dépend fortement de la couverture des événements pendant les exécutions. Bien que la précision soit élevée, le rappel est raisonnable par rapport à la couverture des événements. L'augmentation du nombre d'exécutions et de la durée des exécutions ainsi que la combinaison des générateurs d'entrée ne permettront qu'une amélioration négligeable à légère de la couverture. La couverture pourrait être améliorée en utilisant un générateur d'entrée dédié.

***H*$_3$** DYNAMICS permet la détection **précise et exacte** des défauts de code comportementaux **grâce à sa nature dynamique**

À travers une étude qualitative, nous examinons maintenant chaque défaut de code comportemental détecté par DYNAMICS et discutons de leur nature dynamique pour expliquer pourquoi ils ont été détectés par DYNAMICS. Nous comparons en outre la manière dont les défauts de code sont détectés à l'aide d'outils de détection statiques tels que PAPRIKA et ADOCTOR comme présenté dans le tableau 5.3 et le tableau 5.4, et soulignons l'importance de l'analyse dynamique. Les résultats discutés sont ceux obtenus avec les traces DROIDBOT, DROIDBOTX et MONKEYRUNNER simultanément avec une exécution de 5 minutes et une exécution de 10 minutes, comme présenté dans le tableau 5.2, le tableau 5.4 et le tableau 5.5.

DW : La détection du défaut de code DW par DYNAMICS offre une bien meilleure précision (100%) qu'ADOCTOR (19,6%) alors que le rappel est moins bon (30% contre 64,3%). La grande différence de précision s'explique par le fait que la règle de détection de DYNAMICS pour DW vérifie explicitement qu'un appel à la méthode *acquire* n'est pas suivi d'un appel à la méthode *release* (que ce soit dans la

même méthode ou dans une méthode différente appartenant à la même classe ou à une classe différente) alors qu'aDoctor vérifie uniquement la présence de la chaîne *acquire*. Le faible rappel s'explique par la faible couverture des événements *Acquire* pendant l'exécution ($5/45 = 11\%$ voir tableau 5.5).

Plus précisément, pour aDoctor, la règle de détection statique mentionnée dans leur document de référence (Palomba *et al.*, 2017) spécifie que, si une méthode utilisant une instance de la classe *WakeLock* acquiert le verrou sans appeler la méthode *release*, un défaut est identifié. Cependant, après avoir inspecté l'implémentation concrète de la règle dans l'outil aDoctor, nous avons constaté que la règle d'implémentation utilise une expression régulière "$(.*)acquire(\\s*)()$" en recherchant la chaîne "acquire" dans le code, et non la méthode spécifique *acquire* de la classe *WakeLock*. Un grand nombre de faux positifs avec aDoctor font référence à des méthodes *acquire* qui n'appartiennent pas à *WakeLock*.

Dans les traces d'exécution, sur les 5 événements DW *Acquire* rencontrés (voir le tableau 5.1), trois étaient liés à un défaut de code et deux ne l'étaient pas. En ce qui concerne les deux événements DW *Acquire* non liés à un défaut de code, nous avons constaté que chaque méthode *acquire* est appelée et effectivement suivie d'une méthode *release*. Chacun des 3 événements *Acquire* correctement détectés comme des défauts de code par Dynamics n'est pas suivi d'un appel à une méthode *release* bien que la méthode soit présente dans une autre méthode de la même classe. Cela montre que la présence des méthodes *acquire* et *release* n'est pas suffisante pour les exclure en tant que défauts de code. Un appel concret à ces méthodes au sein du même *WakeLock* est nécessaire. Les outils statiques actuels, tels qu'aDoctor, sont donc insuffisants pour la détection précise de ce défaut de code.

HMU : La détection du défaut de code HMU par Dynamics fournit une pré-

cision (74,7%) et un rappel (68,4%) légèrement meilleurs que ceux du PAPRIKA (respectivement 60% et 40%). Ces résultats s'expliquent par le fait que la règle de détection de DYNAMICS pour HMU identifie explicitement les petites *HashMaps* (moins de 500 éléments) et les grandes *ArrayMap*/*SimpleArrayMap* (plus de 500 éléments) alors que PAPRIKA identifie uniquement la présence de *HashMaps*. De plus, que ce soit pour DYNAMICS ou PAPRIKA, la plupart des *HashMaps* identifiées sont de petite taille. Cette prépondérance de petites *HashMaps* explique la légère différence entre la précision et le rappel obtenus par PAPRIKA, qui détecte toute utilisation de *HashMap* comme un défaut de code.

Plus précisément, dans les traces d'exécution, sur les 449 événements *Implementation* HMU rencontrés (voir tableau 5.5), seuls trois sont des *ArrayMap* et un est une *SimpleArrayMap*, tous les autres événements sont des *HashMaps*. Parmi ces *HashMaps*, la plupart sont de petites *HashMaps*. Seules deux sont des *HashMaps* qui dépassent 500 éléments et cinq vont au-delà de 100 éléments. Les trois *ArrayMaps* et la *SimpleArrayMap* rencontrées sont petites et ne concernent donc pas les instances HMU.

D'un point de vue plus technique, pour la détection de HMU, nous avons identifié deux aspects techniques intéressants. Le premier est que les exécutions générées par les générateurs d'entrée ne permettent pas nécessairement aux structures d'atteindre une taille suffisamment grande pour identifier la présence d'une HMU. En effet, une exécution qui conduit à une taille suffisamment grande dans une structure spécifique peut souvent nécessiter de faire des actions spécifiques dans une boucle, et les générateurs d'entrées effectuent plutôt de multiples explorations aléatoires. Le deuxième aspect technique intéressant concerne la détection au sein des applications KOTLIN. En effet, la détection du défaut de code HMU donne au moins un tiers de plus d'instances HMU lors de l'exécution sur les applications KOTLIN. En effet, la compilation des applications KOTLIN génère automatique-

ment pour les classes *Activity* et *Fragment* un grand nombre de méthodes contenant des *HashMaps*, non présentes dans le code source. Nous les avons exclues de l'échantillonnage et considérons seulement les autres instances de HMU. Nous prévoyons d'affiner la détection de DYNAMICS dans la prochaine version pour exclure les méthodes générées.

HAS : La détection du défaut de code HAS par DYNAMICS fournit une bien meilleure précision (100%) que PAPRIKA (34,8%) alors que le rappel est moins bon (39,5% contre 61,5%). La grande différence de précision s'explique par le fait que la règle de détection de DYNAMICS pour HAS considère le temps d'exécution de la méthode, tandis que PAPRIKA utilise les métriques de complexité cyclomatique et de nombre d'instructions. Cependant, ces deux dernières métriques donnent une estimation très grossière du temps d'exécution, tandis que DYNAMICS considère spécifiquement le temps d'exécution de la méthode. Le faible rappel s'explique par la faible couverture des événements *Begin/End* pendant l'exécution ($143/703 \approx 20\%$ voir tableau 5.5).

En effet, sur les 15 instances détectées avec DYNAMICS, 14 ne sont pas détectées par PAPRIKA (tableau 5.4). En revanche, sur les 143 événements HAS *Begin/End* rencontrés pendant l'exécution (tableau 5.5), 126 n'étaient pas liés à un défaut de code selon DYNAMICS. De même, parmi les 126 méthodes non liées à un défaut de code selon DYNAMICS, 25 ont été détectées comme des défauts de code par le PAPRIKA. Cela montre qu'une méthode courte avec une faible complexité cyclomatique et un faible nombre d'instructions peut prendre beaucoup de temps à s'exécuter, et qu'une méthode longue avec une complexité cyclomatique élevée et un grand nombre d'instructions peut s'exécuter rapidement.

IOD : La détection du défaut de code IOD par DYNAMICS fournit la même précision que PAPRIKA (100%) et le rappel est presque égal (57,1% pour DYNAMICS contre 60,7% pour PAPRIKA). Bien que ces résultats en ce qui concerne la précision et le rappel soient assez similaires, les instances détectées ne le sont pas. DYNAMICS détecte ce défaut de code à l'aide d'une propriété divisée en deux parties. La première partie de la propriété vérifie si le temps d'exécution de la méthode est inférieur à 1/60s. La deuxième partie de la propriété vérifie s'il y a une initialisation d'un objet dans la méthode pendant l'exécution. PAPRIKA identifie ce défaut de code en analysant statiquement s'il y a une initialisation d'un objet dans la méthode *onDraw*. Comme la règle de détection du DYNAMICS utilise deux parties, on pourrait s'attendre à détecter plus de défauts de code avec DYNAMICS. Cependant, les deux outils détectent presque le même nombre d'instances, puisque DYNAMICS rencontre moins de méthodes pendant l'exécution. En effet, le rappel de DYNAMICS s'explique par la couverture de seulement 43% des méthodes pendant l'exécution (voir tableau 5.5).

Plus précisément, parmi les 16 défauts de code IOD du tableau 5.4, 6 sont détectés en raison de la présence d'un événement *New*, 8 sont détectés parce que le temps d'exécution dépasse 1/60ème de seconde, et 2 ont été détectés parce qu'il s'agit à la fois de la présence de l'événement *New* et du dépassement du temps d'exécution. Les 8 (6 + 2) instances détectées par la présence de l'événement *New* auraient dû être également détectées par la règle de détection PAPRIKA, mais elle n'en détecte que 2. Cela peut s'expliquer par la différence entre les implémentations concrètes de la règle de détection dans PAPRIKA et DYNAMICS. Les 8 instances détectées par le dépassement du temps d'exécution ne peuvent pas être détectées par des outils statiques, en raison de la nature dynamique de la règle correspondante. Cela montre l'avantage de la nature dynamique de notre outil.

NLMR : La détection du défaut de code NLMR par DYNAMICS fournit la même précision que PAPRIKA et ADOCTOR (100%) et le rappel est meilleur (90,2% pour DYNAMICS contre 49,4% pour PAPRIKA et 52,2% pour ADOCTOR). Cette différence dans le rappel s'explique par la légère différence dans l'interprétation des méthodes à considérer. Tout d'abord, nous prenons également en compte la méthode *onTrimMemory* contrairement à ADOCTOR et PAPRIKA, puisque la méthode *onLowMemory* a été dépréciée après la publication de ces deux outils. Enfin, ADOCTOR prend également en compte la classe FRAGMENT, qui est une autre classe Android implémentant l'interface *ComponentCallbacks* qui définit les méthodes *onLowMemory* et *onTrimMemory*. Par conséquent, ADOCTOR détecte plus de défauts de code que DYNAMICS. De son côté, PAPRIKA exclut les classes *Activity* qui héritent d'une autre *Activity* interne de l'application, ce qui explique qu'il détecte moins de défauts de code que DYNAMICS. Les trois approches semblent valables selon l'interprétation du défaut de code NLMR, mais ont un impact sur le rappel, qui est donc moins important que les autres défauts de code. DYNAMICS détecte ce défaut de code à l'aide de deux règles, l'une statique et l'autre dynamique. La règle statique vérifie si une méthode *onLowMemory* ou *onTrimMemory* est implémentée dans une classe *Activity*. La propriété dynamique vérifie si l'une de ces méthodes libère moins de 1024 Ko de mémoire.

Plus précisément, sur les 2004 instances NLMR détectées, 2002 défauts de code NLMR (vrais positifs) sont détectés par la règle statique et 2 (vrais positifs) sont détectés par la propriété dynamique, ce qui signifie que les méthodes rencontrées ne vidaient pas correctement les caches. En effet, seules deux méthodes sont rencontrées au cours de l'exécution, comme le montre le tableau 5.5. Nous remarquons également qu'il n'y a que 18 méthodes identifiées dans le code source, donc ce défaut de code est susceptible d'être assez présent, puisque chaque application a au moins une *Activité*.

Par conséquent, la détection de ce défaut de code ne nous permet pas d'évaluer le bénéfice de notre approche dynamique, puisque la grande majorité des défauts de code sont détectés de manière statique. Seuls deux cas de défauts de code sont détectés en raison de la nature dynamique du DYNAMICS. Les règles de détection sont donc quasiment identiques pour les trois outils, en fonction de l'interprétation du défaut de code.

L'étude qualitative montre que DYNAMICS peut effectivement être appliqué aux trois catégories de défauts de code comportementaux identifiées dans le chapitre 3.

> ***H_3*** **:** DYNAMICS permet la détection **précise et exacte** des défauts de code comportementaux **grâce à sa nature dynamique**. Pour chaque défaut de code comportemental, DYNAMICS permet de détecter des instances spécifiques de défauts de code qui n'ont pas pu être détectées par l'analyse statique. Cette détection est possible pour les défauts de code de chaque catégorie de défauts de code identifiée. DYNAMICS permet également de détecter les instances de défauts de code qu'il était possible de détecter par analyse statique.

5.3.6 Menaces à la validité

Validité interne. La principale menace à la validité interne est les outils de génération d'entrées utilisés. En effet, les résultats dépendent beaucoup des traces d'exécution obtenues. Nous aurions pu obtenir des résultats plus significatifs, notamment pour HMU, si nous avions eu une bonne couverture d'événements avec les générateurs d'entrée. De plus, chaque générateur d'entrée utilisé est basé sur un certain degré de hasard, ce qui signifie qu'il peut générer des traces différentes si on les utilise plusieurs fois. Des traces différentes peuvent potentiellement conduire

à une détection différente des défauts de code. Cependant, nous avons présélectionné les outils générateurs d'entrées à partir d'un ensemble extrait de l'état de l'art afin de sélectionner ceux qui semblaient être les plus efficaces à traiter. Une autre menace pour la validité interne est que l'ajout d'instructions liées à l'instrumentation pourrait avoir un léger impact négatif sur les performances et corrompre les défauts de code détectés. Ces instructions ajoutées sont minimes et une comparaison du temps d'exécution avant et après l'instrumentation n'a rien montré d'inhabituel.

Validité externe. La principale menace à la validité externe pourrait être le jeu de données utilisé pour la validation. L'utilisation de projets libres d'accès de bonne qualité et matures peut entraîner la quasi-absence de certains défauts de code. La plupart de ces défauts de code peuvent être trouvés dans des applications en cours de développement. F-Droid nous fournit une variété d'applications, plus ou moins matures et de qualité diverse, ce qui nous permet d'avoir un jeu de données représentatif.

Validité de répétitivité/fiabilité. Pour certains défauts du code, les seuils appliqués pour les identifier peuvent être de nature subjective, bien que les propriétés associées à la détection des défauts de code soient basées sur la définition et les références dont les définitions sont tirées. En outre, l'interprétation de certains défauts de code peut également être subjective, comme le défaut de code NLMR où la propriété garantit que la méthode *onLowMemory* libère une certaine quantité de mémoire. De même, pour le défaut de code IOD, la propriété garantit que le temps d'exécution de la méthode *onDraw* est inférieur à 1/60ème de seconde.

Généralisabilité. Les résultats de la validation sont reproductibles et fiables, puisque DYNAMICS et tous les outils connexes nécessaires sont libres d'accès et disponibles dans les artefacts [14]. Les résultats sont également disponibles dans nos artefacts.

14. https://doi.org/10.21227/49vr-wr08

CHAPITRE VI

CONCLUSION ET PERSPECTIVES

6.1 Conclusion

La présence des défauts de code est un problème bien connu dans le secteur du génie logiciel pour lequel de nombreuses recherches et outils ont été apportés pour les identifier, les détecter et les corriger. Le contexte des applications mobiles a apporté de nouveaux sujets de préoccupations, comme la consommation énergétique, la mémoire et le processus limité des appareils mobiles. Ces dernières années, des études et des outils sont apparus pour répondre à cette nouvelle problématique en introduisant de nouveaux défauts de code spécifiques aux applications mobiles, entre autres des défauts de code spécifiques à Android. Cependant, l'entièreté de ces outils sont à base d'analyse statique et ne prennent pas en compte le comportement de l'application mobile pendant son exécution. Hors, certains de ces défauts de code spécifiques à Android sont dit comportementaux, et ne se manifestent que pendant l'exécution de l'application.

Cette thèse traite des défis à relever pour la détection dynamique des défauts de code comportementaux dans Android. Pour cela, nous proposons : (1) une étude empirique afin d'introduire la notion de défaut de code comportemental, d'étudier la détection de ces défauts de code comportementaux dans Android par les outils de la littérature et montrer l'importance d'utiliser de nouvelles techniques

de détection ; (2) une approche outillée DYNAMICS, permettant la spécification et la détection dynamique et formelle des défauts de code comportementaux dans Android.

(1) Étude empirique : Dans cette thèse, nous avons tout d'abord présenté une étude empirique sur la détection des défauts de code comportementaux dans Android. Comme cette étude est la première de son genre sur la détection des défauts de code comportementaux, elle est encore menée à petite échelle car elle nécessite un travail manuel important. Nous avons analysé et comparé l'approche de PAPRIKA et de ADOCTOR, tous deux basés sur l'analyse statique. Nous nous sommes concentrés uniquement sur les défauts de code spécifiques concernant le comportement de l'application. Nous avons montré que les outils retournent des faux négatifs et des faux positifs pour ces défauts de code comportementaux, ce qui affecte l'efficacité des outils. Nous avons montré par une étude qualitative sur les défauts de code spécifiques que certaines apps qui n'étaient pas signalées comme ayant des défauts, ne répondaient pourtant pas à la définition littérale des défauts. Une partie non négligeable des défauts de code détectés sont des faux positifs, c'est-à-dire des défauts de code qui n'auraient pas dû être détectés. Nous avons étudié les règles de détection afin de découvrir les problèmes à l'origine de ces erreurs de détection. De plus, nous avons discuté des impacts de ces règles de détection sur les résultats. Il s'agit de la première étude préliminaire de ce type sur ce sujet, afin de sensibiliser la communauté à fournir des outils de détection plus adaptés pour traiter les défauts de code comportementaux. Il est essentiel d'éviter les faux positifs et les faux négatifs pour aider au maximum le développeur dans le développement de l'application. En outre, nous mettons en avant la nécessité de prendre en compte le comportement dans le domaine de la détection des défauts de code d'Android, où de nombreux problèmes sont essentiellement comportementaux.

(2) DYNAMICS, une approche outillée pour la détection de défauts de code comportementaux : Dans cette thèse, nous proposons également DYNAMICS, une méthode et l'outil associé pour détecter les défauts de code comportementaux dans Android à l'aide d'une analyse dynamique. DYNAMICS se compose de quatre étapes. Premièrement, nous spécifions les défauts de code comportementaux par le biais d'événements et de propriétés LTL. Ensuite, DYNAMICS prend en entrée une application mobile sous la forme d'un APK pour produire une application instrumentée. Ensuite, il exécute l'application instrumentée pour générer des traces d'exécution à l'aide d'outils de génération d'entrée. Enfin, il analyse les traces d'exécution pour détecter les défauts de code comportementaux qui se manifestent pendant l'exécution de l'application mobile (en s'assurant que les propriétés LTL, liées aux défauts de code comportementaux, sont préservées). Nous avons validé la détection de sept défauts de code spécifiques à Android sur 538 applications mobiles réelles extraites de F-DROID. Nous avons appliqué une analyse à méthode mixte pour comprendre les résultats de notre validation. *Quantitativement*, nous avons analysé les résultats de détection de DYNAMICS sur les 538 applications et les avons comparés à ADOCTOR et PAPRIKA en termes de nombre d'instances, de précision et de rappel. *Qualitativement*, nous avons analysé en détail les défauts de code signalés et les avons comparés aux outils de détection de l'analyse statique afin de souligner l'importance de l'analyse dynamique.

La validation indique que DYNAMICS permet la détection précise et exacte des défauts de code comportementaux en raison de sa nature dynamique. DYNAMICS permet la détection de trois catégories de défauts de code comportementaux. (1) Les défauts de code comportementaux caractérisés par une mauvaise utilisation d'un appel de méthode ou d'une séquence d'appels de méthode pendant l'exécution ; (2) Les défauts de code comportementaux caractérisés par des problèmes d'exécution, tels qu'un long temps d'exécution d'une méthode ou une utilisation

excessive de la mémoire ; (3) Les défauts de code comportementaux caractérisés par des variations de données indésirables pendant l'exécution, telles que la taille d'une structure devenant excessivement grande. Tout type de défaut de code appartenant à ces catégories peut être détecté par l'outil DYNAMICS, avec un effort de développement raisonnable comme nous l'expliquerons dans les perspectives. Nous pensons que DYNAMICS est un outil qui peut être utile en complément des outils statiques traditionnels pour fournir une détection plus fine des défauts comportementaux du code.

Le travail présenté dans cette thèse est donc une avancée dans la détection des défauts de code spécifiques à Android, particulièrement les défauts de code comportementaux. Nous avons introduit la notion de défaut de code comportemental et apporté une base de recherche sur la détection dynamique des défauts de code dans les applications mobiles en apportant un outil.

6.2 Perspectives

Le travail de cette thèse aboutit sur de nombreuses perspectives de recherche. Les perspectives passent par les défauts de code comportementaux, l'amélioration de l'outil actuel ainsi que l'utilisation de nouvelles techniques. Nous organisons cette section en présentant les différentes perspectives.

6.2.1 Identification de nouveaux défauts de code comportementaux

La liste des défauts de code détectés par notre outil pourrait être étoffée par d'autres défauts de code comportementaux présents dans l'état de l'art et identifiés dans l'une des trois catégories de défaut de code comportementaux. L'architecture de l'implémentation de DYNAMICS est pensée pour l'ajout de nouveaux défauts de code comportementaux. Pour prendre en compte un nouveau défaut de code

comportemental, il faut le spécifier. Précisément, il faut spécifier une propriété et ses évènements associés. Il faut ensuite ajouter ces évènements à l'architecture du code, donner les instructions d'instrumentation selon les informations souhaitées et implémenter la vérification de la propriété en utilisant l'interface de programmation BeepBeep 3.

De plus, la plupart des défauts de code spécifiques à Android de l'état de l'art sont introduits en vue d'être détectés par des outils de détection statique. L'introduction d'un outil de détection dynamique, comme DYNAMICS, pourrait permettre de détecter des défauts de code dont la définition est très orientée dynamique. De nouveaux défauts de code comportementaux pourraient donc être identifiés en vue d'être détectés par DYNAMICS. Ces nouveaux défauts de code comportementaux, comme les autres défauts de code spécifiques à Android, pourraient être identifiés à partir de livres spécialisés, sur des sites, des blogues, des conférences ou bien des vidéos spécialisés.

6.2.2 Amélioration de la simulation des exécutions des applications

Lors de la présentation de DYNAMICS, nous avons vu qu'une étape primordiale était l'exécution de l'application. Une bonne et complète exécution permet d'obtenir de meilleurs résultats dans la détection des défauts de code. La validation de DYNAMICS montre qu'il existe un nombre non négligeable de faux négatifs, c'est à dire de défauts de code non-détectés comme défaut de code mais qui auraient dû l'être. Ces faux négatifs proviennent presque tous d'évènements non rencontrés pendant l'exécution, le nombre de ces évènements rencontrés étant assez faible d'après notre validation. La version actuelle de DYNAMICS se base sur différents générateurs d'entrées : MONKEYRUNNER, DROIDBOT et DROIDBOTX. Une perspective envisagée est de travailler sur un générateur d'entrées permettant

d'explorer l'application et de rencontrer plus d'évènements pendant l'exécution. Un générateur d'entrées dédié permettrait donc d'améliorer le rappel.

6.2.3 Utiliser des exécutions réelles plutôt que des simulations d'exécutions

L'étape d'exécution de DYNAMICS se base sur des simulations d'exécutions de cinq ou dix minutes à l'aide de générateurs d'entrées. Nous avons vu que nous envisageons d'améliorer la simulation des exécutions, mais nous envisageons également de prendre de longues traces d'exécutions réelles. Plus précisément, nous envisageons de récupérer des traces d'usages réels en instrumentant des applications fréquemment utilisées et en les partageant à un échantillon d'utilisateurs, par exemple à des étudiants. Pour éviter de surcharger l'appareil d'un usager, il pourrait être possible d'installer les applications instrumentées sur les appareils d'un laboratoire dans le cadre d'une étude. De telles traces générées seraient très intéressantes à étudier et seraient potentiellement plus grandes, avec un grand taux de couverture si l'ensemble des utilisateurs utilisent toutes les fonctionnalités.

6.2.4 Impact des défauts de code comportementaux

Maintenant que nous disposons d'un outil permettant l'identification de défauts de code de façon dynamique dans les applications mobiles, il serait intéressant d'étudier l'impact des défauts de code comportementaux via différentes études. Des études existent déjà pour les défauts de code spécifiques à Android, mais en prenant en compte l'instrumentation, l'exécution, nous pouvons ajouter un autre ensemble de métriques de façon non exhaustives : le nombre d'évènements rencontrés ou la fréquence d'un défaut de code pendant l'exécution. Cela nous permettrait sur un gros volume de données d'analyser la prévalence, c'est-à-dire la fréquence des différents défauts de code, l'impact sur la performance, la consom-

mation énergétique, et tout aspect concernant le comportement.

RÉFÉRENCES

Aljedaani, W., Peruma, A. S., Aljohani, A., Alotaibi, M., Mkaouer, M. W., Ouni, A., Newman, C. D., Ghallab, A. et Ludi, S. (2021). Test smell detection tools : A systematic mapping study. *Evaluation and Assessment in Software Engineering*.

Almalki, K. (2018). *Bad Droid ! An in-depth empirical study on the occurrence and impact of Android specific code smells*. (Thèse de doctorat). Rochester Institute of Technology.

Amin, A., Eldessouki, A., Magdy, M. T., Abdeen, N., Hindy, H. et Hegazy, I. (2019). Androshield : Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. *Inf.*, *10*, 326.

Armando, A., Costa, G. et Merlo, A. (2012). Formal modeling and reasoning about the android security framework. Dans *TGC*.

Bai, G., Ye, Q., Wu, Y., Botha, H., Sun, J., Liu, Y., Dong, J. S. et Visser, W. (2018). Towards model checking android applications. *IEEE Transactions on Software Engineering*, *44*, 595–612.

Banerjee, A., Chong, L. K., Chattopadhyay, S. et Roychoudhury, A. (2014). Detecting energy bugs and hotspots in mobile apps. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

Bartel, A., Klein, J., Traon, Y. L. et Martin, M. (2012). Dexpler : converting android dalvik bytecode to jimple for static analysis with soot. Dans *SOAP '12*.

Battista, P., Mercaldo, F., Nardone, V., Santone, A. et Visaggio, C. A. (2016). Identification of android malware families with model checking. Dans *ICISSP*.

Bauer, A., Küster, J.-C. et Vegliach, G. (2012). Runtime verification meets android security. Dans *NASA Formal Methods*.

Beyer, D., Henzinger, T. A., Jhala, R. et Majumdar, R. (2007). The software model checker blast. *International Journal on Software Tools for Technology Transfer*, *9*, 505–525.

Bierma, M., Gustafson, E., Erickson, J., Fritz, D. et Choe, Y. R. (2014). Andlantis : Large-scale android dynamic analysis. *ArXiv*, *abs/1410.7751*.

Carette, A., Younes, M. A. A., Hecht, G., Moha, N. et Rouvoy, R. (2017). Investigating the energy impact of android smells. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 115–126.

Chambers, C. et Scaffidi, C. (2013). Smell-driven performance analysis for end-user programmers. *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, 159–166.

Chen, H. et Wagner, D. A. (2002). Mops : an infrastructure for examining security properties of software. Dans *ACM Conference on Computer and Communications Security*.

Chen, K. Z., Johnson, N. M., D'Silva, V. V., Dai, S., MacNamara, K., Magrino, T. R., Wu, E. X., Rinard, M. C. et Song, D. X. (2013). Contextual policy enforcement in android applications with permission event graphs. Dans *Network and Distributed System Security Symposium*.

Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby et Zheng, H. (2000). Bandera : extracting finite-state models from java source code. Dans *ICSE*.

Dennis, C., Krutz, D. E. et Mkaouer, M. W. (2017). P-lint : A permission smell detector for android applications. *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 219–220.

Elsayed, E. K., ElDahshan, K. A., El-Sharawy, E. E. et Ghannam, N. E. (2019). Reverse engineering approach for improving the quality of mobile applications. *PeerJ Computer Science*, *5*.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P. D. et Sheth, A. (2014). Taintdroid : an information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, *57*, 99–106.

Espada, A. R., del Mar Gallardo, M., Salmerón, A. et Merino, P. (2015). Runtime verification of expected energy consumption in smartphones. Dans *SPIN*.

Falcone, Y., Currea, S. et Jaber, M. (2012). Runtime verification and enforcement for android applications with rv-droid. Dans *RV*.

Fard, A. M. et Mesbah, A. (2013). Jsnose : Detecting javascript code smells. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 116–125.

Fontana, F. A., Lenarduzzi, V., Roveda, R. et Taibi, D. (2017). Are architectural smells independent from code smells ? an empirical study. *ArXiv*, *abs/1904.11755*.

Fowler, M. (1999). *Refactoring - Improving the Design of Existing Code* (1 éd.). Addison-Wesley.

Gadient, P., Ghafari, M., Frischknecht, P. et Nierstrasz, O. (2018). Security code smells in android icc. *Empirical Software Engineering*, 1–31.

Ghafari, M., Gadient, P. et Nierstrasz, O. (2017). Security smells in android. *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 121–130.

Gottschalk, M., Josefiok, M., Jelschen, J. et Winter, A. (2012). Removing energy code smells with reengineering services. Dans *GI-Jahrestagung*.

Gunadi, H. et Tiu, A. (2013). Efficient runtime monitoring with metric temporal logic : A case study in the android operating system. *ArXiv*, *abs/1311.2362*.

Haase, C. (2015). Developing for android II the rules : Memory. `https://medium.com/google-developers/developing-for-android-ii-bb9a51f8c8b9`. Online, Accédé : Février 2022.

Habchi, S. (2019). *Understanding Mobile-Specific Code Smells*. (Thèse de doctorat). Université de Lille.

Habchi, S., Blanc, X. et Rouvoy, R. (2018). On adopting linters to deal with performance concerns in android apps. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 6–16.

Habchi, S., Hecht, G., Rouvoy, R. et Moha, N. (2017). Code smells in ios apps : How do they compare to android ? *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 110–121.

Habchi, S., Moha, N. et Rouvoy, R. (2019a). The rise of android code smells : Who is to blame ? *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 445–456.

Habchi, S., Rouvoy, R. et Moha, N. (2019b). On the survival of android code

smells in the wild. *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 87–98.

Hallé, S. et Khoury, R. (2017). Event stream processing with beepbeep 3. Dans *RV-CuBES*.

Havelund, K. et Pressburger, T. (2000). Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, *2*, 366–381.

Havelund, K. et Rosu, G. (2001). Monitoring java programs with java pathexplorer. *Electr. Notes Theor. Comput. Sci.*, *55*, 200–217.

Hecht, G. (2016). *Détection et analyse de l'impact des défauts de code dans les applications mobiles. (Detection and analysis of impact of code smells in mobile applications)*. (Thèse de doctorat). Lille 1 en cotutelle avec l'Université du Québec à Montréal.

Hecht, G., Benomar, O., Rouvoy, R., Moha, N. et Duchien, L. (2015a). Tracking the software quality of android applications along their evolution (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 236–247.

Hecht, G., Moha, N. et Rouvoy, R. (2016). An empirical study of the performance impacts of android code smells. *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 59–69.

Hecht, G., Rouvoy, R., Moha, N. et Duchien, L. (2015b). Detecting antipatterns in android apps. *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, 148–149.

Iannone, E., Pecorelli, F., Nucci, D. D., Palomba, F. et Lucia, A. (2020). Refactoring android-specific energy smells : A plugin for android studio. *Proceedings of the 28th International Conference on Program Comprehension*.

Ibrahim, R., Ahmed, M., Nayak, R. et Jamel, S. (2020). Reducing redundancy of test cases generation using code smell detection and refactoring. *Journal of King Saud University - Computer and Information Sciences*, *32*(3), 367–374.

Jin, D., Meredith, P. O., Lee, C. et Rosu, G. (2012). Javamop : Efficient parametric runtime monitoring framework. *2012 34th International Conference on Software Engineering (ICSE)*, 1427–1430.

Jing, Y., Ahn, G.-J. et Hu, H. (2012). Model-based conformance testing for android. Dans *IWSEC*.

Kaelbling, L. P., Littman, M. L. et Moore, A. W. (1996). Reinforcement learning : A survey. *J. Artif. Intell. Res.*, *4*, 237–285.

Kessentini, M. et Ouni, A. (2017). Detecting android smells using multi-objective genetic programming. *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 122–132.

Kumar, S. et Chhabra, J. (2014). Two level dynamic approach for feature envy detection. *2014 International Conference on Computer and Communication Technology (ICCCT)*, 41–46.

Li, Y., Yang, Z., Guo, Y. et Chen, X. (2017). Droidbot : A lightweight ui-guided test input generator for android. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 23–26.

Li, Y., Yang, Z., Guo, Y. et Chen, X. (2019). Humanoid : A deep learning-based approach to automated black-box android app testing. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1070–1073.

Lim, D. (2018). *Detecting code smells in Android applications*. (Mémoire de maîtrise). TU Delft.

Lin, Y., Okur, S. et Dig, D. (2015). Study and refactoring of android asynchronous programming (t). Dans *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 224–235. http://dx.doi.org/10.1109/ASE.2015.50

Lu, Z. et Mukhopadhyay, S. (2012). Model-based static source code analysis of java programs with applications to android security. *2012 IEEE 36th Annual Computer Software and Applications Conference*, 322–327.

Machiry, A., Tahiliani, R. et Naik, M. (2013). Dynodroid : an input generation system for android apps. Dans *ESEC/FSE 2013*.

Mannan, U. A., Ahmed, I., Almurshed, R. A. M., Dig, D. et Jensen, C. (2016). Understanding code smells in android applications. *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 225–236.

Mariotti, G. (2013a). Antipattern : freezing a ui with broadcast receiver. http://gmariotti.blogspot.ca/2013/02/

antipattern-freezing-ui-with-broadcast.html. Online, Accédé : Février 2022.

Mariotti, G. (2013b). Antipattern : freezing the ui with a service and an intentservice. http://gmariotti.blogspot.com/2013/03/antipattern-freezing-ui-with-service.html. Online, Accédé : Février 2022.

Mariotti, G. (2013c). Antipattern : freezing the ui with an asynctask. http://gmariotti.blogspot.com/2013/02/antipattern-freezing-ui-with-asynctask.html. Online, Accédé : Février 2022.

McIlroy, S., Ali, N. et Hassan, A. E. (2015). Fresh apps : an empirical study of frequently-updated mobile apps in the google play store. *Empirical Software Engineering*, *21*, 1346 – 1370.

Morales, R., Saborido, R., Khomh, F., Chicano, F. et Antoniol, G. (2018). [journal first] earmo : An energy-aware refactoring approach for mobile apps. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 59–59.

Nakajima, S. (2013). Model-based power consumption analysis of smartphone applications. Dans *Model Based Architecting and Construction of Embedded Systems @ Model-Driven Engineering Languages and Systems*.

Nakajima, S. (2015). Using real-time maude to model check energy consumption behavior. Dans *International Symposium on Formal Methods*.

Ni-Lewis, I. (2015). Avoiding allocations in ondraw() (100 days of google dev). https://youtu.be/HAK5acHQ53E. Online, Accédé : Février 2022.

Palomba, F., Nucci, D. D., Panichella, A., Zaidman, A. et Lucia, A. D. (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.

Paternò, F., Schiavone, A. G. et Conte, A. (2017). Customizable automatic detection of bad usability smells in mobile accessed web applications. *Proceedings of the 19th International Conference on Human-Computer Interaction with Mobile Devices and Services*.

Pathak, A., Hu, Y. C. et Zhang, M. (2011). Bootstrapping energy debugging on smartphones : a first look at energy bugs in mobile devices. Dans *HotNets*.

Peruma, A. S. (2018). *What the Smell ? An Empirical Investigation on the*

Distribution and Severity of Test Smells in Open Source Android Applications. (Thèse de doctorat). Rochester Institute of Technology.

Pnueli, A. (1977). The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 46–57.

Rahman, A. A. U., Parnin, C. et Williams, L. A. (2019). The seven sins : security smells in infrastructure as code scripts. Dans *ICSE*.

Rasool, G. et Ali, A. (2020). Recovering android bad smells from android applications. *Arabian Journal for Science and Engineering*, *45*. http://dx.doi.org/10.1007/s13369-020-04365-1

Razagallah, A., Khoury, R. et Poulet, J.-B. (2022). Twindroid : A dataset of android app system call traces and trace generation pipeline. Dans *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, p. 591–595., New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/3524842.3528502

Razgallah, A. et Khoury, R. (2021). Behavioral classification of android applications using system calls. Dans *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, 43–52. http://dx.doi.org/10.1109/APSEC53868.2021.00012

Reimann, J., Brylski, M. et Aßmann, U. (2014). A tool-supported quality smell catalogue for android developers. *Softwaretechnik-Trends*, *34*.

Rubin, J., Henniche, A. N., Moha, N., Bouguessa, M. et Bousbia, N. (2019). Sniffing android code smells : An association rules mining-based approach. *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 123–127.

Shin, W., Kiyomoto, S., Fukushima, K. et Tanaka, T. (2009). Towards formal analysis of the permission-based security model for android. *2009 Fifth International Conference on Wireless and Mobile Communications*, 87–92.

Shin, W., Kiyomoto, S., Fukushima, K. et Tanaka, T. (2010). A formal model to analyze the permission authorization and enforcement in the android framework. *2010 IEEE Second International Conference on Social Computing*, 944–951.

Song, F. et Touili, T. (2014). Model-checking for android malware detection. Dans *APLAS*.

Suryanarayana, G., Samarthyam, G. et Sharma, T. (2014). *Refactoring for Software Design Smells : Managing Technical Debt*. Morgan Kaufmann

Publishers Inc.

Thangaveloo, R., Jing, W. W., Leng, C. K. et bin Abdullah, J. (2020). Datdroid : Dynamic analysis technique in android malware detection. *International Journal on Advanced Science, Engineering and Information Technology*, *10*, 536–541.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M. D., Lucia, A. D. et Poshyvanyk, D. (2015). When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, *43*, 1063–1088.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. et Sundaresan, V. (1999). Soot - a java bytecode optimization framework. Dans *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, p. 13. IBM Press.

Yasin, H. N., Hamid, S. H. A. et Yusof, R. (2021). Droidbotx : Test case generation tool for android applications using q-learning. *Symmetry*, *13*, 310.

Zheng, M., Sun, M. et Lui, J. C. S. (2014). Droidtrace : A ptrace based android dynamic analysis system with forward execution capability. *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, 128–133.