UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ÉVALUATION ET CORRECTION DES DÉFAUTS DE CODE LIÉS À LA CONSOMMATION D'ÉNERGIE DANS LES APPLICATIONS MOBILES ANDROID

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MEHDI ADEL AIT YOUNES

AOÛT 2017

REMERCIEMENTS

Je tiens d'abord à remercier Naouel Moha, ma directrice de recherche, pour le temps et la patience qu'elle m'a accordé tout au long de ma maîtrise. Je désire aussi remercier toute l'équipe *Nit* : Alexis, Alexandre, Lucas, Pepos, Romain, Marie-Pier, Fred et Alexandre (le blanc) pour les bons moments passés avec eux durant ces années d'études. J'adresse aussi mes remerciements aux professeurs du département d'informatique Jean Privat, Étienne Gagnon, Marie-Jean Meurs et Abdoulaye Diallo pour toutes les connaissances qu'ils m'ont partagées ainsi que tous les débats constructifs qu'on a pu avoir autour des pizzas des séminaires du LATECE. Je tiens également à remercier les personnes que j'ai pu rencontrer au niveau du département d'informatique, Geoffrey, Antonin, Karim, Aymen, Jehan, Ellen, Nabil, Dylan (plus connus sous le nom de doudou) et bien d'autres. Je remercie aussi toutes les personnes que j'ai pu rencontrer durant mon séjour, mes amis Rafik, Raouf, Samy, Soraya, Reda, Badido, Dalil et bien d'autres ainsi que mon club de volley-ball (Everton). Je désire aussi remercier ma tante, mon oncle, mes cousins : Kenza, Amine, Mehdi (pas moi), Rafik, Achour ; et les amis de mes parents pour m'avoir hébergé, aidé, soutenus et guidé durant mon cursus. Je tiens aussi à remercier très spécialement ***Rapid*** (qui saura s'identifier) qui m'a aidé émotionnellement et financièrement à faire cette maîtrise. À ma sœur et mon frère qui m'ont énormément manqué durant ces années de recherche. Pour finir, je dédie ce mémoire à ma mère et mon père, mes piliers fondamentaux, ainsi que mes moteurs d'inspiration et de motivation tout au long de ma maîtrise : *merci maman, merci papa !*

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

Le périphérique mobile est devenu l'objet indispensable dans la vie de plusieurs personnes. Cet outil aussi petit soit-il a réussi à convertir de nombreux développeurs à l'adopter. Actuellement, l'une des plate-formes les plus prisées par ces derniers est Android. Néanmoins, les développeurs sont souvent confrontés à des contraintes de temps lors du développement de leurs applications. Ils sont donc forcés de développer rapidement les applications afin de respecter les délais de livraison et faire face à la concurrence. Par conséquent, les développeurs se focalisent plus sur les fonctionnalités attendues de l'application et négligent trop souvent des critères primordiaux tels que la performance, la consommation d'énergie et l'expérience utilisateur. Dans le cadre de notre recherche, nous allons nous intéresser spécifiquement au volet de la consommation d'énergie des applications mobiles sous Android. En particulier, nous proposons d'identifier les défauts de code qui ont un impact néfaste sur la consommation énergétique. Notre travail de recherche consiste à évaluer l'impact énergétique des défauts de code Android et de les corriger automatiquement. Les défauts de code sont de mauvaises pratiques d'implémentation dans le code source des applications qui peuvent entraîner des dégradations de la qualité de ces dernières. Dans ce but, nous avons développé l'approche HOT-PEPPER qui actuellement est en mesure d'évaluer et de corriger trois types de défauts de code : *InternalGetter/Setter* (IGS), *HashMapUsage* (HMU) et *Member Ignoring Method* (MIM). L'approche HOT-PEPPER est basée sur deux outils : PAPRIKA pour la détection et la correction des défauts de code ainsi que NAGA VIPER pour l'évaluation de la consommation énergétique. Nous validons notre approche à travers les résultats obtenus lors de nos expérimentations réalisées sur cinq applications. Nous avons observé que la consommation d'énergie sur une application a baissé de 4.83% après la correction des trois défauts de code.

MOTS CLES : Android, défauts de code, consommation d'énergie, applications mobiles, détection, correction.

INTRODUCTION

Le périphérique mobile (en anglais, *Smartphone*) est incontestablement l'une des plus grandes innovations des dix dernières années. Ce dernier existe sous différents systèmes d'exploitation et est commercialisé par plusieurs constructeurs. Dans le cadre de nos recherches, nous avons opté pour le système d'exploitation proposé par Google : Android. Ce dernier connait un franc succès chez les utilisateurs et les développeurs (IDC, 2015).

D'abord, passons brièvement en revue ces dernières années en matière de téléphonie mobile. Apple est le premier à avoir mis sur le marché la nouvelle génération de périphériques mobiles (GPS, appareil photo, écran tactile, ...) : l'iPhone en 2007. En 2005 Android *inc.* est rachetée par Google. Dans un premier temps, Google ne pousse pas Android sur le devant de la scène, il faudra attendre 2007 pour que Google décide de promouvoir Android – son système d'exploitation mobile – sur le marché de la téléphonie afin d'obliger les autres compagnies à améliorer leurs propres systèmes mobiles. Android est un système d'exploitation pour plateforme mobile (tablette, téléphone, montre, etc.). Basé sur un noyau Linux, Android est devenu la distribution la plus populaire[1], avec plus de 50% des périphériques vendus et une hausse de 500% des ventes depuis 2011 (Statista, 2015).

Comme le nombre de périphériques mobiles a augmenté, le nombre d'applications qui vient avec a aussi connu une forte croissance durant ces dernières années ainsi que le nombre de développeurs qui les conçoivent. Les applications mobiles sont gé-

1. `https://goo.gl/ai2yUA`

néralement écrites en utilisant des langages orientés objet comme Java, Objective-C, Swift, ou encore C#. Néanmoins, le développement mobile n'est pas totalement similaire au développement traditionnel (Wasserman, 2010) et cet aspect a obligé les développeurs à prendre en compte les spécifications et les ressources proposées par la plate-forme mobile. Autre contrainte, la demande des utilisateurs qui ne cesse d'augmenter et qui force les développeurs à ajouter de nouvelles fonctionnalités et à maintenir leur application le plus rapidement possible afin de ne pas se faire devancer par d'autres applications similaires. Malheureusement, cette pression sur les développeurs les conduit souvent à travailler dans la hâte et à adopter involontairement des mauvaises pratiques d'implémentation aussi connues sous le nom de *Défauts de code* (Fowler *et al.*, 1999).

Il est estimé que plus de 18% des applications Android présentes sur les magasins en ligne contiennent des défauts de code (Liu *et al.*, 2014). Les défauts de code peuvent facilement impliquer des problèmes au niveau de l'application comme par exemple des pertes de performance au niveau du processeur, de la mémoire, etc (Brylski, 2013). Ces pertes peuvent détériorer la qualité de l'application en terme de stabilité, réactivité, maintenabilité, etc. Des études ont été faites sur l'impact des défauts de code sur les performances d'une application et il en est ressorti que la correction des défauts de code améliore les performances (Hecht *et al.*, 2016a), en particulier la correction de trois types de défauts de code : *HashMap Usage* (HMU), *Internal Getter/Setter* (IGS) et *Member Ignoring Method* (MIM) (Android, 2017b; Android-Doc, 2017a; Hecht *et al.*, 2016a), que nous allons détailler dans les prochains chapitres. Un autre facteur critique au niveau des plate-formes mobiles est l'autonomie de la batterie[2] qui dépend de plusieurs paramètres dont la consommation énergétique des applications.

---

2. `https://developer.android.com/distribute/essentials/quality/core.html`

Dans le cadre de notre étude, nous nous concentrons sur les défauts de code liés à la performance et leur impact au niveau de la consommation d'énergie d'une application Android. L'une des questions qui peut se poser est : Quel est le coût énergétique de ces défauts de code au niveau des applications Android ?

Des études et approches existent afin de suivre la consommation des logiciels traditionnels (Noureddine *et al.*, 2013; Noureddine *et al.*, 2014) et applications mobiles (Dong et Zhong, 2012; Zhang, 2013; Wan *et al.*, 2015). Cependant, peu de recherches ont été effectuées afin d'évaluer l'impact des défauts de code sur la consommation énergétique des applications Android. Les recherches se sont dirigées vers d'autres axes, comme par exemple la consommation énergétique des différentes machines virtuelles qu'a utilisé la plateforme Android tout au long de son évolution (Kundu et Paul, 2011), ou encore au niveau des patrons de conception et de leur impact sur la consommation d'énergie (Linares-Vásquez *et al.*, 2014). D'autres se sont concentrés sur les différentes implémentations d'algorithmes ou de structures de données, comme par exemple les collections Java (Samir *et al.*, 2015), et de mettre en avant leur influence au niveau de l'empreinte énergétique. Des études ont aussi été faites sur la performance des applications mobiles (Hecht *et al.*, 2015a). Un suivi de leur évolution a été fait afin de comparer la performance des différentes versions et de voir les différentes conceptions des développeurs.

Nos recherches visent à atteindre les objectifs cités ci-dessous :

- Proposer une approche automatique pour corriger les défauts de code présents dans les applications Android ;
- Proposer une approche automatique pour évaluer l'impact énergétique des défauts de code.

À travers ce mémoire, nous investiguons les différentes études et méthodes utilisées dans l'évaluation énergétique des applications mobiles afin de proposer une

approche permettant d'évaluer et de mesurer l'impact des défauts de code liés à la performance sur la consommation énergétique d'une application Android. De plus, nous étendons l'outil existant PAPRIKA (Hecht *et al.*, 2015b), initialement conçu pour la détection des défauts de code Android, avec l'ajout d'une phase de correction des défauts de code présents dans les applications. Le travail présenté a fait l'objet d'une publication dans la conférence international *SANER 2017* (*International Conference on **S**oftware **A**nalysis, **E**volution, and **R**eengineering*) (Carette *et al.*, 2017). L'approche développée dans cette recherche se nomme HOT-PEPPER et elle repose sur deux outils : PAPRIKA, pour le processus de détection et correction, et NAGA VIPER, pour la phase de mesure et d'évaluation de l'impact. Les contributions apportées par nos recherches dans ce mémoire sont les suivantes :

- Une catalogue des défauts de code ayant un impact sur les applications Android au niveau de la consommation d'énergie ;
- Une approche automatique de correction des défauts de code présents dans les applications Android ;
- Une approche d'évaluation automatique de l'impact énergétique des défauts de code liés à la performance dans les applications Android ;
- Une étude empirique sur cinq applications Android pour évaluer les approches proposées et une étude de validation sur un cas industriel.

Ce mémoire est structuré de la manière suivante. Le chapitre I décrit brièvement la plate-forme Android et introduit la notion de défauts de code. Le chapitre II est dédié à l'état de l'art réalisé lors de cette recherche. Par la suite, le chapitre III, introduit et explique le fonctionnement et l'implémentation de HOT-PEPPER, l'approche développée lors de cette étude. Dans le chapitre IV, nous décrivons les expérimentations réalisées et les résultats obtenus. Nous concluons ce mémoire par une synthèse du travail effectué et une présentation des travaux futurs.

CHAPITRE I

CONCEPTS PRÉLIMINAIRES

Afin de mieux comprendre notre recherche, nous introduisons dans ce chapitre la notion de défauts de code qui sont les sujets étudiés à travers ce mémoire. De plus, nous présentons, brièvement, le système Android ainsi que l'architecture et le cycle de vie d'une application.

1.1 Les défauts de code

Les défauts de code (Fowler, 1999), ou encore mauvaises odeurs (en anglais *Code Smells*), sont des mauvaises pratiques d'implémentation pouvant dégrader la qualité d'un logiciel et faire apparaître des problèmes au cours du temps. Ces derniers rendent généralement la maintenance et l'évolution du logiciel plus complexe. Par exemple, une méthode ou une classe trop longue et/ou complexe peut être considérée comme un défaut de code (Fowler, 1999; Pérez-Castillo et Piattini, 2014) car cette dernière deviendra difficile à comprendre et à maintenir. Autre exemple, du code mort, du code qui n'est jamais exécuté, peut être aussi considéré comme défaut de code (Mäntylä et Lassenius, 2006) car ce dernier aussi nuit à la compréhension du code et à sa maintenance. Il faut aussi noter que les défauts de code ne sont pas des problèmes mais des indicateurs de future désagréments, ce ne sont pas des bugs qui surviennent lors de la compilation de l'application ou encore lors de son exécution, mais plus des violations de normes et/ou principes

fondamentaux de conception ou d'implémentation (Girish *et al.*, 2015). À ce jour, la notion de défauts de code reste encore très vaste et de nombreuses recherches et études sont encore à faire à fin de déterminer une définition ou une classification propre de ce concept.

1.2 La plate-forme Android

1.2.1 Architecture de la plate-forme

Afin de mieux comprendre les défauts de code présents dans les applications Android, nous devons d'abord introduire l'environnement ou elles sont exécutées, le système d'exploitation Android. Android est un système à code source ouvert, principalement développé pour les périphériques embarqués comme les téléphones et les montres intelligents. Le système Android est basé sur un noyau Linux. La dernière version en date du système est la version 7.1.1 (Nougat). Néanmoins, en raison du périphérique utilisé lors de nos expérimentations, nous avons été contraints d'opter pour la version 4.4.4 (KitKat) qui reste, cependant, assez répondue au niveau des périphériques mobiles avec plus de 20% de présence et représente la troisième distribution Android la plus utilisée après les versions 5.0 (Lolipop) et 6.0 (Marshmallow) [1].

L'architecture des systèmes Android peut être divisée en quatre couches comme illustré dans la figure 1.1.

1. `https://developer.android.com/about/dashboards/index.html`

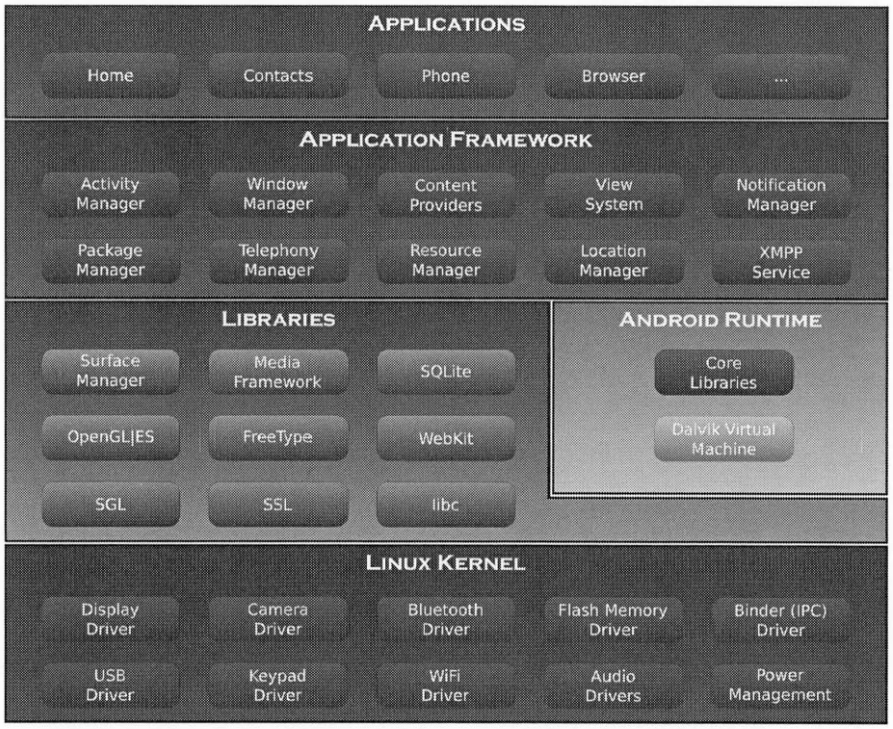

Figure 1.1 Architecture du système Android pour la version 4.4.4 (KitKat) [2].

La couche supérieure, *Applications*, regroupe les applications Android. Les applications développées sont installées au niveau de cette couche. *Application Framework*, est la couche qui regroupe les services, sous forme de classes Java, proposés aux développeurs lors du développement de leur application. La couche inférieure regroupe les librairies offertes par Android en plus des librairies standards, équivalentes au Java SE, qui sont localisées dans le *Core Libraries*. Au niveau de cette couche, plus précisément dans le *Android Runtime*, on retrouve aussi le cadre d'exécution de nos applications Android, *Dalvik Virtual Machine*. Dalvik est une implémentation de la machine virtuelle Java spécialement optimisée et conçue pour les systèmes Android. Dans la dernière couche, *Linux Kernel*, on retrouve le

2. Source : `http://elinux.org/Android_Architecture`

noyau Linux sur lequel le système Android est basé.

1.2.2 Les applications Android

Afin de mieux comprendre le fonctionnement des applications Android, nous allons introduire l'architecture d'une application Android ainsi que certains composants clefs. Les applications Android sont écrites en Java et sont distribuées sous forme de paquets compressés appelés APK (**A**ndroid **P**ackage **K**it). On retrouve dans cet APK le code source compilé de l'application, les ressources, les librairies et le manifest.

Composants d'une application Android

Une application Android est généralement formée d'un ou plusieurs composants spécifiques à la plate-forme : activités (*Activities*), services, des fournisseurs de contenus (*Content providers*), etc. L'activité est un composant Android qui fournit, dans une seule fenêtre, un affichage permettant à l'utilisateur d'interagir avec. La première activité que l'utilisateur aperçoit, au lancement de l'application, est appelée *Main Activity*. Les services eux sont des composants qui tournent pendant une longue durée en tâche de fond. Pour ce qui est des fournisseurs de contenus (*Content providers*), ils sont en charge de gérer les accès à un certain ensemble de données liées à l'application et au périphérique mobile, comme par exemple fournir à l'application la liste des contacts présente sur le périphérique mobile. Un autre concept spécifique à Android et qui permet à plusieurs autres composants de communiquer entre eux, les *Intents*. Les *Intents* sont, de manière simplifié, des messages qui permettent aux composants de communiquer entre eux ou de faire une action définie, comme par exemple démarrer une activité, arrêter un service, etc. Les composants décrits ne représentent pas la totalité des composants disponibles sur une application Android, nous avons décrit ici certains composants formant le cœur d'une application.

Les ressources

Les ressources d'une application représentent généralement les images, les textes traduits, les couleurs, les styles, etc. On retrouve aussi les *layout* qui définissent les interfaces utilisateur de l'application et qui sont déclarés sous forme de fichiers XML. Toutes ces ressources se trouvent dans le dossier *res* de l'application Android. Aussi, afin de pouvoir référer les ressources dans le code source d'une application, Android génère une classe **R** où chaque ressource est affiliée à un identifiant unique (ID).

Android Manifest

L'Android Manifest, présent sous la forme de *AndroidManifest.xml*, est un fichier XML se situant au niveau de la racine de l'application. Il fournis les informations essentielles au système Android avant de pouvoir démarrer l'application :

- le nom de l'application ;
- le nom de l'activité principale (*Main Activity*), elle représente le point d'entrée de l'application ;
- la liste des composants de l'application, les activités, les services, etc ;
- la liste des permissions qui permettent à l'application d'avoir accès à des parties protégées de l'API, comme par exemple permettre à l'application d'avoir accès à Internet ;
- la liste des librairies requises ;
- etc.

Cycle de vie d'une activité

Nous décrivons ici le cycle de vie des activités dans les applications Android. Les activités Android ont des états variables qui changent en fonction de l'environnement (le système Android) ou des actions utilisateur. Ces changements d'état

sont illustrés dans la Figure 1.2 et expliqués, brièvement, ci-dessous :

— **onCreate()** est la première méthode appelée au lancement, la création, d'une activité, elle initialise les *layout* de l'application en plus d'autres paramétrés. Il est généralement conseillé aux développeurs d'implémenter leurs initialisations de départ au niveau de cette méthode, car elle n'est appelée qu'une fois durant le cycle de vie de l'activité ;

— **onStart()** est la méthode appelée quand l'activité passe à l'état *Started.* À ce moment là, l'activité devient visible pour l'utilisateur et se prépare à passer au premier plan et à devenir interactive ;

— **onResume()** est appelée quand l'activité est dans l'état *Resumed.* L'activité est au premier plan et interagie avec l'utilisateur ;

— **onPause()** est la méthode appelée quand l'activité n'est plus au premier plan, elle passe donc à l'état de *Paused.* Plusieurs événements peuvent mettre l'activité dans cet état, comme par exemple l'ouverture d'une fenêtre de dialogue ;

— **onStop()** est la méthode appelée quand l'activité passe à l'état *Stopped.* Cet état se produit quand l'activité n'est plus visible pour l'utilisateur, par exemple quand une autre activité est démarrée ;

— **onDestroy()** est appelée avant la destruction de l'activité. Cette méthode est appelée une fois l'activité terminée ou lors de la destruction de l'activité par le système.

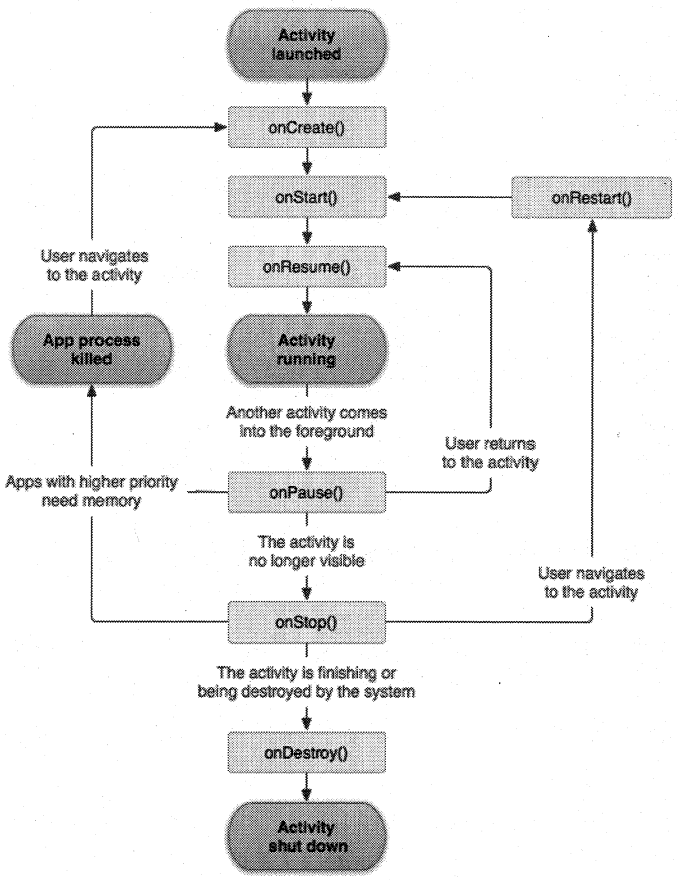

Figure 1.2 Cycle de vie d'une application d'Android [3].

1.3 Conclusion

Lors de ce chapitre, nous avons décrit la notion de défauts de code afin de pouvoir mieux comprendre ce concept et notre axe de recherche. Nous avons aussi présenté et introduit le système d'exploitation Android qui représente le cadre d'exécution des applications étudiées lors de cette recherche. Aussi, afin d'avoir une idée globale sur le paradigme de programmation Android, nous avons expliqué, brièvement, la structure d'une application Android ainsi que certains de ces

3. Source : `https://developer.android.com/guide/components/activities`

composants. Dans le prochain chapitre, nous allons présenter l'état de l'art que nous avons construit à travers les recherches effectuées sur : les défauts de code Android, l'aspect énergétique au niveau des applications mobile (causes des pertes énergétique et méthode d'analyse) et la détection et correction des défauts de code Android.

CHAPITRE II

ÉTAT DE L'ART

L'aspect énergétique commence à émerger dans le domaine du développement logiciel, que ce soit dans les applications à client lourd, les applications web ou les applications mobiles. Nous abordons à travers cet état de l'art les principaux axes d'études qui nous permettent de construire et de présenter notre recherche. Nous commençons par présenter les différents défauts de code identifiés dans les applications mobiles Android ainsi que ceux ayant un impact connu sur la performance. Nous investiguons aussi les mauvaises pratiques d'implémentation qui conduisent aux fuites énergétiques dans les applications Android. Nous parcourons les études et méthodes d'analyse existantes sur la consommation énergétique des applications Android. Nous concluons cet état de l'art par une étude des différent(e)s recherches et outils existant(e)s sur la détection et la correction des défauts de code.

2.1 Défauts de code Android

Les défauts de code sont des mauvaises pratiques qui apparaissent facilement lors du développement d'une application indépendamment de la plate-forme utilisée. Néanmoins, chaque plate-forme de développement peut être sujette à des types bien spécifiques de défauts de code qui ne peuvent apparaître qu'avec certaines spécificités ou éléments liés à la plate-forme. Dans le cas de la plate-forme Android,

les défauts de code liés au domaine de l'**OO** (Orienté Objet) sont souvent retrouvé mais il en existe d'autres bien spécifiques aux composants de l'API (Interface de Programmation Applicative) ou aux optimisations faites par la machine virtuelle, les *défauts de code Android.* Un catalogue des défauts de code Android a été produit (Brylski, 2013) afin de répertorier ces derniers et de proposer leur ré-usinage. La catalogue répertorie 30 défauts de code catégorisés en cinq catégories : implémentation, réseau, interface utilisateur (UI), base de donnée, et entrées/sorties. La majorité des défauts de code répertoriés sont des défauts d'implémentation.

En partant de ce catalogue, de nombreuses recherches ont été effectuées sur l'impact de ces défauts de code sur la performance des applications Android. La documentation d'Android a mis à disposition des développeurs une liste de mauvaises pratiques qui affectent négativement les performances des applications (Android, 2017b). Trois des défauts de code cités dans le catalogue sont présents dans la documentation : *Internal Getter/Setter (IGS), Member Ignoring Method (MIM), Slow Loop.* La documentation d'Android met aussi en avant d'autres mauvaises pratiques pouvant avoir un impact non négligeable sur les performances : éviter l'utilisation de flottants (Android, 2017c), la façon d'utiliser les méthodes natives (Android, 2017e), ou encore l'utilisation des `ArrayMap` à la place des `HashMap` connus aussi sous le nom de *HashMap Usage (HMU).* Une étude plus détaillée a été faite sur l'impacte des IGS, des HMU et des MIM sur la performance des applications (Hecht *et al.*, 2016b). Il en est ressorti qu'avec la correction des MIM, il est possible d'améliorer les performances d'affichage de 12,4% et la performance globale de l'application jusqu'à plus de 3,6% lors de la correction des trois défauts de code. D'autres études, un peu plus générales, ont été faites sur les causes de la dégradation de la performance dans les applications Android (Liu *et al.*, 2014; Li et Halfond, 2014) ou sur la différence de performance entre le développement traditionnel et le développement natif (Lin *et al.*, 2011; Lee et Jeon, 2010).

Notre axe de recherche s'est orienté vers les défauts de code d'implémentation liés à la performance afin de corréler l'aspect de la performance avec l'aspect énergétique. Nous avons choisi d'investiguer les défauts de code suivants : *Internal Getter/Setter (IGS), Member Ignoring Method (MIM)* et *HashMap Usage (HMU).*

2.1.1 Internal Getter/Setter

Les IGS sont des défauts de code qui se produisent lorsqu'un attribut de classe est utilisé, dans la classe déclarante, via un accesseur (`var = getField()`) et/ou un mutateur (`setField(var)`). Cet accès indirect à l'attribut peut diminuer les performances de l'application. L'utilisation d'IGS est une pratique courante dans des langages OO tels que C ++, C# ou Java, car les compilateurs et/ou les machines virtuelles sont optimisés et peuvent linéariser (opération d'*inlining*) l'accès. Cependant, la machine virtuelle d'Android ne fait que des optimisations basiques (Ben, 2011), elle est donc incapable de linéariser les accès. Par conséquent, l'utilisation d'accesseur ou de mutateur trivial est souvent convertie en appel de méthode virtuelle, ce qui rend l'opération au moins **trois fois** (Android, 2017a) plus lente qu'un accès direct. L'IGS peut être corrigé en accédant à l'attribut de classe directement (`var = this.MyField`, `this.MyField = var`) ou en déclarant les méthodes d'accesseur et de mutateur dans une interface publique afin de supprimer l'appel virtuel (Android, 2017b; Brylski, 2013). Néanmoins, les accesseurs et mutateurs non triviaux, comme illustré dans Listing 2.1, ne sont pas considérés comme étant des IGS.

Listing 2.1 Exemple d'un accesseur non trivial dans l'application SoundWaves Podcast étudiée.

```
public String getURL() {
  String itemURL = "";
  if (this.url != null && this.url.length() > 1)
    itemURL = this.url;
  else if (this.resource != null && this.resource.length() > 1)
    itemURL = this.resource;
  return itemURL;
}
```

2.1.2 Member Ignoring Method

Dans les applications Android, lorsqu'une méthode n'accède pas à un attribut d'objet ou n'est pas un constructeur, il est recommandé d'utiliser une méthode statique pour augmenter les performances. Selon le guide de recommandations officielles d'Android (Android, 2017b), les invocations de méthodes statiques seraient plus rapide d'environ 15% à 20% que les invocations dynamiques (Android, 2017b). De plus, l'utilisation d'une méthode statique est également considérée comme une bonne pratique pour la lisibilité du code, car elle garantit que l'appel de la méthode ne modifiera pas l'état de l'objet (Android, 2017b; Brylski, 2013). Cependant, il existe possiblement un effet de bord en termes d'héritage puisque toutes les classes étendues doivent déclarer ou se référer aux mêmes méthodes statiques. Listing 2.2 est un exemple de MIM dans une des applications étudiées.

Listing 2.2 Exemple de Member Ignoring Method dans l'application SoundWaves Podcast.

```
private boolean animationStartedLessThanOneSecondAgo(long lastDisplayed)
    {
  return System.currentTimeMillis() - lastDisplayed < 1000 &&
      lastDisplayed != -1;
}
```

2.1.3 HashMap Usage

L'API d'Android offre les `ArrayMap` et `SimpleArrayMap` en remplacement aux traditionnels `HashMap` fournis par Java. Selon la documentation d'Android (Android-Doc, 2017a), ils sont supposés être plus efficaces sur le plan de la mémoire et déclenchent moins le ramasse-miettes (*Garbage Collector*). De plus, leurs utilisations n'a aucun impact significatif sur les performances des opérations effectuées sur une `Map` contenant plus d'une centaines d'entrées (Android-Doc, 2017a). Ainsi l'utilisation de l'`ArrayMap` devrait être préférée à l'utilisation du `HashMap` pour les `Map` ne contenant pas est un grand nombre d'éléments. Par conséquent, la création de petites instances `HashMap` peut être considérée comme un défaut de code (Android-Doc, 2017a; Haase, 2015). Cependant, une dégradation des performances lors de l'utilisation d'un `ArrayMap` peut se produire en faisant face à une croissance imprévue du nombre d'éléments de la `Map`. Listing 2.3 est un exemple de HMU détecté dans une des applications étudiées.

Listing 2.3 Exemple de HMU dans l'application SoundWaves Podcast étudiée.

```
if (itemMap == null) {
    itemMap = new HashMap<>();
    for (int i = 0; i < ItemColumns.ALL_COLUMNS.length; i++)
      itemMap.put(ItemColumns.ALL_COLUMNS[i], ItemColumns.ALL_COLUMNS[i]);
}
```

2.2 Analyse de la répartition de la consommation énergétiques au niveau des applications mobiles

L'aspect énergétique est devenu un facteur clef dans le développement d'applications en tout genre et spécifiquement dans les applications mobiles, en raison de la faible capacité des batteries présentes sur ces périphériques. Plusieurs recherches se sont donc focalisées sur les principaux composants des périphériques mobiles ayant un fort impact sur la consommation énergétique. L'une des principales causes trouvée est l'écran (Li *et al.*, 2014), qui représente en moyenne plus de 60% de la consommation d'une application (Dong et Zhong, 2012). Une partie de la littérature scientifique a donc investigué l'impact de l'écran sur la consommation des applications mobiles en proposant par exemple des moyens d'identifier les principales zones de l'écran d'une application ayant une haute consommation (Wan *et al.*, 2015). Différentes recommandations et bonnes pratiques ont aussi été faites sur les couleurs à utiliser dans les applications en fonction du type d'écran du périphérique (Dong et Zhong, 2012; Zhang, 2013) : diode électroluminescente organique (DELo) ou écran à cristaux liquides (ACL, affichage à cristaux liquides). Il a été prouvé lors de ces études que les couleurs sombres consomment moins d'énergie que les couleurs vives dans le cas des écrans DELo et inversement dans les cas des écrans ACL.

D'autres recherches ont été menées sur l'impact des communications réseaux ef-

fectuées par les applications et il a été prouvé que les composants liés au réseau (carte Wi-Fi et données GSM) sont la plus grosse source de consommation d'énergie, après l'écran, au niveau des applications mobiles (Li *et al.*, 2014). À partir de ces résultats, des recherches se sont focalisées sur la façon dont les communications réseaux sont faites (Balasubramanian *et al.*, 2009; Duribreux *et al.*, 2014) afin de pouvoir proposer par la suite des modèles et des optimisations permettant de réduire le coût énergétique (Marcu et Tudor, 2011; Li *et al.*, 2016). Plus spécifique à la plate-forme Android, de nombreux travaux se sont concentrés sur les fuites énergétiques causées par l'utilisation excessive de certains services système comme les *Wakelocks*[1] (Banerjee *et al.*, 2016; Gottschalk *et al.*, 2012) ou l'intégration abusif des publicités (Gui *et al.*, 2015; Gui *et al.*, 2016) dans les applications.

Néanmoins, il existe peu de ressources traitant de l'impact énergétique des défauts de code dans les applications Android. Deux études scientifique se sont dirigées vers le défauts de code Android répertoriés dans le catalogue cité dans la Section 2.1. Pérz-Castillo et Piattini (Pérez-Castillo et Piattini, 2014) ont étudié le contre-coup énergétique du ré-usinage des *God Class*[2] dans les applications Android. Il s'est avéré que le ré-usinage des *God Class* crée une augmentation de la consommation énergétique des applications testées allant jusqu'à 20,1%. L'origine principale de cette hausse est lié à l'augmentation du nombre de classes et de méthodes causée par le ré-usinage.

L'équipe de Li (Li et Halfond, 2014) ont étudié l'impact de la correction des IGS

---

1. Les *Wakelocks* est une permission attribuée aux applications afin que ces dernières puissent avoir accès au CPU et ce même après la mise en veille du périphérique, comme par exemple la synchronisation des mails.

2. La *God Class*, aussi connus sous le nom de *Blob*, est une grande classe avec un grand nombre de méthodes et attributs qui est fréquemment utilisée comme contrôleur dans un projet.

et MIM au niveau de la consommation des applications Android. Leur étude a démontré que la correction des IGS et des MIM peut réduire la consommation énergétique de 35% et 15%, respectivement. Dans notre cas, la correction des IGS et des MIM réduit la consommation énergétique de 2.08% et 3.86% respectivement. Toutefois, cet important écart avec nos résultats s'explique du fait de la différence du contexte expérimentale. De notre côté, nous avons exécuté nos tests sur un ensemble d'applications réelles disponibles en ligne tandis que leur expérimentation est basée sur une application développée exclusivement pour leur essai. D'autre part, nous nous sommes basés sur des scénarios orientés utilisateur lors de l'exécution des applications alors que de leur côté ils stressent leur application avec une boucle de 50 millions d'itérations sur l'appel des défauts de code. Par ailleurs, nous investiguons un défaut de code supplémentaire (HMU). Une autre publication a été réalisée sur l'amélioration de l'efficacité énergétique en utilisant des services de ré-usinage (Gottschalk *et al.*, 2012) qui s'appliquent sur les défauts de code présents au niveau des applications Android. Cependant, les défauts de code qu'ils définissent sont différents des notre, à savoir des bugs de boucles, du code mort, de l'*inlining* de méthode et l'utilisation du cache

2.3 Méthodes d'analyse de la consommation énergétique des applications Android

Nous pouvons diviser les études effectuées sur les méthodes d'analyse en deux grandes familles : analyse *statique* et analyse *dynamique*. Les méthodes d'analyse statiques ne sont pas très nombreuses et sont généralement basées sur l'extraction d'informations à partir de structures de données comme les *API Invocation Tree* (Li *et al.*, 2016; Zhang *et al.*, 2010) ou les *function call graph* (Guo *et al.*, 2013). L'équipe de Gue (Guo *et al.*, 2013) ont développé *Relda*, un outil d'analyse statique qui permet de détecter les ressources (Wi-Fi, camera, audio, ... etc) ayant une forte consommation lors de l'exécution d'une application en se basant

sur un *function call graph* modifié qui récupère la gestion des événements faite par la plate-forme Android. L'analyse dynamique reste la méthode la plus courante dans la littérature scientifique. Ce genre de méthodes calculent les métriques nécessaires à l'exécution de l'application via du profilage (Hao *et al.*, 2012), l'utilisation de scripts (Hindle *et al.*, 2014; Li et Halfond, 2014; Li *et al.*, 2013), la récupération d'appels système (Pathak *et al.*, 2011), l'exécution d'applications ou services en fond (Zhang *et al.*, 2010), ou encore de l'émulation (Mittal *et al.*, 2012; Marcu et Tudor, 2011) afin de récupérer les métriques à partir du moniteur de l'émulateur. Jabbarvand et son groupe (Jabbarvand *et al.*, 2015) ont utilisé l'analyse statique et dynamique pour détecter les fuites énergétiques sur les applications afin de les classer par la suite. L'analyse statique a été effectuée afin d'annoter le graphe d'appels alors que l'analyse dynamique a été utilisée pour du profilage sur l'application.

Au delà du type d'analyse effectuée pour évaluer la consommation énergétique d'une application, nous nous apercevons à travers les travaux réalisés sur le sujet qu'il est d'abord question de récupérer les métriques énergétiques et pour ce faire, nous répertorions deux techniques : l'utilisation de modèle énergétique (Hao *et al.*, 2012; Pathak *et al.*, 2011; Jabbarvand *et al.*, 2015) ou l'utilisation d'appareils physiques de mesures (Duribreux *et al.*, 2014; Linares-Vasquez *et al.*, 2014; Hindle *et al.*, 2014). Les inconvénients des modèles énergétiques sont leur manque de précision et le risque potentiel de créer des effets de bord sur la consommation lors du processus de récolte d'informations (construction du modèle). Les appareils de mesures physiques représentent la meilleure solution en terme de précision. Dans le cas de cette solution, Monsoon (Inc, 2017) est l'appareil de mesures qui apparaît le plus souvent dans les revues scientifiques. Néanmoins, le prix de ce dernier n'est pas forcément accessible ($829.00 USD au moment de la réalisation de nos expérimentations). Dans le cadre de notre recherche, nous avons choisi le

YOCTO-AMP [3] comme appareil de mesure afin de récupérer l'intensité du périphérique au cours de l'exécution d'une application, puis d'analyser dynamiquement la consommation énergétique de cette dernière. Le YOCTO-AMP représente un bon compromis, car il nous permet de passer outre la limite financière tout en offrant une excellente précision (2 mA, 1%).

2.4 Détection et correction des défauts de code Android

A l'exception de LINT (Android, 2017d), proposé avec le *Android Developer Tools* (ADT), et de PAPRIKA, il n'existe pas d'outils dédiés à la plate-forme Android pour l'analyse et la correction des défauts de code. Pour ce qui est de la détection, en étudiant de plus proche la documentation de LINT, nous découvrons qu'il ne peut détecter qu'un seul type de défaut de code étudié (IGS). De son coté, PAPRIKA est en mesure de détecter 17 types de défauts de code différents, dont les trois défauts de code Android précédemment cités dans la Section 2.1. Par contre, à date, il n'existe pas d'outils de correction de ces défauts de code. Toutefois, les applications Android sont majoritairement écrites en Java qui est un langue OO. Il existe des outils et des recherches sur l'analyse et la correction du code Java. Pour effectuer la correction et l'analyse du code source des applications, nous recherchons des outils qui peuvent effectuer deux types d'actions : de l'*analyse de code* et de la *transformation de code.*

L'analyse de code est utilisée dans plusieurs spécialités : la compilation, le débogage (Hovemeyer et Pugh, 2004), la détection de mauvaises pratiques (Binkley, 2007) etc. PMD [4] et CheckStyle [5] sont des outils d'analyse de code source Java qui

---

3. `http://www.yoctopuce.com/EN/products/usb-electrical-sensors/yocto-amp`

4. `https://pmd.github.io/`

5. `http://checkstyle.sourceforge.net/`

permettent la détection de mauvaises pratiques. Or, la liste des défauts de code détectés par ces outils n'est pas très large et est plus orientée vers les mauvaises pratiques OO. Par exemple, ils peuvent détecter les *God Class*, *Long Method*, *Long Param List* et *Duplicated Code*. Il existe également des environnements spécifiques et des plate-formes dédiées à l'analyse de logiciels comme IPlasma (Marinescu *et al.*, 2005), qui est un environnement spécifique pour l'analyse qualitative de logiciels ou encore DECOR (Moha, 2007) pour la détection et la correction des anti-patrons OO.

La transformation de code est utilisée dans de nombreux domaines comme l'optimisation (Loveman, 1977) ou le ré-usinage de code source (Kniesel et Koch, 2004). Il existe des outils et des études sur le ré-usinage du code Java lié aux défauts de code. JDEODORANT (Fokaefs *et al.*, 2011) est un plugin ECLIPSE qui détecte et corrige les défauts de code. Il est basé sur ASTPARSER et ASTRWRITER présent dans l'API de développement d'ECLIPSE. Néanmoins, JDEODORANT peut détecter et corriger seulement cinq défauts de code qui ne sont pas spécifiques à Android. D'autres travaux existent sur la transformation du code non spécifique aux défauts de code. Par exemple, Tatsubori et sont équipe ont développé OPENJAVA (Tatsubori *et al.*, 1999), un système de macro basé sur un méta-modèle pour représenter la structure du code source en utilisant le concept de réflexion. Borba lui a travaillé sur CODER (Borba, 2002), un outil de génération, de maintenance et de ré-usinage de programmes Java. De nombreux outils sont disponibles pour la génération et la transformation de code source Java comme Java Poet [6], le successeur de JavaWriter [7], qui est une API pour la génération de fichiers source Java ou

6. `https://github.com/square/javapoet`

7. `https://github.com/square/javapoet/tree/javawriter\_2`

encore THE VELOCITY ENGINE[8], un des projets Apache Velocity, qui utilise des templates, codé avec leur propre langage, pour effectuer de la génération de code. Stratego[9], qui fait maintenant partie du langage Spoofax Language Workbenchis, est un langage dédié à la transformation de code. Dans la transformation de bytecode Java, ASM (Bruneton *et al.*, 2002) est un framework qui peut effectuer des analyses et des modifications directement sur le bytecode des applications. Zhang et son groupe ont développé DPartner (Zhang *et al.*, 2012). C'est un outil destiné à analyser et transformer le bytecode d'application Android. Cet outil a été utilisé afin d'injecter des patrons spécifiques ayant pour objectif de permettre à l'application d'effectuer du *computation offloading*[10].

Dans le cadre de notre travail, nous avons opté pour PAPRIKA, car c'est le seul outil qui permet de détecter les défauts code Android cités dans la Section 2.1. Comme expliqué précédemment, les défauts de code IGS, MIM et HMU sont spécifiques à Android. Or, tous les outils et travaux de recherche sur les défauts de code décrits, y compris DECOR et JDEODORANT, sont dédiés uniquement aux défauts de code orientés objet. Pour ce qui est de la transformation de code source, nous avons choisi SPOON (Pawlak *et al.*, 2015), une librairie d'analyse et de transformation de code source. Nous décrivons SPOON plus en détail dans la Section 3.1 du Chapitre 3. SPOON offre une compréhension et une manipulation aisées de l'AST des applications comparativement à ASM qui travaille sur le bytecode ou *The Velocity Engine* qui utilise un langage spécifique afin de manipuler le modèle de l'application. SPOON offre également des performances très intéressantes lors

---

8. `http://velocity.apache.org/`

9. `http://www.metaborg.org/en/latest/`

10. La capacité de transférer une partie des tâches ou opérations à une plate-forme distante, par exemple en *cloud*

de la construction de son modèle. Il lui faut moins d'une seconde pour construire les modèles d'applications comme Minecraft [11] (0.7s), JavaWriter (0.8s) ou encore Scribe Java [12] (0.5s). Néanmoins, il existe une corrélation entre la taille de l'application, les fonctionnalités du langage Java et le temps prit par SPOON pour créer les modèles. Par exemple, il faut plus de temps à SPOON pour construire les modèles de JUnit [13] (2.5s) ou Joda Time [14] (5s) car les applications sont grandes et complexes. Cependant, les applications Android ne sont généralement pas aussi grandes que les applications à client lourd, SPOON est donc adéquat pour ce genre de programme.

2.5 Conclusion

En résumé, cet état de l'art met en avant les différentes recherches dans l'analyse des défauts de code Android ainsi que leur impact au niveau de la consommation énergétique des applications. Peu d'études existent sur les conséquences énergétiques de la présence des défauts de code Android. Notre travail de recherche a donc pour objectif de pallier ce manque. Nous avons orienté nos travaux sur l'effet des IGS, HMU, et MIM dans la consommation des applications Android. De plus, nous avons opté pour l'emploi d'analyses dynamiques et l'utilisation de dispositif physique de mesure pour l'évaluation des défauts de code énergétiques. Pour conclure, nous avons choisi PAPRIKA comme outil de détection des défauts

---

11. `https://minecraft.net/en/`

12. `https://github.com/scribejava/scribejava`

13. `http://junit.org/junit4/`

14. `http://www.joda.org/joda-time/`

de code et l'avons étendu avec SPOON afin de pouvoir corriger ces derniers. Dans le prochain chapitre, nous introduirons HOT PEPPER, l'approche développée et utilisée dans notre recherche afin d'évaluer l'impact énergétique des défauts de code.

CHAPITRE III

HOT-PEPPER : AMÉLIORATION DE LA CONSOMMATION ÉNERGÉTIQUE DES APPLICATIONS ANDROID

HOT-PEPPER est une approche que nous proposons visant à détecter et corriger automatiquement les défauts de code présents dans les applications Android ainsi que d'évaluer l'impact énergétique de la correction appliquée. L'approche HOT-PEPPER repose sur deux outils : PAPRIKA (Hecht *et al.*, 2015a) et NAGA VIPER (Carette *et al.*, 2017) que nous présenterons successivement dans les deux sections suivantes. PAPRIKA est en charge de détecter et de corriger les défauts de code présents dans l'application, NAGA VIPER, quant à lui, est en charge de mesurer et d'évaluer l'impact en terme de consommation énergétique de la correction effectuée avec PAPRIKA au niveau de l'application (Figure 3.1).

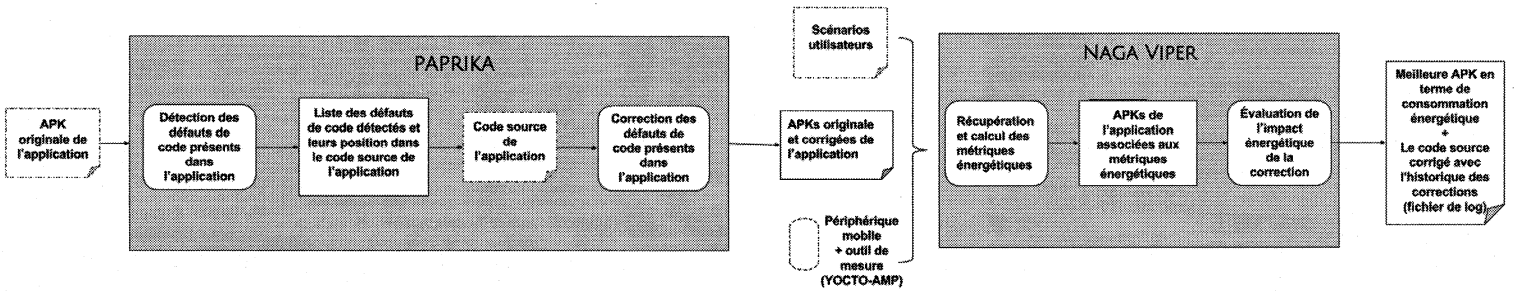

Figure 3.1 Vue d'ensemble de l'approche Hot-Pepper.

3.1 PAPRIKA : Détection et correction automatiques des défauts de code

PAPRIKA est un outil d'analyse statique, développé à l'UQAM, qui permet la détection des défauts de code Android (Hecht *et al.*, 2015a). PAPRIKA effectue son analyse directement sur l'APK de l'application. Dans cette étape, PAPRIKA prend comme entrée l'APK et exécute la phase d'analyse, qui consiste à détecter les défauts de code dans l'application. Pour effectuer son analyse, PAPRIKA doit construire un modèle de l'application basé sur huit entités : `App`, `Class`, `Method`, `Attribute`, `Variable`, `ExternalClass`, `ExternalMethod` et `ExternalArgument`. PAPRIKA est basé sur la plate-forme SOOT (Vallée-Rai *et al.*, 1999) et son module DEXPLER (Bartel *et al.*, 2012) pour analyser l'APK afin de construire le modèle de l'application. SOOT convertit le bytecode des applications en une représentation interne, composée des entités précédentes, qui sont par la suite représentées sous forme d'un graphe NEO4J (Neo4J, 2017) où chaque nœud du graphe représente une entité. PAPRIKA effectue des requêtes sur le graphe NEO4J en utilisant le langage CYPHER (Query, 2017) (Listing 3.1) pour détecter les défauts de code présents dans l'application.

Listing 3.1 Requête Cypher pour la détection des Internal Getter/Setter (IGS).

```
MATCH (m1:Method)-[:CALLS]->(m2:Method), (cl:Class)
WHERE (m2.is_setter OR m2.is_getter)
AND cl-[:CLASS_OWNS_METHOD]->m1
AND cl-[:CLASS_OWNS_METHOD]->m2
RETURN m1
```

Une fois l'application analysée, PAPRIKA renvoie la liste des défauts de code détectés, leur proportion, leur type, et leur emplacement dans le code source de l'application.

Nous avons par la suite étendu PAPRIKA afin que ce dernier puisse aussi corriger

automatiquement les défauts de code détectés. Dans le but d'offrir à PAPRIKA la possibilité d'effectuer des corrections au niveau du code source de l'application, nous avons utilisé SPOON (Pawlak *et al.*, 2015), une librairie d'analyse statique et de transformation de code source JAVA. Pour effectuer les transformations de code, SPOON utilise le code source de l'application afin de générer un méta-modèle qui peut être vu comme un `AST` (*Abstract Syntax Tree*) du code source de l'application et où chaque nœud correspond à un élément source du programme : `Classe`, `Méthode`, `Attribut`, `Déclaration`, `Expression`, `Type`, `Annotation`, ... etc. SPOON offre la possibilité de parcourir ses éléments à travers des processeurs SPOON (peuvent être vus comme des visiteurs de l'AST).

Les processeurs SPOON permettent d'effectuer des analyses et des transformations sur le type d'éléments qu'ils parcourent : vérification de type, changement d'un type de variable, modification de la portée d'un attribut, ajout d'un bloc de code, etc. L'écriture des processeurs SPOON se fait en JAVA. Voir ci-dessous (Listing 3.2) un exemple de processeur SPOON parcourant les éléments de type `CtMethod` qui représentent les méthodes d'un programme et vérifiant si le corps de celles-ci est vide.

Listing 3.2 Exemple de processeur Spoon permettant la détection de méthodes avec un corps vide.

```
public class MethodBodyProcessor extends AbstractProcessor<CtMethod> {
  public void process(CtMethod element) {
    if(element.getBody().getStatements().size() == 0)
    { getEnvironment().report(this, Level.WARN, element,
        element.getReference()); }
  }
}
```

En ce qui concerne la correction des défauts de code, nous avons intégré à PAPRIKA un processeur SPOON par type de défaut de code :

— Un processeur SPOON pour la correction des IGS ;

— Un processeur SPOON pour la correction des HMU ;

— Un processeur SPOON pour la correction des MIM ;

En conclusion, pour une application donnée (APK et code source de l'application), PAPRIKA détecte les défauts de code présents en effectuant son analyse sur l'APK de l'application. À partir des informations récoltées lors de la phase de détection, PAPRIKA lance les trois processeurs de correction indépendamment afin de générer autant d'APK que de type de défaut de code présents dans l'application. Les différents APK de l'application permettent par la suite à NAGA VIPER d'évaluer l'impact énergétique de la correction de chaque type de défaut de code. PAPRIKA lance ensuite les trois processeurs de correction simultanément afin de générer une version de l'application sans aucun défaut de code.

3.1.1 Correction de Internal Getter/Setter (IGS)

Afin de corriger les IGS présents dans l'application, le processus de correction procède au remplacement des appels aux accesseurs / mutateurs (`this.myGetter()` / `this.mySetter(arg1)`) par des accès directs aux attributs de la classe.
L'algorithme 1 représente le fonctionnement du processeur SPOON en charge de la correction des IGS. L'implémentation de l'algorithme est disponible sur GitHub [1].
Le processeur prend comme entrée le fichier d'analyse de PAPRIKA (Figure 3.2) contenant les classes et les invocations de méthodes faisant appel à un IGS en plus du code source de l'application.

---

1. `https://github.com/MehdiAit/spoon-processors/blob/master/src/main/java/io/paprika/spoon/InvokMethodProcessor.java`

full_name	gs_name
stop#org.bottiger.podcast.Player.SoundWavesPlayer	isInitialized#org.bottiger.podcast.Player.SoundWavesPlayer
compareTo#org.bottiger.podcast.provider.QueueEpisode	getPriority#org.bottiger.podcast.provider.QueueEpisode
toString#org.bottiger.podcast.provider.QueueEpisode	getPriority#org.bottiger.podcast.provider.QueueEpisode

Figure 3.2 Exemple de fichier d'analyse de Paprika pour la détection des IGS répertoriant les noms des classes et méthodes contenant des IGS (colonne 1) et le nom de l'accesseur/mutateur appelé (colonne 2).

Le processeur de correction parcourt toutes les classes répertoriées dans le fichier d'analyse de PAPRIKA (Algorithme 1 : Ligne 1). Pour chaque classe, le processeur examine les *invocations* de méthodes (*appels de méthodes*) de cette dernière afin de les comparer et de les faire correspondre avec le fichier d'analyse de PAPRIKA (Algorithme 1 Ligne 3). Par la suite, le processeur confirme si l'IGS rencontré fait appel à un accesseur ou à un mutateur. Pour cela, le processeur analyse la signature de la méthode invoquée afin d'évaluer le nombre de paramètres qu'elle possède. Si la méthode contient un ou plusieurs paramètres, la processeur l'identifie comme étant un mutateur (`public void mySetter(arg1)`), dans le cas contraire elle est identifiée comme accesseur (`public int myGetter()`) (Algorithme 1 Lignes 5-11). Une fois l'IGS correctement identifié, le processeur commence les transformations. Dans le cas d'un accesseur, le processeur analyse le corps de l'accesseur et récupère la variable (*attribut de la classe*) que renvoie ce dernier (`return myField`) en utilisant les filtres SPOON (Listing 3.3).

Listing 3.3 Filtre Spoon pour la récupération de l'attribut de classe retourné.

```
method.getBody().getLastStatement().getElements(new
    AbstractFilter<CtReturn>(CtReturn.class) {
  @Override
  public boolean matches(CtReturn element) {
    getField = element.getReturnedExpression().toString();
    return super.matches(element); }
});
```

Par la suite, le processeur transforme l'appel à l'accesseur par un accès direct à la variable précédemment récupérée (`var = this.myGetter() -> var = myField`) (Algorithme 1 Lignes 12-15, Listing 3.4).

Listing 3.4 Replacement de l'accesseur par un accès direct à la variable.

```
public void process(CtInvocation invok) {
  ...
  try{
    CtExpression igsGetter =
        getFactory().Code().createCodeSnippetExpression(getField);
    invok.replace(igsGetter);
  }
```

Dans le cas d'un mutateur, le processeur récupère la variable assignée dans le corps du mutateur (`this.myField = arg1`). Le processeur récupère la valeur passée en paramètre dans l'appel du mutateur et l'assigne directement à la variable précédemment récupérée (`mySetter(var) -> myField = var`) (Algorithme 1 Lignes 16-19).

Entrée : Fichier d'analyse PAPRIKA, Code source de l'application

Sortie : Code source de l'application sans la présence d'IGS

```
1  pour toutes les classes c dans le fichier d'analyse de PAPRIKA faire
2      accesseur, mutateur ← faux;
3      pour toutes les invocations de méthodes im dans c faire
4          si im est présent dans le fichier d'analyse de PAPRIKA alors
5              Lire la signature de la méthode im;
6              Vérifier si im est un accesseur ou un mutateur;
7              si le nombre de paramètres de la signature de im > 0 alors
8                  accesseur ← vrai
9              sinon
10                 mutateur ← vrai
11             fin
12             si accesseur alors
13                 Récupérer la dernière expression du corps de l'accesseur;
14                 Récupérer la variable que renvoie l'expression;
15                 Remplacer l'appel de l'accesseur par la variable;
16             si mutateur alors
17                 Récupérer la variable assignée dans le corps du mutateur;
18                 Récupérer la valeur passée en paramètre dans l'appel du mutateur;
19                 Remplacer l'appel du mutateur par l'assignation de la valeur du
                    paramètre à la variable;
20         fin
21     fin
22 fin
```

Algorithme 1 : Processus de correction de IGS.

3.1.2 Correction de HashMapUsage (HMU)

Il est important de d'abord comprendre la différence entre `HashMap` et un `ArrayMap` en terme de fonctionnalités avant d'expliquer le processus de correction de HMU. Les deux classes implémentent l'interface `Map<K,V>` et offrent les mêmes fonctionnalités (Android-Doc, 2017a; Android-Doc, 2017c). La différence entre les deux est que la classe `HashMap` implémente aussi l'interface `Cloneable` (Android-Doc, 2017b) qui permet aux instanciations de la classe `HashMap` d'avoir accès à la méthode `clone`. Cette méthode permet de créer une copie parfaite de l'instance de la classe.

Au niveau de la structure, une `HashMap` est un tableau d'objets `Map.Entry<K, V>` où les paires clefs valeurs sont non-primitives. Lors de l'ajout d'un élément dans une `HashMap`, le `hashCode` de la clé est calculé et la méthode *indexFor()* est appelée afin de déterminer l'index de l'instance `Entry` créée et de la stocker dans le *bucket* (les *buckets* sont utilisés par les `HashMap` afin de stocker les paires clefs valeurs). L'opération d'insertion (*put*) est de O(1). Il en va de même pour la méthode (*get*) (récupérer un élément) qui est elle aussi de O(1). Concernant la structure des `ArrayMap`, ils utilisent deux tableaux, le premier pour stoker des entiers qui correspondent aux `hashCode` des clefs et le deuxième pour stoker les objets (pas d'objet de type Entry pour les ArrayMap). Lors de l'ajout d'un élément dans un `ArrayMap`, la clef de l'objet est placée dans la prochaine position disponible du tableau d'objet. La valeurs quant elle est placée juste après la clef dans le tableau d'objet. L'opération d'insertion (*put*) est donc de O(n). Afin de récupérer un élément de la collection, une recherche dichotomique est faite sur le tableau contenant les `hashCode` afin de récupérer le `hash` et par la suite récupérer la paire clef valeur équivalente dans le tableau d'objet. L'opération de récupération des éléments (*get*) est donc de O(log(n)).

Le principale problème des `HashMap` est au niveaux de la mémoire (les buckets et les `Entry`) et des appels au GC pour l'allocation et la désallocation de mémoire. Ces opérations peuvent ralentir et/ou stopper le fonctionnement de l'application mobile. La manière d'en sont conçus les `ArrayMap` (l'utilisation de deux tableaux pour stocker les `hash` et les paires clef valeurs) pallie au problème de mémoire. Néanmoins, le principale problème du `ArrayMap` est que ces performances sont proportionnelles au nombre d'éléments, plus il contient d'éléments est plus le temps d'insertion et de récupération augment, O(n) et O(log(n)), ce qui réduit fortement les performances de l'application lors des itérations sur un grand nombre d'éléments. Il n'y a pas de documentation officiel sur le nombre optimal d'éléments que doit contenir un `ArrayMap`, mais la communautés web s'accorde sur un nombre inférieur à 1000 éléments.

Il est **très important** de noter que la détection effectuée par PAPRIKA est naïve et s'occupe juste de récupérer toutes les occurrences de `HashMap` présentes dans l'application. Aussi, il est impossible de détecter *statiquement* les HMU, car le nombre d'éléments que peut contenir une `Map` est variable selon l'exécution. Dans le cadre de notre étude, nous émettons l'hypothèse que les `Map` instanciées dans les applications testées sont relativement petites. Pour ce qui est donc de la correction des HMU, il s'agit de transformer toutes les occurrences de `HashMap` (type de variable, type de méthode, etc.) en `ArrayMap`. Par la suite, une analyse est effectuée afin de détecter les appels aux méthodes `clone`. Les méthodes `clone` sont remplacées par des nouvelles instanciations d'`ArrayMap` avec comme paramètre de construction l'instance de la `Map` qui appelle la méthode `clone` : `myMap.clone() -> new ArrayMap<>(myMap)`.

L'algorithme 2 représente le fonctionnement du processeur SPOON en charge de la correction des HMU. Le processeur effectue les corrections sur trois niveaux :

— Les instanciations de `HashMap` aux niveaux des attributs de classe ;

— Les types ;

— Les appels aux méthodes `clone` ;

L'implémentation de l'algorithme est disponible sur GitHub [2]. Le processeur prend comme entrée le fichier d'analyse de PAPRIKA (Figure 3.3) contenant les noms des classes correspondant à des HMU en plus du code source de l'application (Algorithme 2 Entrée).

full_name
updateDatabase#org.bottiger.podcast.parser.FeedUpdater
onCreate#org.bottiger.podcast.provider.PodcastProvider
<init>#org.bottiger.podcast.utils.DateUtils$1

Figure 3.3 Exemple de fichier d'analyse de Paprika pour la détection des HMU répertoriant les noms des classes et méthodes contenant des HMU.

Le processeur de correction parcourt toutes les classes de l'application répertoriées par PAPRIKA et vérifie les attributs de cette dernière (Algorithme 2 Ligne 1-2). Tout d'abord, le processeur récupère les instanciations des `HashMap` au niveau des attributs de la classe (`anyField = new HashMap<>(arg1)`) (Algorithme 2 Ligne 3-4, Listing 3.5).

2. `https://github.com/MehdiAit/spoon-processors/blob/master/src/main/java/io/paprika/spoon/SimpleHashMapProcessor.java`

Listing 3.5 Filtre Spoon pour la récupération des attributs instanciant des HashMap.

```
List<CtConstructorCall<?>> listConstrCall = ctClass.getElements(new
    AbstractFilter<CtConstructorCall<?>>(CtConstructorCall.class) {
            @Override
            public boolean matches(CtConstructorCall<?> element) {
                return element.getType().getSimpleName().equals("HashMap");
            }
});
```

Ensuite, il récupère les arguments de l'instanciation, transforme l'instanciation du `HashMap` en `ArrayMap`, et réutilise les précédents arguments (`anyField = new ArrayMap<>(arg1)`) (Algorithme 2 Lignes 5-7). Par la suite, le processeur vérifie le type de tous les éléments de la classe (Algorithme 2 Lignes 9) Le processeur récupère et remplace tous les éléments de types `HashMap` présents dans la classe en `ArrayMap` : type de retour des méthodes, paramètres des méthodes, variables locales des méthodes, variables locales des constructeurs, types de retour des méthodes des classes anonymes, etc (Algorithme 2 Lignes 10-15, Listing 3.6).

Listing 3.6 Transformations des HashMap en ArrayMap.

```
private void HashMapToArrayMap(CtTypeReference<?> ref){
      List<CtTypeReference<?>> types = ref.getActualTypeArguments();
      ref.replace(getFactory().Code().createCtTypeReference(ArrayMap.class));
      ref.setActualTypeArguments(types);
}
```

Le procédé de transformation est similaire à celui des transformations d'attributs. (`HashMap<K,V> var1 -> ArrayMap<K,V> var1`). Une fois que tous les `HashMap` ont été transformés en `ArrayMap`, le processeur va rechercher toutes les invocations

de la méthode `clone` dans la classe (`myArrayMap.clone()`). Le processeur sauvegarde d'abord le nom de variable du `ArrayMap` (`myArrayMap`) et va par la suite transformer l'invocation de la méthode en une nouvelle instanciation d'`ArrayMap` avec comme paramètre de construction la variable précédemment stockée (`new ArrayMap<>(myArrayMap)`) (Algorithme 2 Lignes 17-22).

Entrée : Fichier d'analyse PAPRIKA, Code source de l'application

Sortie : Code source de l'application sans la présence des HMU

```
1  pour toutes les classes c présentes dans le fichier d'analyse de PAPRIKA faire
2      Vérifier les attributs;
3      pour toutes les instanciations d'attributs ia dans c faire
4          si ia est une instanciation de HashMap alors
5              Récupérer les paramètres de l'instanciation;
6              Remplacer l'instance HashMap en instance d'ArrayMap;
7              Replacer les précédents arguments dans l'instanciation du ArrayMap;
8      fin
9      Vérifier les types;
10     pour tous les types des éléments te dans c faire
11         si te est de type HashMap alors
12             Récupérer les paramètres de l'élément de type HashMap;
13             Remplacer le type HashMap en type ArrayMap;
14             Replacer les précédents arguments dans l'élément de type ArrayMap;
15     fin
16     Vérifier les invocations aux méthodes clone;
17     pour toutes les invocations de méthodes im dans c faire
18         si im est une invocation à la méthode clone alors
19             Sauvegarder le nom de variable de l'instance du ArrayMap;
20             Supprimer l'invocation de méthode;
21             Créer une instanciation d'ArrayMap et mettre en paramètre l'instance
                du précédent ArrayMap;
22     fin
23 fin
```

Algorithme 2 : Processus de correction de HMU.

3.1.3 Correction de Member Ignoring Method (MIM)

Le processus de correction de MIM est relativement simple vu qu'il consiste à ajouter le mot clef **static** aux méthodes détectées par PAPRIKA. La seule vérification effectuée par le processeur est faite sur les redéfinitions de méthodes qui ne doivent pas être modifiées.

L'algorithme 3 représente la logique du processeur SPOON pour la correction de MIM. L'implémentation de l'algorithme est disponible sur GitHub [3]. Le processeur prend comme entrée le fichier d'analyse de PAPRIKA (Figure 3.4) contenant les noms des classes et méthodes correspondant à des MIM.

full_name
animationStartedLessThanOneSecondAgo#views.PlayPauseImageView
createHandlerMessage#DiscoveryFragment
CreateSyncAccount#service.syncadapter.CloudSyncUtils

Figure 3.4 Exemple de fichier d'analyse de Paprika pour la détection des MIM répertoriant les noms des classes et méthodes contenant des MIM.

Le processeur parcourt toutes les classes identifiées par PAPRIKA comme des MIM (Algorithme 3 Ligne 1). Pour chaque classe de l'application, le processeur prospecte les méthodes de cette dernière afin de voir si elles sont répertoriées dans le fichier d'analyse de PAPRIKA (Algorithme 3 Lignes 2-3). Si le processeur trouve une concordance, il va d'abord vérifier si la méthode trouvée n'est pas une redéfinition de méthode, car cette dernière ne doit pas être modifiée. Pour cela, le processeur examine l'annotation de la méthode (`@Override public void myMethodOverrided()`) (Algorithme 3 Lignes 4-6, Listing 3.7).

---

3. `https://github.com/MehdiAit/spoon-processors/blob/master/src/main/java/io/paprika/spoon/StaticProcessor.java`

Listing 3.7 Vérification de l'annotation de la méthode.

```
private boolean checkAnnotation(CtMethod candidate){
        for(CtAnnotation annotation : candidate.getAnnotations()){
            if(annotation.toString().trim().matches("(.*)@Override(.*)")){
                return false;
            }
}
```

Le processeur va à la fin ajouter le mot clef **static** à la signature de la méthode (`public static void myMethod()`) (Algorithme 3 Ligne 7).

Entrée : Fichier d'analyse PAPRIKA, Code source de l'application

Sortie : Code source de l'application sans la présence des MIM

```
1  pour toutes les classes c dans le fichier d'analyse de PAPRIKA faire
2      pour toutes les méthodes m dans c faire
3          si m dans le fichier d'analyse de PAPRIKA alors
4              Vérifier la redéfinition de la méthode;
5              si l'annotation de m = "Override" alors
6                  Arrêter et passer à la prochaine méthode;
7              Ajouter le mot clef static à la méthode m;
8          sinon
9              Passer à la prochaine méthode;
10         fin
11     fin
12 fin
```

Algorithme 3 : Processus de correction de MIM.

3.2 NAGA VIPER : Évaluation automatique de la consommation énergétique des applications Android

Afin d'évaluer l'impact de la correction des défauts de code présents dans une application, NAGA VIPER récupère l'APK original et celles générées par PAPRIKA afin de les comparer. NAGA VIPER est un outil d'analyse dynamique que nous avons développé afin d'analyser l'impact énergétique d'une application sur des scénarios utilisateurs. NAGA VIPER existe en deux versions : une version expérimentale écrite en PYTHON et une version robuste et stable en JAVA. Les deux versions sont disponibles sur GitHub[4][5]. NAGA VIPER est basé sur deux étapes qui lui permettent par la suite de délivrer un comparatif des applications ainsi qu'un pack ("kit") contenant la version de l'application la plus optimisée et le code source de cette dernière. La première phase a pour but de récolter les différentes métriques pour chacune des versions de l'application corrigées ainsi que la version originale. La deuxième phase va calculer la consommation moyenne de chaque version de l'application afin d'identifier la meilleure version de l'application et de renvoyer le kit. En plus de l'évaluation des applications, NAGA VIPER a la mission d'automatiser les tests de mesures. Il est en charge d'initialiser la connexion entre le périphérique mobile et le serveur, de configurer l'environnement d'essai, de démarrer et arrêter les tests etc. Les détails de l'automatisation sont abordés dans le chapitre 4.

3.2.1 Récupération et calcul des métriques énergétiques

La première étape de NAGA VIPER est de récolter les métriques de chaque version de l'application afin de pouvoir les utiliser par la suite dans la phase d'évaluation.

---

4. `https://github.com/SOMCA/hot-pepper-java`

5. `https://github.com/SOMCA/hot-pepper`

Pour cette phase, NAGA VIPER prend en entrée les différentes versions de l'application et un scénario utilisateur qui va être utilisé pour l'automatisation des tests. De plus, NAGA VIPER utilise le YOCTO-AMP, un instrument physique de mesure afin de récupérer l'intensité du courant lors de l'exécution de l'application (Figure 3.5). Le YOCTO-AMP offre une API simple d'utilisation pour pouvoir récupérer différentes informations à propos du périphérique mobile.

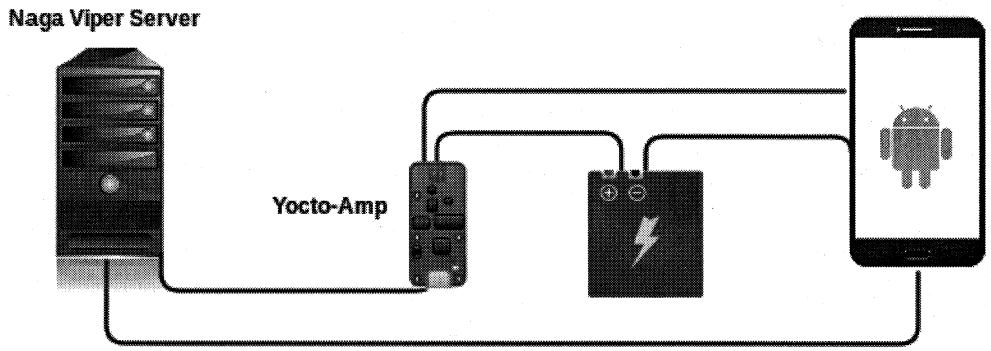

Figure 3.5 Branchement entre le Yocto-Amp, Naga Viper et le périphérique mobile.

NAGA VIPER est en charge de récupérer trois métriques lors de l'exécution d'une version de l'application : La moyenne de l'intensité du courant lors de la durée du scénario ($I_{moyenne}$), le temps d'exécution de chaque scénario (t) et le voltage de la batterie du périphérique mobile. La dernière métrique est constante, elle est donc configurée au niveau de NAGA VIPER avant le début des tests. Afin de réduire l'effet des facteurs externes (stabilité du débit de la connexion, activation du GC (Garbage Collector), ... etc), NAGA VIPER effectue un nombre configurable de fois l'exécution de l'application et la récolte des métriques. Les métriques récupérées sont envoyées au serveur interne de NAGA VIPER afin de calculer la moyenne de chaque test. NAGA VIPER effectue ces opérations pour chaque version de l'appli-

cation. Une fois toutes les versions évaluées, le serveur de NAGA VIPER envoie les métriques récoltées pour chaque version de l'application à la prochaine étape.

3.2.2 Évaluation de l'impact énergétique de la correction

Avant d'expliquer plus en détails cette étape, il est important d'abord d'expliquer la façon pour calculer la consommation énergétique d'une application Android.

Afin d'évaluer la consommation énergétique d'une application Android, nous nous sommes appuyés sur les recherches et le modèle énergétique élaboré par l'équipe de l'Université de Californie (Seo *et al.*, 2007; Seo *et al.*, 2008) pour évaluer la consommation d'énergie d'une application Java. Nous proposons d'évaluer la consommation d'énergie globale en mesurant l'intensité moyenne, la tension moyenne de l'appareil et le temps entre deux mesures de l'ampèremètre. L'intensité et la tension sont le flux et la force de l'électricité par l'intermédiaire d'une ligne électrique, respectivement. La consommation d'énergie est exprimée en JOULES. L'intensité de la batterie est limitée, et la tension est constante tout au long de l'utilisation du périphérique mobile. Par conséquent, nous estimons la consommation d'énergie d'une application Android comme suit :

$$E_{global} = \sum(V * \Delta t * I_{moyenne}) \tag{3.1}$$

Où E_{global} représente la consommation d'énergie générale de l'application Android (joules), V la tension actuelle de la batterie (volts), Δt le temps d'exécution de l'application étudiée (secondes) et $I_{moyenne}$ l'intensité moyenne (A) de l'application mobile et du système d'exploitation.

NAGA VIPER se sert des métriques récupérées de l'étape précédente afin de calculer la consommation énergétique de chaque version de l'application en utilisant l'équation 3.1. Par la suite, NAGA VIPER calcule le pourcentage d'énergie sauvé entre chaque version d'application corrigée et la version originale. NAGA VIPER

va par la suite récupérer la version de l'application avec le plus haut gain d'énergie. Cette dernière va faire partie de kit généré par NAGA VIPER en plus du code source associé à la version de l'application et un fichier contenant les modifications effectuées sur le code source.

3.3 Conclusion

Dans ce chapitre, nous avons exposé l'approche HOT PEPPER ainsi que son implémentation qui est basée sur PAPRIKA et NAGA VIPER. Nous avons étendu PAPRIKA avec SPOON afin de lui permettre d'effectuer la détection et la correction des défauts de code. Quant à NAGA VIPER, il a spécifiquement été développé dans le cadre de notre recherche. Ce dernier permet d'automatiser et de configurer les jeux de tests. De plus, il est en charge de calculer et d'évaluer l'impact énergétique de chaque application testée. Dans le prochain chapitre, nous utiliserons HOT PEPPER sur plusieurs applications afin de valider notre approche et de répondre à nos questions de recherche.

CHAPITRE IV

EXPÉRIMENTATIONS ET RÉSULTATS

Nous abordons dans ce chapitre les expérimentations effectuées afin de répondre à nos questions de recherches et de tester HOT PEPPER. Nous décrivons les applications utilisées et les spécifications de l'environnement dans lequel ces dernières ont été évaluées. Nous analysons les résultats obtenus lors des expérimentations et nous mettons en avant les obstacles à la validité concernant leur exactitude ainsi que la reproductibilité de notre protocole.

4.1 Questions de recherche

Nos expérimentations ont pour but de répondre aux questions de recherche (**QR**) suivantes :

QR 1 : *La correction de HashMap Usage (HMU) réduit-t-elle la consommation énergétique des applications Android ?*

QR 2 : *La correction de Internal Getter/Setter (IGS) réduit-t-elle la consommation énergétique des applications Android ?*

QR 3 : *La correction de Member Ignoring Method (MIM) réduit-t-elle la consommation énergétique des applications Android ?*

QR 4 : *La correction des trois types de défauts de code améliore-t-elle significativement la consommation énergétique comparativement à la correction d'un seul défaut de code ?*

4.2 Sujets

Les sujets de notre recherche représentent les trois défauts de code précédemment cités dans la Section 2.1 : HMU, IGS, et MIM.

4.3 Objets

Dans le but d'étudier l'impact énergétique des trois défauts de code, nous devions trouver des applications Android à code source ouverts/libres afin de permettre à PAPRIKA d'appliquer les correctifs nécessaires au niveau du code source. Les applications ont été choisies selon les critères suivants :

- La catégorie et l'utilité de l'application ;
- La popularité de l'application, en se basant sur le nombre de téléchargement et la notation ;
- La présence de défauts de code, au minimum deux des trois types de défauts de code précédemment cités doivent être présents dans l'application.

Nous avons lancé PAPRIKA sur plus de 1900 applications téléchargées à partir du magasin *F-droid*[1]. En s'appuyant sur les précédents critères, nous avons trouvé 34 applications classées en six catégories différentes : musique, lecture, productivité, utilité, sport et éducation. Pour chaque catégorie, nous avons choisi l'application contenant le plus de défauts de code. De plus, pour chaque application choisie nous avons vérifié si cette dernière peut être assemblée (*build*) et compilée. Sur les 34 applications ressorties, cinq d'entre elles, de différentes catégories, ont été choisies (Tableaux 4.1 et 4.3). Nous n'avons pas choisi d'application appartenant à la catégorie sport car ces applications nécessitaient l'utilisation d'une montre ou d'un bracelet connecté.

---

1. Magasin d'applications Android gratuites et en code source libre, `https://f-droid.org/`

Tableau 4.1 Liste des applications sélectionnées.

Application	Lien
Aizoban	`https://f-droid.org/repository/browse/?fdfilter=manga\\&fdid=com.jparkie.aizoban`
Calculator	`https://play.google.com/store/apps/details?id=com.android2.calculator3`
SoundWaves	`https://play.google.com/store/apps/details?id=org.bottiger.podcast`
Todo	`https://f-droid.org/repository/browse/?fdfilter=todo&fdid=com.xmission.trevin.android.todo`
Web Opac	`https://play.google.com/store/apps/details?id=de.geeksfactory.opacclient`

Tableau 4.2 Versions expérimentales des applications Android corrigées.

Version	Défaut de code corrigé
V_0	Version original
V_{HMU}	HashMap Usage (HMU)
V_{IGS}	Internal Getter/Setter (IGS)
V_{MIM}	Member Ignoring Method (MIM)
V_{ALL}	Tous les défauts de code sont corrigés

L'application *Aizoban* est un catalogue de mangas. Elle permet aux utilisateurs de lire, sauvegarder et télécharger leur manga préféré. La version testée de l'application est la version 1.2.5. L'application *Calculator* est la calculatrice par défaut de CyanogenMod[2]. La version testée de l'application est la version 5.1.1. L'application *SoundWaves Podcast* permet aux utilisateurs d'écouter et de télécharger des podcasts. La version testée de l'application est la version 0.130. *Todo* est une application qui permet aux utilisateurs de créer des listes de notes. La version testée de l'application est la version 1.0. L'application *Web Opac* offre aux utilisateurs la possibilité d'accéder à un catalogue de plus de 500 librairies à travers 30 pays. Elle propose aux utilisateurs la possibilité de chercher une librairie, avoir accès au catalogue de livres d'une librairie et réserver un livre. La version testée de l'application est la version 4.5.9. Le code source des applications citées est disponible sur GitHub et SourceForge.

2. `http://www.cyanogenmod.org`

Tableau 4.3 Liste des cinq applications Android (catégorie, paquet principal, nombre de classes, de méthodes et de défauts de code).

Applications	Catégorie	Paquet Principal	#Classe	#Méthode	#HMU	#IGS	#MIM	All
Aizoban	Lecture	com.jparkie.aizoban	524	2773	39	190	110	339
Calculator	Utilité	com.android2.calculator3	147	830	0	10	8	18
SoundWaves	Musique	org.bottiger.podcast	520	2,672	5	47	14	66
Todo	Productivité	com.xmission.trevin.android.todo	161	610	9	3	0	12
Web Opac	Éducation	de.geeksfactory.opacclient	367	2176	48	77	43	168

Une fois les applications choisies, nous avons corrigé ces dernières avec PAPRIKA afin d'obtenir différentes versions pour chaque application comme illustré dans le Tableau 4.2. V_0 est la version originale de l'application, V_{HMU}, V_{IGS} et V_{MIM} sont dérivées de V_0 en corrigeant respectivement HMU, IGS et MIM. Enfin, V_{ALL} est la version où tous les défauts de code Android sont corrigés.

4.4 Environnement

Un environnement spécifique a été assemblé afin d'effectuer nos expérimentations dans les meilleures conditions possibles. Le périphérique mobile utilisé est un Google Nexus 4 avec comme système d'exploitation CyanogenMod 11 qui est basé sur la version 4.4.4 (KitKat) d'Android. Le Google Nexus 4 dispose d'un processeur Snapdragon S4 quad-core de 1,5 GHz, un écran HD IPS Plus de 4,7 pouces (1280 x 768 pixels), 2 Go de RAM et une batterie de 2100 mAh. La différence entre la version Android 4.4.4 et CynogenMod 11 se situe au niveau des options développeurs proposées par CynogenMod 11, comme par exemple le mode super utilisateur qui est facilement activable et désactivable (option indispensable pour nos expérimentations).

De plus, CynogenMod 11 embarque, de base, moins d'applications et de services que les systèmes Android standards, ce qui nous permet d'avoir moins d'interférences lors du processus de mesure. Comme vu dans la Figure 3.5, le YOCTO-AMP est directement branché à la batterie du périphérique mobile et afin de stabiliser ce branchement, nous avons utilisé un harnais, illustré dans la Figure 4.1, spécialement conçu pour le périphérique mobile (une des raisons de notre choix pour le Nexus 4) afin de fixer le branchement entre la batterie, le téléphone et le YOCTO-AMP.

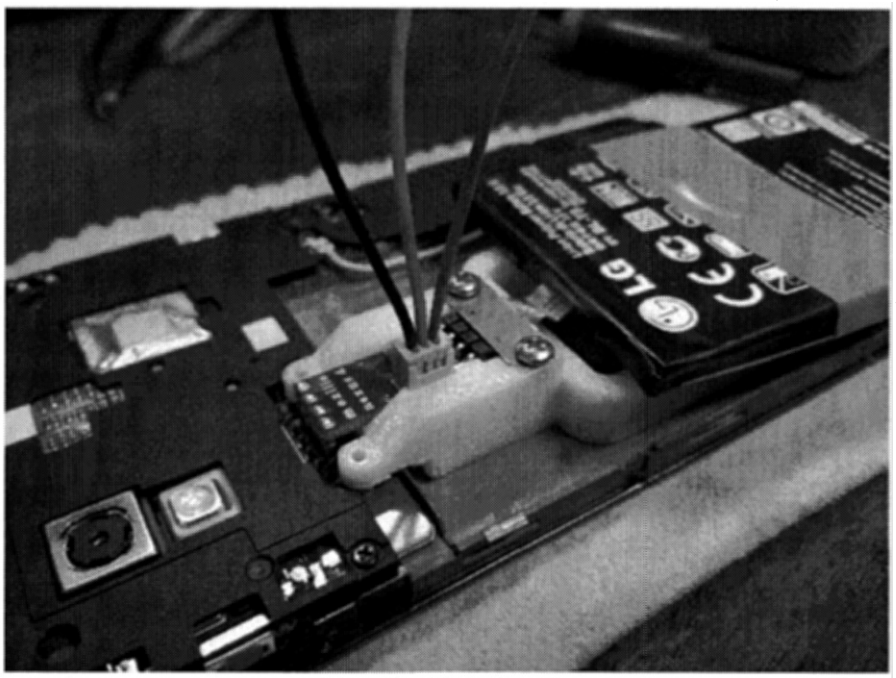

Figure 4.1 Montage de la batterie du périphérique mobile avec le harnais qui est relié au Yocto-Amp.

4.5 Minimisation des facteurs externes

Afin de réduire au maximum les interférences lors des mesures et de pouvoir reproduire nos expérimentations, nous avons configuré un environnement spécifique avec un ensemble de règles à suivre. Tous d'abord, pour établir la communication entre NAGA VIPER et le périphérique mobile durant les expérimentations nous utilisons le port USB de ce dernier pour profiter de l'ANDROID DEBUG BRIDGE (ADB). L'ADB est un outil de ligne de commande qui permet de communiquer avec un périphérique (un émulateur ou un appareil Android connecté) et de faciliter une variété d'actions avec ce dernier, telles que l'installation et le débogage des applications. Pour ne pas biaiser nos mesures, NAGA VIPER désactive la charge du téléphone avant de commencer les expérimentations et la réactive à la fin.

Ci-dessous, les règles à suivre afin de pouvoir reproduire les expérimentations et obtenir les mêmes résultats :

- pas d'utilisation des données SIM (3G, 4G LTE, etc...) ;
- pas d'utilisation de fond d'écran dynamique/interactif ;
- les essais doivent être répétés le même nombre de fois qu'expliqués dans la sous Section 4.6.2 ;
- le niveau de la batterie doit être en-dessous de 100% avant le début de chaque expérimentation afin de permettre à NAGA VIPER de désactiver la charge du périphérique mobile ;
- le niveau de la batterie du périphérique mobile ne doit pas être en-dessous de 50% durant les expérimentations, car nous avons noté des effets de bord au niveau de la consommation énergétique une fois ce seuil dépassé ;
- il est recommandé d'éteindre le périphérique mobile après chaque jeu d'essai afin de le laisser refroidir ;
- le Wi-Fi doit être désactivé si l'application testée ne nécessite pas d'accès Internet ;
- si le Wi-fi est requis, il est impératif d'assurer une connexion stable ;
- le Bluetooth doit être désactivé ;
- la luminosité et le volume du périphérique mobile doivent être au minimum.

Aussi, il faut noter que pour les deux derniers points NAGA VIPER est en mesure de les configurer lui-même via l'utilisation de l'ADB. De plus, avant le début de chaque expérimentation nous devons nous assurer qu'aucune application ne tourne en fond afin de ne pas biaiser les mesures (cette vérification est effectuer manuellement).

4.6 Procédure

Dans cette section, nous allons expliquer les procédures réalisées pour exécuter nos tests expérimentaux et récupérer nos résultats. Globalement, trois processus ont été appliqués sur les objets expérimentaux : (1) détection et correction des défauts de code, (2) réalisation et exécution des scénarios utilisateurs, et (3) récupération et calcul des métriques énergétiques.

4.6.1 Détection et correction des défauts de code

La première étape consiste donc à corriger les défauts de code présents dans la version originale de chaque application (V_0). D'abord, pour une application choisie, nous récupérons l'APK de la version originale (V_0) et son code source afin de lancer la phase de détection des défauts de code de PAPRIKA. Après la phase de détection, PAPRIKA effectue la correction des défauts de code trouvés. Pour chaque type de défaut de code, PAPRIKA corrige ce dernier et génère la version corrigée (V_{HMU}, V_{IGS}, ou V_{MIM}). À la fin, PAPRIKA lance la correction des trois défauts de code simultanément afin de tous les corriger et de générer la version corrigée adéquate (V_{ALL}).

Aussi, nous avons comparé la correction manuelle de l'application SoundWaves avec la correction automatique afin de valider le processus de correction automatique de PAPRIKA. Nous avons comparé les versions corrigées automatiquement et manuellement sur trois critères :

- La consommation énergétique : la consommation des deux versions sont similaires ;
- Le comportement de l'application : lors de l'exécution des scénarios utilisateurs, aucune interruption de l'application ou comportement anormale n'a été rapporté sur la version corrigée automatiquement. De plus, la durée

d'exécution des scénarios des deux versions est identique ;

— Le code source des applications : les codes sources des deux versions de l'application sont parfaitement identiques.

4.6.2 Scénarios

Une fois la correction effectuée, NAGA VIPER prend en charge la partie exécution des tests et évaluation de la consommation énergétique. Pour les tests, NAGA VIPER exécute un nombre de fois le scénario donné pour chaque application. La réalisation des scénarios est faite avec CALABASH[3], une plate-forme de test pour application mobile écrit en Ruby. Pour chaque application, nous créons un scénario Calabash qui parcourt les fonctionnalités de cette dernière. Chaque scénario Calabash est basé sur un nombre x d'étapes et y de temps d'attente. Les étapes représentent les actions de l'utilisateur : clique, défilement, entrée de texte, etc. (Figure 4.2)

```
Then(/^I touch on settings$/) do
        touch("AppCompatTextView marked:'Settings'")
end
```

Figure 4.2 Exemple d'étapes Calabash définissant une action dans l'application SoundWaves.

Les temps d'attente, en second, sont utilisés pour permettre d'assurer que les vues de l'application soient complètement chargées avant d'effectuer une action afin de ne pas faire planter l'application et donc le scénario de test. Comme rapporté dans les Tableaux 4.3 et 4.5, les scénarios couvrent un nombre représentatif de types de défauts de code et d'occurrences.

---

3. http://calaba.sh

En plus des étapes et des temps d'attente, chaque scénario Calabash doit d'abord initialiser la communication avec le serveur NAGA VIPER afin de pouvoir lui indiquer le début des tests pour commencer les mesures et la fin du scénario pour arrêter les mesures et commencer le calcul des métriques (Figure 4.3).

```
$sock = TCPSocket.new("127.0.0.1", 3000)

Then(/^I begin$/) do
        $sock.send("STARTED", 0)
end

Then(/^I end run$/) do
        $sock.send("END", 0)
        $sock.flush
        $sock.close
end
```

Figure 4.3 Connexion avec le serveur Naga Viper et envoie de message pour le début et la fin des tests.

Les scénarios pour les cinq applications sont disponible sur notre Github [4].

4.6.3 Récupération et calcul des métriques

Chaque scénario a été exécuté 20 fois par version d'application. Le YOCTO-AMP est configuré pour récupérer 75 mesures par seconde durant le temps d'exécution d'un scénario. Les mesures récupérées sont associées à un timestamp qui permet à NAGA VIPER de calculer le temps d'exécution du scénario. Une fois les mesures d'intensité et le temps d'exécution du scénario récupéré, NAGA VIPER calcule la consommation énergétique de chaque exécution comme expliqué dans la Sec-

4. https://github.com/SOMCA/hot-pepper-data/tree/master/scenarios

tion 3.2. Par la suite, la consommation énergétique de chaque exécution est utilisée pour calculer la consommation globale de l'application. De plus, nous exécutons les tests statistiques de Cliff's δ afin de confirmer si la différence énergétique entre chaque version d'application est significative ou pas.

4.7 Variables

Variables indépendantes : Lors de nos expérimentations, les variables indépendantes correspondent au nombre de défauts de code corrigés (HMU, IGS et MIM) dans chaque version d'application.
Variables dépendantes : Dans nos expérimentations, les variables dépendantes correspondent aux métriques énergétiques précédemment citées dans la Section 3.2 : le temps d'exécution de l'application, le voltage du périphérique mobile et la moyenne d'intensité.

4.8 Méthodes d'analyse

Afin d'analyser nos résultats, nous exécutons un test non paramétrique afin d'évaluer l'exactitude de nos résultats, le test de taille d'effet Cliff's δ (Romano *et al.*, 2006). Ce dernier permet de comparer deux ensembles de données, A et B par exemple, et de savoir s'ils sont disjoints ou pas. Le test renvoie une valeur entre $+1$ (si toutes les valeurs de A sont inférieures à celles de B) et -1 (si toutes les valeurs de A sont supérieures à celles de B). Le test renvoie 0 si les deux ensembles sont identiques (Cliff, 1993).

Les tests sont exécutés avec un niveau de confiance de 99%. Une faible taille d'effet ($\delta > 0.147$) représente une légère différence très difficilement visible, une moyenne taille d'effet ($\delta > 0.330$) représente une différence un peu plus notable et visible. Une forte taille d'effet ($\delta > 0.474$) représente une différence significative et très facilement observable.

4.9 Explication des résultats obtenus

Nous présentons et expliquons dans cette section les résultats obtenus lors de nos expérimentations effectuées sur les cinq applications Android, précédemment citées dans la Section 4.3, qui nous ont permis de répondre à nos questions de recherche.

Tous d'abord, en analysant les Tableaux 4.3 et 4.5, on remarque que le nombre d'appels de certains défauts de code est faible comparativement à leur présence dans les applications. Cela résulte du fait que les scénarios réalisés ne parcourent pas toutes les actions faisant appel aux défauts de code. En effet, les scénarios sont destinés à reproduire des actions courantes d'utilisateurs et non pas à stresser les applications avec des appels constants aux défauts de code.

4.9.1 QR1 : La correction de HashMap Usage (HMU) réduit-elle la consommation énergétique des applications Android ?

Pour les résultats concernant les HMU ils sont en faveur des versions corrigées des applications. Comme le montre le Tableau 4.4, la consommation énergétique globale des applications corrigées, trois sur quatre, baisse significativement de 2.00%, 2.40% et 2.06% pour les applications respectives de *Aizoban*, *Todo* et *Web Opac*. De plus, pour l'application *Aizoban*, V_{HMU} est la version qui consomme le moins comparativement aux autres versions corrigées de l'application. Une très légère baisse de 0.38% est aussi notée sur l'application *SoundWaves* mais elle n'est pas très significative. Aussi, les résultats des tests de Cliff's δ illustrés dans le Tableau 4.6 démontrent une forte taille d'effet supérieur à 0.474 pour les applications *Aizoban* (0.58) et *Todo* (0.66) ainsi qu'une moyenne taille d'effet de 0.43 qui peut facilement être observée sur l'application *Web Opac*. En ce qui concerne

l'application *SoundWaves*, la taille d'effet récupérée est très faible, 0.08, et donc non significative. La faible différence de consommation d'énergie pour l'application *SoundWaves* peut-être sujette à question, tandis que cette dernière fait plus d'appels aux méthodes contenant des HMU que les autres applications. En effet, il est possible que la taille des `ArrayMap` ait fortement augmentée, pouvant dépasser une centaine d'éléments, durant l'exécution de l'application ce qui rend l'utilisation des `ArrayMap` moins efficace.

Néanmoins, en vu des résultats reportés, nous considérons que la correction de HMU peut réduire la consommation énergétique d'une application Android.

4.9.2 QR2 : La correction de Internal Getter/Setter (IGS) réduit-elle la consommation énergétique des applications Android ?

En observant le Tableau 4.4 on remarque une baisse de la consommation d'énergie sur toutes les applications corrigées ne contenant pas de IGS. Une forte baisse de 2.04% et 2.08% pour les applications respectives *Todo* et *Web Opac* est constatée. Alors qu'une moins forte baisse de 1.09% et 1.43% est remarquée respectivement sur les applications *Aizoban* et *SoundWaves*. Toutefois, V_{IGS} reste la meilleure version des applications corrigées pour l'application *SoundWaves*. Une très faible baisse de 0.18% est remarquée pour l'application *Calculator* alors que cette dernière fait plus de 6100 invocations d'IGS durant l'exécution du scénario. En analysant le Tableau 4.6, les résultats des tests de Cliff's δ appuient les observations faites par rapport au Tableau 4.4. Une forte taille d'effet est reportée sur l'application *Todo* alors qu'une moyenne taille d'effet, qui se rapproche d'une taille d'effet significative, est observée sur les applications *Web Opac* et *Aizoban*. Pour l'application *SoundWaves*, une faible taille d'effet de 0.26% est notée alors que les résultats pour l'application *Calculator* sont non significatifs avec une taille d'effet de 0.11%. La consommation d'énergie de toutes les applications corrigées s'est

améliorée. Néanmoins des questions, sans réponses pour le moment, se posent sur les résultats peu significatifs obtenus sur l'application *Calculator* alors que cette dernière invoque souvent le défaut de code. Aucune hypothèse ou supposition n'a pu être émise à ce niveau de la recherche, d'autre tests et explorations sont nécessaires afin d'approfondir le sujet.

Cependant, d'après les résultats obtenus, nous considérons que la correction d'IGS peut réduire la consommation énergétique d'une application Android.

Tableau 4.4 La différence de consommation d'énergie de chaque version par rapport à la version V_0.

Applications	V_{HMU}	V_{IGS}	V_{MIM}	V_{ALL}
Aizoban	**-2.00%**	-1.09%	+0.08%	-1.38%
Calculator	-	-0.18%	-0.45%	**-1.69%**
SoundWaves	-0.38%	**-1.43%**	+0.29%	-1.29%
Todo	-2,40%	-2,04%	-	**-4,83%**
Web Opac	-2.06%	-2.08%	**-3.86%**	-3.50%

4.9.3 QR3 : La correction de Member Ignoring Method (MIM) réduit-t-elle la consommation énergétique des applications Android ?

En examinant le Tableau 4.4 on s'aperçoit que les résultats pour MIM sont assez partagés. D'une part, nous avons une baisse de la consommation d'énergie de 0.45% et 3.86% respectivement pour les applications de *Calculator* et *Web Opac* (à noter que V_{MIM} est la version qui consomme le moins parmi les versions corrigées de l'application *Web Opac*). D'autre part, nous avons une augmentation de la consommation d'énergie de 0.08% et 0.29% respectivement pour les applications de *Aizoban* et *SoundWaves*. Ces résultats sont aussi supportés par le Tableau 4.6. On peut facilement observer la forte et faible taille d'effet, concernant la baisse de

la consommation énergétique respectives sur les applications *Web Opac* et *Calculator*. Une autre faible taille d'effet est remarquée sur l'application *SoundWaves* mais qui est liée cette fois à l'augmentation de la consommation d'énergie. Pour l'application *Aizoban*, la taille d'effet est très faible et donc non significative. L'augmentation de la consommation au niveau des applications *Aizoban* et *SoundWaves* n'est pas encore totalement expliquée. Pour le moment aucune hypothèse n'a pu être émise concernant cette augmentation. Une piste de solution, afin d'approfondir l'étude de la hausse rencontrée, aurait été d'isoler les appels des MIM (corrigés et non corrigés), par exemple en élaborant des scénarios qui sollicitent uniquement ces derniers, afin d'étudier de manière plus granulaire la correction du défaut de code.

Au vu des résultats, nous considérons que la correction de MIM peut réduire la consommation énergétique d'une application Android, néanmoins des investigations additionnelles doivent être menées sur ce défaut de code concernant l'augmentation rencontrée sur certaines applications.

Tableau 4.5 Consommation énergétique globale des différentes versions des cinq applications (# étapes, temps d'attente seconds, # défauts de code appelés, consommation énergétique globale en Joules).

Applications	#Étape	Attente	$\#HMU_c$	$\#IGS_c$	$\#MIM_c$	V_0	V_{HMU}	V_{IGS}	V_{MIM}	V_{ALL}
Aizoban	169	17s	10	1300	0	0.0424	**0.0415**	0.0419	0.0425	0.0418
Calculator	325	0s	0	6122	1350	0.0300	-	0.0300	0.0299	**0.0295**
SoundWaves	172	53s	420	8053	6560	0.0859	0.0856	**0.0847**	0.0867	0.0848
Todo	248	5s	40	20	0	0.0369	0.0361	0.0362	-	**0.0352**
Web Opac	136	79s	6	133	40	0.0392	0.0384	0.0384	**0.0377**	0.0378

4.9.4 QR4 : La correction des trois types de défauts de code améliore-t-elle significativement la consommation énergétique comparativement à la correction d'un seul défaut de code ?

Les résultats du Tableau 4.4 montrent que la consommation de toutes les applications baisse après la correction de tous les défauts de code. Pour les applications *Calculator* et *Todo*, V_{ALL} est la meilleure version avec une baisse respective de la consommation d'énergie de 1.69% et 4.83%. Néanmoins, pour les autres applications, on remarque que la différence entre la meilleure version et V_{ALL} est minime. Dès lors, la dernière colonne du Tableau 4.6 compare la meilleure version de chaque application avec V_{ALL}. Avec des résultats de -0.01, 0 et 0.11 respectivement pour les applications *Aizoban*, *SoundWaves* et *Web Opac*, nous ne remarquons aucune différence significative entre la meilleure version et V_{ALL}. Nous pouvons donc considérer V_{ALL} comme étant statistiquement la meilleure application.

La correction de tous les défauts de code améliore significativement la consommation d'énergie. Bien que V_{ALL} ne soit pas toujours la meilleure version, cette dernière reste une excellente version qui est très proche de la version optimale.

Tableau 4.6 Résultats du test de Cliff's δ sur chaque version par rapport à la version V_0, en plus de la version V_M qui représente la meilleure version de l'ensemble et qui est comparée a V_{ALL}.

Apps	V_0 , V_{HMU}	V_0 , V_{IGS}	V_0 , V_{MIM}	V_0 , V_{ALL}	V_M , V_{ALL}
Aizoban	0.58	0.46	-0.06	0.58	-0.01
Calculator	—	0.11	0.18	0.42	0
SoundWaves	0.08	0.26	-0.24	0.26	0
Todo	0.66	0.62	—	0.92	0
Web Opac	0.43	0.46	0.69	0.60	-0.11

4.10 Menaces à la validité

Dans cette section, nous mettons en avant les menaces à la validité concernant nos expérimentations en se basant sur le guide proposé par Wohlin (Wohlin *et al.*, 2012).

Validité conceptuelle : La menace à la validité conceptuelle représente la relation entre l'aspect théorique et les observations effectuées. Dans nos expérimentations, cette menace peut être liée aux erreurs de mesure. Pour cette raison, nous avons exécuté 20 fois chaque scénario de chaque version d'applications et utilisé les valeurs moyennes plutôt que les valeurs instantanées. Nous avons aussi défini un nombre de règles à respecter dans la Section 4.5 afin de limiter l'impact de facteur externe durant le processus de mesure. Aussi, il est possible que les mesures de notre ampèremètre (YOCTO-AMP) ne soient pas exactes, car ce sont des mesures physiques. Néanmoins, afin de valider les mesures de notre YOCTO-AMP, nous avons utilisé un deuxième ampèremètre et il s'est avéré que les mesures des deux périphériques été équivalentes durant les tests effectués avec HOT-PEPPER.

Validité interne : La menace à la validité interne représente la relation entre les changements effectués et les résultats obtenus. Les résultats ont été obtenus uniquement suite à la correction des défauts de code, il nous est donc possible de faire la relation entre le processus de correction et les interprétations effectuées sur nos résultats. De plus, nous avons manuellement vérifié le code source des applications après le processus de correction de PAPRIKA afin de s'assurer que les modifications ont été faites uniquement sur les parties contenant des défauts de code.

Validité externe : La menace à la validité externe représente la possibilité de généraliser nos résultats. Les résultats obtenus durant nos expérimentations sont dépendants de l'environnement configuré (Section 4.4), des applications utili-

sées (Section 4.3), des scénarios réalisés (Section 4.6), du nombre de défauts de code présents dans l'application et le nombre de leur occurrence durant un scénario. Nous ne pouvons donc pas généraliser les résultats obtenus à travers nos expérimentations. Toutefois, nous avons varié les applications utilisées, les défauts de code détectés ainsi que les scénarios utilisateurs pour pouvoir appuyer nos résultats, qui ont prouvés que dans certains contextes la correction des défauts de code impact positivement la consommation d'énergie des applications Android. Il est donc possible que cet effet soit applicable sur un bon nombre d'applications. Néanmoins, l'investigation d'autre défauts de code ainsi qu'un registre d'applications plus vaste sont nécessaires afin de pouvoir approfondir le sujet de l'impact des défauts de code sur les applications Android.

Validité à la fiabilité : La menace à la fiabilité représente la possibilité de pouvoir répliquer nos expérimentations. Nous avons essayé de détailler le plus possible notre étude et de fournir un maximum d'informations afin que cette dernière soit reproductible. Les applications, les scénarios et les résultats sont disponibles sur Github[5].

Validité des conclusions : La menace à la validité des conclusions appuie le fait que les conclusions déduites durant l'étude sont justes. Durant notre étude, nous avons respecté les suppositions faites par nos tests statistiques. Néanmoins, les données traitées sont continues et donc l'hypothèse que les mesures suivent la lois normale est validée. D'autre tests statistiques plus appropriés au contexte auraient pu donc être utilisés, comme le test de *Student* qui peut être utilisé pour déterminer si deux ensembles sont significativement différents l'un de l'autre (dans notre cas les mesures avant et après correction).

---

5. `https://github.com/SOMCA/hot-pepper-data`

4.11 Conclusion

Lors de ce chapitre, nous avons présenté notre protocole expérimental ainsi que les résultats obtenus. Nous avons pu affirmer que la correction d'un des trois défauts de code peut réduire la consommation d'énergie d'une application Android. Aussi, nos résultats ont démontré que la correction de tous les défauts de code, simultanément, améliore la consommation d'énergie d'une application et est dans le pire des cas est équivalant à la correction d'un défaut de code ayant un impact significatif. Nous conseillons donc de corriger tous les défauts de code présents dans une application via l'utilisation de HOT-PEPPER et en particulier PAPRIKA. Dans le prochain chapitre, nous évaluons HOT-PEPPER dans un cas industriel afin d'évaluer la consommation énergétique d'une application Android au cours de son développement.

CONCLUSION

La consommation d'énergie est devenue un facteur de qualité clef et critique dans le développement de logiciel particulièrement dans le développement des applications mobiles en raison de la limite d'énergie fournie par les batteries des périphériques mobiles. Autre contrainte du développement mobile, le temps de conception, très court, consacré au développement des applications, ce qui conduit facilement à l'apparition de défauts de code. De nombreuses études ont été réalisées afin de démontrer l'impact des défauts de code sur la performance des applications Android, ce qui a conduit à l'avènement d'outils permettant aux développeurs de détecter certains défauts de code durant leur développement. Néanmoins, peu de recherches ont été faites sur l'impact des défauts de code sur la consommation énergétique des applications Android. De plus, à date, il n'existe pas d'outils ou de plate-formes permettant d'évaluer l'impact des défauts de code sur la consommation d'une application Android et de les corriger automatiquement.

Notre approche HOT-PEPPER est basée sur deux outils et permet d'évaluer l'impact énergétique dans les applications Android de trois défauts de code liés à la performance : *InternalGetter/Setter* (IGS), *HashMapUsage* (HMU) et *Member Ignoring Method* (MIM). Le premier outil se nome PAPRIKA, il a été conçu initialement pour détecter les défauts de code Android (Hecht *et al.*, 2015b) et nous l'avons étendu afin de lui permettre aussi d'effectuer la correction des trois défauts de code précédemment cités. Le deuxième outil se nomme NAGA VIPER, nous l'avons développé afin d'évaluer l'impact des défauts de code avant et après leur corrections dans les applications Android. De plus, NAGA VIPER permet l'automatisation de tous les tests d'évaluation via l'utilisation de scénarios

utilisateurs. Afin de valider notre approche, nous l'avons expérimentée sur cinq applications Android. Nous avons démontré à travers les résultats obtenus lors de nos expérimentations que la correction des défauts de code Android impactée positivement la consommation d'énergie d'une application sur des scénarios utilisateurs. Néanmoins, comme expliqué dans la Section 4.10, les résultats sont dépendant de l'environnement utilisé, nous ne les généralisons donc pas. De plus, la version Android étudiée et la 4.4, il est donc probable que les défauts de code présentés lors de cette étude soient obsolètes sur les versions plus récentes.

Nous avons aussi pu déployer HOT-PEPPER dans un contexte industriel via un partenariat[6] entre l'UQAM et l'entreprise *Savoir-faire Linux* (*SFL*)[7] afin d'évaluer la consommation énergétique d'une de leurs application, *Ring*[8]. Aussi, les travaux présentés dans ce mémoire ont fait l'objet cette année d'une publication dans la conférence *SANER 2017* (*International Conference on **S**oftware **A**nalysis, **E**volution, and **R**eengineering*) (Carette *et al.*, 2017).

Actuellement, plusieurs perspectives d'évolution sont projetées par notre équipe de recherche. La première perspective est d'investiguer plus de défauts de code afin d'augmenter la portée de HOT-PEPPER. Par la suite, il est question de travailler sur une méthode de récolte des métriques moins intrusive afin de pouvoir répliquer facilement le protocole expérimental. Aussi, nous planifions de déployer HOT-PEPPER sur le *Cloud* afin de le rendre facilement accessible aux développeurs. Finalement, nous espérons que cette recherche permettra aux développeurs d'éviter l'implémentation de défauts de code dans leurs applications et ainsi améliorer la qualité de cette dernière.

---

6. `https://blog.savoirfairelinux.com/fr-ca/s/eco-conception/`

7. `https://www.savoirfairelinux.com//`

8. `https://ring.cx/`

BIBLIOGRAPHIE

Android (2017a). Performance tips, avoid internal getters/setters. `https://stuff.mit.edu/afs/sipb/project/android/docs/training/articles/perf-tips.html`.

Android, D. (2017b). Android performance tips. `http://developer.android.com/training/articles/perf-tips.html`. [Online ; accessed January-2016].

Android, D. (2017c). Avoid using floating-point. `https://developer.android.com/training/articles/perf-tips.html\#AvoidFloat`.

Android, D. (2017d). Improve your code with lint. `https://developer.android.com/studio/write/lint.html`. [Online ; accessed 2017].

Android, D. (2017e). Use native methods carefully. `https://developer.android.com/training/articles/perf-tips.html#NativeMethods`.

Android-Doc (2017a). Arraymap. `https://developer.android.com/reference/android/support/v4/util/ArrayMap.html`.

Android-Doc (2017b). Cloneable. `https://developer.android.com/reference/java/lang/Cloneable.html`.

Android-Doc (2017c). Hashmap. `https://developer.android.com/reference/java/util/HashMap.html`.

Balasubramanian, N., Balasubramanian, A. et Venkataramani, A. (2009). Energy consumption in mobile phones : a measurement study and implications for network applications. Dans *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, 280–293.

Banerjee, A., Guo, H.-F. et Roychoudhury, A. (2016). Debugging energy-efficiency related field failures in mobile apps. MobileSoft. To appear.

Bartel, A., Klein, J., Le Traon, Y. et Monperrus, M. (2012). Dexpler : Converting android dalvik bytecode to jimple for static analysis with soot. Dans *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, 27–38., New York, NY, USA.

ACM. http://dx.doi.org/10.1145/2259051.2259056. Récupéré de http://doi.acm.org/10.1145/2259051.2259056

Ben, G. E. (2011). What optimizations can i expect from dalvik and the android toolchain? http://stackoverflow.com/a/4930538. [Online; accessed January-2016].

Binkley, D. (2007). Source code analysis : A road map. Dans *2007 Future of Software Engineering*, FOSE '07, 104–119., Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/FOSE.2007.27. Récupéré de http://dx.doi.org/10.1109/FOSE.2007.27

Borba, P. (2002). Integrating code generation and refactoring.

Bruneton, E., Lenglet, R. et Coupaye, T. (2002). Asm : a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, *30*, 19.

Brylski, M. (2013). Android smells catalogue. http://www.modelrefactoring.org/smell_catalog. [Online; accessed January-2016].

Carette, A., Ait Younes, M. A., Hecht, G., Moha, N. et Rouvoy, R. (2017). Investigating the Energy Impact of Android Smells. Dans A. Marcus et G. Bavota (dir.). *24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER)*, Proceedings of the 24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER), p. 10., Klagenfurt, Austria. IEEE. Récupéré de https://hal.inria.fr/hal-01403485

Cliff, N. (1993). Dominance statistics : Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, *114*(3), 494.

Dong, M. et Zhong, L. (2012). Chameleon : A color-adaptive web browser for mobile oled displays. *IEEE Transactions on Mobile Computing*, *11*(5), 724–738. http://dx.doi.org/10.1109/TMC.2012.40

Duribreux, J., Rouvoy, R. et Monperrus, M. (2014). *An Energy-efficient Location Provider for Daily Trips.* Rapport technique.

Fokaefs, M., Tsantalis, N., Stroulia, E. et Chatzigeorgiou, A. (2011). Jdeodorant : Identification and application of extract class refactorings. Dans *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, 1037–1039., New York, NY, USA. ACM. http://dx.doi.org/10.1145/1985793.1985989. Récupéré de http://doi.acm.org/10.1145/1985793.1985989

Fowler, M. (1999). *Refactoring : improving the design of existing code.* Pearson

Education India.

Fowler, M., Beck, K., Brant, J., Opdyke, W. et Roberts, D. (1999). Refactoring : improving the design of existing code. 1999. *ISBN : 0-201-48567-2.*

Girish, S., Ganesh, S. et Tushar, S. (2015). Refactoring for software design smells : Managing technical debt. *SIGSOFT Softw. Eng. Notes*, *40*(6), 36–36. Reviewer-Tracz, Will, `http://dx.doi.org/10.1145/2830719.2830739`. Récupéré de `http://doi.acm.org/10.1145/2830719.2830739`

Gottschalk, M., Josefiok, M., Jelschen, J. et Winter, A. (2012). Removing energy code smells with reengineering services. Dans U. Goltz, M. A. Magnor, H.-J. Appelrath, H. K. Matthies, W.-T. Balke, et L. C. Wolf (dir.). *GI-Jahrestagung*, volume 208, 441–455.

Gui, J., Li, D. et Wan, Mianand Halfond, W. G. (2016). Lightweight measurement and estimation of mobile ad energy consumption. Dans *Proceedings of the 5th International Workshop on Green and Sustainable Software - GREENS.* To appear.

Gui, J., Mcilroy, S., Nagappan, M. et Halfond, W. G. J. (2015). Truth in advertising : the hidden cost of mobile ads for software developers. Dans *Proceedings of the 37th International Conference on Software Engineering - ICSE*, 100–110. IEEE.

Guo, C., Zhang, J., Yan, J., Zhang, Z. et Zhang, Y. (2013). Characterizing and detecting resource leaks in android applications. Dans *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 389 - 398. IEEE.

Haase, C. (2015). Developing for android, ii the rules : Memory. `https://medium.com/google-developers/developing-for-android-ii-bb9a51f8c8b9`. [Online ; accessed January-2016].

Hao, S., Li, D., Halfond, W. G. J. et Govindan, R. (2012). Estimating android applications' cpu energy usage via bytecode profiling. Dans *2012 First International Workshop on Green and Sustainable Software (GREENS)*, 1–7. `http://dx.doi.org/10.1109/GREENS.2012.6224263`

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

Hecht, G., Moha, N. et Rouvoy, R. (2016b). An empirical study of the performance impacts of android code smells. *1*(1). Récupéré de `https://hal.inria.fr/`

hal-01276904

Hecht, G., Omar, B., Rouvoy, R., Moha, N. et Duchien, L. (2015a). Tracking the software quality of android applications along their evolution. p. 12. Récupéré de https://hal.inria.fr/hal-01178734

Hecht, G., Rouvoy, R., Moha, N. et Duchien, L. (2015b). Detecting antipatterns in android apps. Dans *Research Report - INRIA, Lille*.

Hindle, A., Wilson, A., Rasmussen, K., Barlow, E. J., Campbell, J. C. et Romansky, S. (2014). Greenminer : A hardware based mining software repositories software energy consumption framework. 12–21. http://dx.doi.org/10.1145/2597073.2597097. Récupéré de http://doi.acm.org/10.1145/2597073.2597097

Hovemeyer, D. et Pugh, W. (2004). Finding bugs is easy. *SIGPLAN Not.*, *39*(12), 92–106. http://dx.doi.org/10.1145/1052883.1052895. Récupéré de http://doi.acm.org/10.1145/1052883.1052895

IDC (2015). Smartphone os market share 2015. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

Inc, M. S. (2017). Monsoon power monitor : a robust power measurement solution. https://www.msoon.com/LabEquipment/PowerMonitor/.

Jabbarvand, R., Sadeghi, A., Garcia, J., Malek, S. et Ammann, P. (2015). Ecodroid : an approach for energy-based ranking of android apps. Dans *Proceedings of the Fourth International Workshop on Green and Sustainable Software*, 8–14. IEEE.

Kniesel, G. et Koch, H. (2004). Static composition of refactorings. *Science of Computer Programming*, *52*(1–3), 9 – 51. Special Issue on Program Transformation, http://dx.doi.org/http://dx.doi.org/10.1016/j.scico.2004.03.002. Récupéré de http://www.sciencedirect.com/science/article/pii/S0167642304000462

Kundu, T. K. et Paul, K. (2011). Improving android performance and energy efficiency. 256–261. http://dx.doi.org/10.1109/VLSID.2011.63

Lee, S. et Jeon, J. W. (2010). Evaluating performance of android platform using native c for embedded systems. Dans *ICCAS 2010*, 1160–1163. http://dx.doi.org/10.1109/ICCAS.2010.5669738

Li, D. et Halfond, W. G. (2014). An investigation into energy-saving programming practices for android smartphone app development. Dans *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 46–53. ACM.

Li, D., Hao, S., Gui, J. et Halfond, W. G. (2014). An Empirical Study of the Energy Consumption of Android Applications. *2014 IEEE International Conference on Software Maintenance and Evolution*, 121–130. http://dx.doi.org/10.1109/ICSME.2014.34. Récupéré de http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6976078

Li, D., Hao, S., J., H. W. G. et Govindan, R. (2013). Calculating source line level energy information for android applications. Dans *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA*, 78–89.

Li, D., Lyu, Y., Gui, J. et Halfond, W. G. (2016). Automated energy optimization of http requests for mobile applications. Dans *Proceedings of the 38th International Conference on Software Engineering - ICSE*. To appear.

Lin, C. M., Lin, J. H., Dow, C. R. et Wen, C. M. (2011). Benchmark dalvik and native code for android system. Dans *2011 Second International Conference on Innovations in Bio-inspired Computing and Applications*, 320–323. http://dx.doi.org/10.1109/IBICA.2011.85

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M. et Poshyvanyk, D. (2014). Mining energy-greedy api usage patterns in android apps : An empirical study. 2–11. http://dx.doi.org/10.1145/2597073.2597085. Récupéré de http://doi.acm.org/10.1145/2597073.2597085

Linares-Vasquez, M., Bavota, G., Bernal-Cardenas, C., Oliveto, R., Di Penta, M. et Poshyvanyk, D. (2014). Mining energy-greedy api usage patterns in android apps : an empirical study. Dans *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR*, 2–11.

Liu, Y., Xu, C. et Cheung, S.-C. (2014). Characterizing and detecting performance bugs for smartphone applications. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, 1013–1024. http://dx.doi.org/10.1145/2568225.2568229. Récupéré de http://dl.acm.org/citation.cfm?doid=2568225.2568229

Loveman, D. B. (1977). Program improvement by source-to-source transformation. *J. ACM*, *24*(1), 121–145. http://dx.doi.org/10.1145/321992.322000. Récupéré de http://doi.acm.org/10.1145/321992.322000

Mäntylä, M. V. et Lassenius, C. (2006). Subjective evaluation of software evolvability using code smells : An empirical study. *Empirical Software Engineering*, *11*(3), 395–431. http://dx.doi.org/10.1007/s10664-006-9002-8. Récupéré de http://dx.doi.org/10.1007/s10664-006-9002-8

Marcu, M. et Tudor, D. (2011). Energy consumption model for mobile wireless communication. Dans *Proceedings of the 9th ACM international symposium on Mobility management and wireless access*, 191–194. ACM.

Marinescu, C., Marinescu, R., Mihancea, P. F. et Wettel, R. (2005). iplasma : An integrated platform for quality assessment of object-oriented design. 77–80.

Mittal, R., Kansal, A. et Chandra, R. (2012). Empowering developers to estimate app energy consumption. Dans *ACM Mobicom*. ACM.

Moha, N. (2007). Detection and correction of design defects in object-oriented designs. 949–950.

Neo4J (2017). Neo4j : The world's leading graph database. `https://neo4j.com/`.

Noureddine, A., Rouvoy, R. et Seinturier, L. (2013). A review of energy measurement approaches. *SIGOPS Oper. Syst. Rev.*, *47*(3), 42–49. `http://dx.doi.org/10.1145/2553070.2553077`. Récupéré de `http://doi.acm.org/10.1145/2553070.2553077`

Noureddine, A., Rouvoy, R. et Seinturier, L. (2014). Unit Testing of Energy Consumption of Software Libraries. 1200–1205. `http://dx.doi.org/10.1145/2554850.2554932`. Récupéré de `https://hal.inria.fr/hal-00912613`

Pathak, A., Hu, Y. C., Zhang, M., Bahl, P. et Wang, Y.-M. (2011). Fine-grained power modeling for smartphones using system call tracing. Dans *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, 153–168., New York, NY, USA. ACM. `http://dx.doi.org/10.1145/1966445.1966460`. Récupéré de `http://doi.acm.org/10.1145/1966445.1966460`

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C. et Seinturier, L. (2015). Spoon : A library for implementing analyses and transformations of java source code. *Software : Practice and Experience*, p. na. `http://dx.doi.org/10.1002/spe.2346`. Récupéré de `https://hal.archives-ouvertes.fr/hal-01078532/document`

Pérez-Castillo, R. et Piattini, M. (2014). Analysing the harmfull effect of god class refactoring on power consumption. Dans *IEEE Software*, volume 31, 48–54. IEEE.

Query, C. (2017). Cypher : The neo4j's open graph query language. `https://neo4j.com/developer/cypher/`.

Romano, J., Kromrey, J. D., Coraggio, J. et Skowronek, J. (2006). Appropriate statistics for ordinal level data : Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. Dans *annual*

meeting of the Florida Association of Institutional Research, 1–33.

Samir, H., Zachary, K., Munawar, H., Mohammed, S., Bram, A. et Abram, H. (2015). Energy profiles of java collections classes. *ICSE 2016.*

Seo, C., Malek, S. et Medvidovic, N. (2007). An energy consumption framework for distributed java-based systems. Dans *Proceedings of the 22nd IEEE/ACM International Conference on Automated software engineering (ASE '07)*, 421–424. ACM.

Seo, C., Malek, S. et Medvidovic, N. (2008). Estimating the energy consumption in pervasive java-based systems. Dans *Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, 243–247. IEEE.

Statista (2015). Global smartphone sales by operating system from 2009 to 2015. `https://www.statista.com/statistics/263445/global-smartphone-sales-by-operating-system-since-2009/`. [Online ; accessed October-2016].

Tatsubori, M., Chiba, S., Killijian, M.-O. et Itano, K. (1999). Openjava : A class-based macro system for java. 117–133.

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*, p. 13. IBM Press.

Wan, M., Jin, Y., Li, D. et Halfond, W. G. J. (2015). Detecting display energy hotspots in android apps. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings.* `http://dx.doi.org/10.1109/ICST.2015.7102585`

Wasserman, A. I. (2010). Software Engineering Issues for Mobile Application Development. *ACM Transactions on Information Systems*, 1–4. `http://dx.doi.org/10.1145/1882362.1882443`. Récupéré de `http://works.bepress.com/cgi/viewcontent.cgi?article=1003{\&}context=tony{\_}wasserman`

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. et Wesslén, A. (2012). *Experimentation in software engineering.* Springer.

Zhang, L. (2013). *Power, Performance Modeling and Optimization for Mobile System and Applications.* (Thèse de doctorat). Northwestern University.

Zhang, L., Tiwana, B., Dick, R. P. et Qian, Z. (2010). Accurate online power estimation and automatic battery behavior based power model generation

for smartphones. Dans *Hardware/Software Codesign and System Synthesis – CODES+ISSS*, 105–114. IEEE.

Zhang, Y., Huang, G., Liu, X., Zhang, W., Mei, H. et Yang, S. (2012). Refactoring Android Java Code for On-Demand Computation Offloading. Dans *OOPSLA 2012*, Tucson Arizona, United States. Récupéré de `https://hal.inria.fr/hal-00751656`