Towards Improving the Quality of Mobile Apps by Leveraging Crowdsourced Feedback

Maria Gomez

Towards Improving the Quality of Mobile Apps by Leveraging Crowdsourced Feedback

Maria Gomez

Supervisors: Prof. Romain Rouvoy and Prof. Lionel Seinturier

Inria Lille Nord Europe
University of Lille

This dissertation is submitted for the degree of
Doctor of Philosophy in Computer Science

Thesis Committee:
Prof. Xavier Blanc, Dr. Jacques Klein and Prof. Michele Lanza (referees)
Dr. Alessandra Gorla and Prof. Luigi Lancieri (examiners)

University of Lille 2nd December 2016

"Difficult things take a long time;
impossible things, a little longer"
—Dr. André A. Jackson

A mi tío Miguel, por ser mi referente.
A mis padres y mi hermano, por ser mi fuente de motivación.
A Antonio, por su paciencia y apoyo infinitos.

Acknowledgements


failure
crowd
enable
study
fixed
appear
buckets
thrown
running
assist
Information
automatic
faults
crowds
Checkers
want
recreate
API
market
User
tests
real
tool
log
presence
java
Risk
means
run
priori
historic
bug
incorrect
executions
runs
static
logs
Vivo
services
reduce
five
block
errors
fix
frame
logs
help
hence
execute
wild
bugs
bucket
isolate
topics
usage
prevent
test
Patch
Dune
solutions
field
major
umObj
build
Study
Apps
device
nodes
Vitro
versions
feature
arises
service

Abstract

The development of mobile applications (*apps*) is exploding, largely due to the widespread use of mobile devices. App stores (*e.g., Google Play*) are the common distribution channels for apps. The excessive competition in current app markets is forcing app vendors to release high-quality apps. In fact, previous studies have repeatedly demonstrated that app users are intolerant to quality issues (e.g., crashes, unresponsive apps). Users who encounter issues frequently uninstall apps and move to alternative apps. Hence, quickly detecting and preventing issues is crucial for staying competitive in the market.

Although developers use emulators and test apps before deployment, many bugs emerge in the wild. Developing apps which run without errors along time remains a primary concern for app developers. The big challenge is that the environment is out of the app developers' control. More specifically, the mobile ecosystem faces rapid platform evolution (OSs, APIs), high device fragmentation, and high diversity of execution contexts (networks, locations, sensors).

This thesis introduces a new generation of app stores—App Store 2.0, which exploit crowdsourced information about apps, devices and users to increase the overall quality of the delivered mobile apps. App stores already have access to these different types of crowds. We claim that app stores can exploit the *wisdom of the crowd* to distill actionable insights from the feedback returned by the crowds. These actionable insights assist app developers to deal with potential errors and threats that affect their apps prior to publication or even when the apps are in the hands of end-users.

We sketch a prototype of the envisioned app store for Android apps. We validate the proposed solution with real bugs and apps. Our results have proven the applicability and feasibility of our approach. These new app stores have the potential to reduce human effort and save precious time for developers, which is a decisive factor for the success of mobile apps in current app markets.

Résumé

Le développement des applications mobiles (*apps*) est en pleine explosion, en grande partie en raison de l'utilisation généralisée des appareils mobiles. Les magasins d'applications (*e.g.,* Google Play) sont les canaux de distribution courants pour les apps. La concurrence excessive sur les marchés d'applications actuels oblige les fournisseurs d'applications à diffuser des applications de haute qualité. En fait, des études antérieures ont démontré que les utilisateurs d'applications sont intolérants à des problèmes de qualité (e.g., des arrêts inopinés, des applications qui ne répondent pas). Les utilisateurs qui rencontrent des problèmes désinstallent fréquemment les applications et se dirigent vers des applications concurrentes. Par conséquent, détecter et prévenir rapidement des problèmes dans les applications est crucial pour rester compétitif sur le marché.

Même si les développeurs utilisent des émulateurs et testent les applications avant le déploiement, de nombreux bugs peuvent encore apparaître dans la nature. Développer des applications qui fonctionnent sans erreur à travers le temps reste donc une préoccupation majeure pour les développeurs. Le grand défi qui demeure est que l'environnement reste hors du contrôle des développeurs d'applications. Plus précisément, l'écosystème mobile est confronté à une rapide évolution des plateformes mobiles, une forte fragmentation des équipements, et une grande diversité des contextes d'exécution (les réseaux, les localisations, les capteurs).

Cette thèse présente donc une nouvelle génération de magasins d'applications mobiles—App Store 2.0—qui exploite des données collectées sur les applications, les appareils et les utilisateurs afin d'augmenter la qualité globale des applications mobiles publiées en ligne. Les magasins d'applications ont déjà accès à différents types de communautés (de smartphones, d'utilisateurs, et d'applications). Nous affirmons que cette nouvelle génération de magasins d'applications peut exploiter l'intelligence collective pour obtenir des indications pratiques à partir des données retournées par les utilisateurs. Ces indications concrètes aident les développeurs d'applications à traiter les erreurs et

les menaces potentielles qui affectent leurs applications avant la publication ou même lorsque les applications sont dans les mains des utilisateurs finaux.

Nous avons conçu un prototype du magasin d'applications envisagé pour les applications Android. Nous validons la solution proposée avec des bugs et des applications réels. Nos résultats ont prouvé l'applicabilité et la faisabilité de notre approche. Ces nouveaux magasins d'applications ont le potentiel de réduire l'effort humain et de gagner du temps précieux pour les développeurs, ce qui est un facteur déterminant pour le succès des applications mobiles sur les marchés d'applications actuels.

Table of contents

List of figures

List of tables

Part I

Preface

The first part of this manuscript introduces the motivation, scope and goals of this thesis.

Chapter 1

Introduction

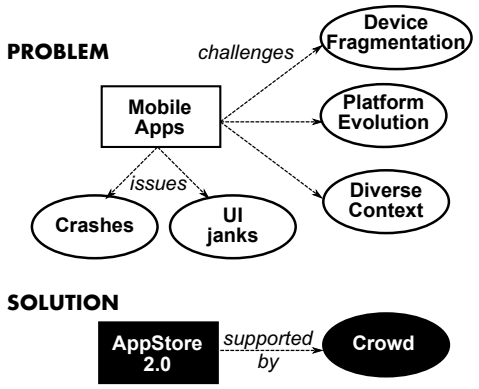


Fig. 1.1 Chapter 1 at a glance

The widespread use of mobile devices (such as smartphones and tablets) has accelerated the development of *mobile applications*—broadly called *'apps'*. These apps are commonly distributed through centralized platform-specific *app stores*, such as Google Play (Android), Apple Store (iOS), Windows Store (Windows Phone), etc. The *Google Play Store* (the official market for Android apps) publishes 2.2 million apps [147] and registered 65 billion app downloads during the last year [16]. Revenue-wise, app developers earned more than 41 billion dollars in 2015, while projections estimate 101 billion dollar to be collected by 2020 [145].

While the popularity of mobile apps keeps on increasing, the quality of these apps varies widely. Unfortunately, end-users frequently experience bugs and crashes with some apps installed on their devices; as reported by user feedback in the form of reviews [99, 109] and ratings [119]. Given the increasing competition in the mobile app ecosystem, improving the experience of users is a major goal for app vendors.

This thesis introduces a new generation of app stores—App Store 2.0, which exploits crowdsourced information about apps, devices and users to increase the overall quality of the delivered mobile apps. Since app stores already have access to these different types of crowds, the key insight is to make them exploit the *wisdom of the crowds*

to generate actionable insights for the stakeholders of the mobile ecosystem (app developers, app users, and app store moderators). These new app stores have the potential to save precious time for developers, which is a crucial factor for the success of apps in current app markets.

The rest of the chapter is organized as follows. Section 1.1 describes the motivation of this research. Section 1.2 discuss the problem that this thesis aims to tackle. Section 1.3 presents the main goals of this thesis. Section 1.4 introduces our proposed solution. Section 1.5 details the publications derived from this research. Section 1.6 describes the international visits performed during this thesis. Finally, Section 1.7 summarizes the structure of this document.

1.1 Motivation

The severe competition in the app market challenges app vendors to release high-quality apps for keeping popularity among users. Previous studies showed that app users who encounter issues (*e.g.*, crashes) are likely to stop using the app and even to look for alternatives [88].

The quality of apps depends on both functional and non-functional requirements (*e.g.* absence of failures and performance). In particular, app crashes and unresponsive apps heavily disrupt the users' experience, as highlighted by user feedbacks [99]. When issues surface, app developers need to quickly detect, replicate, and fix issues. Otherwise, they risk to loose customers and be forced out of the market.

Currently, a wide range of testing tools for mobile apps are available [89, 112, 46, rob]. Although developers use emulators and test apps extensively prior to release, many bugs still emerge once the apps are deployed on users' mobile devices [29]. As an illustration, let us consider a developer who has developed and published an app in a store. Eventually, Android releases a new OS version, for example prompting devices to update from the `6.0 Marshmallow` version to the latest `7.0 Nougat` version. Immediately after the OS update, the app starts crashing on users' devices and some users complain and report negative reviews on the store [119]. This crash is originated by an OS update which is out of control of the app developer. The main challenge faced by app developers is that they do not own the ecosystem, as it is the case of web applications [27].

In particular, guaranteeing the proper functioning of the deployed apps along time is challenging for the following reasons:

- **Rapid platform evolution.** The Android operating system is extremely fragmented (*i.e.*, SDKs, APIs). For example, new Android versions are released frequently [156]; during 2015 two Android versions were released (Lollipop 5.1 and Marshmallow 6.0), and in 2016 a new version (Nougat 7.0) has been released.
- **High device fragmentation.** Despite the platform fragmentation, there is a high device fragmentation. There are over 24,000 distinct devices and more that 1,200 different device brands [126]. Thus, it is challenging to test an app in all available devices.
- **Heterogeneous execution contexts.** There is a high diversity of configuration settings and conditions of execution (*e.g.*, sensors, networks, locations, etc.). Hence, it is practically impossible to simulate all possible runtime environments *in vitro,* i.e., before release.

1.2 Problem Statement

Beyond their initial publication, maintaining apps remains one of the most complex tasks in mobile developments. More specifically, app developers face the following challenges:

- First, app developers need to detect when users face issues. This is typically done by manual investigations of user reviews and/or bug reports (commonly collected via reporting libraries).
- Second, the conditions triggering the defects must be isolated to reproduce the issues. Given the high combinatorial explosion of device characteristics and operating conditions, this quickly becomes an arduous task.
- Lastly, the defects should be fixed and a new app version must be released to the store. However, as long as the developer does not upload a fixed version, the store continues spreading the defective app across users and devices. Since the release process of a new app version can be long [117], developers risk to loose customers who move to the competitors' apps. In addition, apps lose a certain percentage of users when they are asked to re-download apps, because many users never do it [27].

The work presented in this thesis aims to alleviate the challenges presented above. More specifically, this thesis aims to answer the following four research questions:

- **Research question 1**: *Is it possible to predict occurrences of mobile app bugs from the feedback returned by the crowd?*
- **Research question 2**: *Is it possible to faithfully reproduce* in vitro *mobile app bugs experienced by users in the wild?*
- **Research question 3**: *Is it possible to prevent recurrences of mobile app bugs* in vivo*?*
- **Research question 4**: *Is it possible to exploit the diversity of the crowd to improve user experience with mobile apps?*

These research questions are explored in next sections.

1.3 Thesis Goals

The main goal of this thesis is to drastically improve the quality of mobile apps. We pursue to complement existing testing solutions with novel mechanisms to monitor and debug apps after their deployment in the wild. We focus on two types of bugs: crashes and UI janks.

Regarding *research question 1*, this thesis aims to study mechanisms to anticipate the emergence of bugs before deploying apps to users. This is especially important when considering that negative reviews in early releases of an app make it almost impossible to recover afterwards [119]. These mechanisms learn from observations of existing buggy apps executed by the crowd.

Regarding *research question 2*, this research aims to provide mechanisms to automate the reproduction of bugs faced by users in the wild. These mechanisms systematically analyze crash reports collected from a crowd of devices and users to automatically synthesize execution traces which reproduce wild failures. The goal is to minimize developer effort and automate the reproduction step.

Regarding *research question 3*, this thesis intends to provide mechanisms to prevent the exhibition of further instances of a previously observed failure. These mechanisms aim to mitigate the raise of unhanded exceptions. For such purpose, crowd feedback will be exploited to isolate the conditions triggering failures and the defective functionalities

of apps. The prevention actions need to be transparent to users. The final goal is to minimize the time that users are exposed to failures and the number of affected users.

Finally, regarding *research question 4*, this thesis aims to provide an approach to automatically detect and isolate UI performance defects by exploiting the diversity of devices and operating contexts.

1.4 Proposed Solution

This thesis proposes leveraging the ***wisdom of the crowd*** to improve the quality of mobile apps. Our key insight is that app stores should leverage the different types of crowds to which they have access—*crowd of apps*, *crowd of devices*, and *crowd of users*. We therefore claim that app stores can leverage the value of this crowdsourced information to deliver new services, supporting the development of mobile apps in several ways.

This thesis introduces App Store 2.0, which enhances standard app stores by exploiting the wisdom of the crowd. The crowd contributes to the development of a new generation of app stores, which include the capability to assist developers to deal with potential errors and threats that affect their apps prior to publication or even when the apps are in the hands of end-users. The App Store 2.0 has two main missions: 1. to reduce developer effort to maintain mobile apps, and 2. to enhance the experience and loyalty of users with apps. The App Store 2.0 relies on self-healing principles and aims to support and automate the mobile app maintenance process.

The App Store 2.0 contributes to the delivery of actionable insights to the stakeholders of a mobile app ecosystem. These feedbacks span over *risk reports* to support the decision process of app store moderators, *reproducible scenarios* to support the reproduction task of app developers, *performance reports* for app developers, and *app patches* for app users. The modular software architecture of App Store 2.0 paves the way for the deployment of additional modules that an app store moderator and app developers can decide to activate.

Finally, we sketch a prototype of the envisioned App Store 2.0 for Android apps. We perform a series of empirical studies to validate the feasibility of our proposal. Our research shows that crowdsourcing can overcome the challenges identified above and improve the overall quality of mobile apps.

1.5 Publications

The contributions derived from this research have been published on international peer-review conferences. In this section, we detail the publications and awards resulted from this thesis. The publications are ordered by year of publication.

1.5.1 Publication Details

- **Maria Gomez**, Bram Adams, Walid Maalej, Martin Monperrus, Romain Rouvoy. App Store 2.0*: From Crowd Information to Actionable Feedback in Mobile App Ecosystems.* IEEE Software - Special Issue in Crowdsourcing for Software Engineering, 2017 *(to appear).*
- ***Maria Gomez**, Romain Rouvoy, Bram Adams, Lionel Seinturier.* Mining Test Repositories for Automatic Detection of UI Performance Regressions in Android Apps*. 13th International Conference on Mining Software Repositories (MSR'16) [76] (acceptance rate: 36/103, 27%).*
- ***Maria Gomez**, Romain Rouvoy, Bram Adams, Lionel Seinturier.* Reproducing Context-sensitive Crashes of Mobile Apps using Crowdsourced Monitoring*. 3rd IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'16) [77] (acceptance rate: 22,6%).*
- ***Maria Gomez**, Matias Martinez, Martin Monperrus, Romain Rouvoy.* When App Stores Listen to the Crowd to Fight Bugs in the Wild*. 37th International Conference on Software Engineering (ICSE'15), track on New Ideas and Emerging Results (NIER) [75] (acceptance rate: 25/135, 18%).*
- ***Maria Gomez**, Romain Rouvoy, Martin Monperrus, Lionel Seinturier.* A Recommender System of Buggy App Checkers for App Store Moderators*. 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'15) [78] (acceptance rate: 7/42, 16.6%).*

In addition, one journal article is under submission:

- **Maria Gomez**, Romain Rouvoy, Lionel Seinturier, Walid Maalej. Crowdseer*: Preventing App Crashes by Learning from Users.* IEEE Transactions on Software Engineering. *(article to be submitted).*

1.5.2 Awards

This thesis has received two awards.

- **1st Place Winner ACM Student Research Competition Award**. **Maria Gomez**. *Debugging of Mobile Apps in the Wild Guided by the Wisdom of the Crowd*. 2nd ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'15 - SRC) [74]. May 2015. This work was invited to participate in the grand final competition.
- **Best Poster Award**. PhD Welcome Seminar organized by Pôle de Recherche et d'Enseignement Supériur (PRES) - Université Lille Nord de France (ULNF). Nov. 2013.

1.6 International Research Visits

During this PhD, two research visits have been performed. Table 1.1 summarizes the details of these visits.

Dr. Bram Adams and Prof. Walid Maalej have a deep expertise in software engineering and mobile computing. In particular, Dr. Bram Adams is an expert on software release engineering. While, Prof. Walid Maalej is an expert on app store analysis and crowdsourcing. These expertises excellently complements this PhD topic. As a result, these collaborations have influenced and tremendously enriched this thesis.

1.7 Thesis Outline

This dissertation is organized in 5 parts ,which contain 9 chapters and 3 appendixes as shown in Figure 1.2.

Part I: Preface. This manuscript starts by presenting the motivation of this research and the challenges that this thesis intends to tackle. First, Chapter 2 reviews the state of the art on the major research areas that are related to this research. Second, Chapter 3 provides and overview of the App Store 2.0 proposed in this thesis.

Part II: Contributions. The second part describes in detail the contributions of this thesis. In Chapter 4, we present the monitoring block to gather different types

Table 1.1 International Research Visits

Research visit	
Duration	July 2015 – October 2015 (3 months)
Host	Dr. Bram Adams
Destination	Lab on Maintenance, Construction and Intelligence of Software (MCIS) Polytechnique Montreal Montreal, Canada
Results	During this visit, we performed an empirical study regarding crash reproduction with a group of students simulating a diverse crowd. In addition, we explored performance bugs in mobile apps and propose an approach to automatically detect UI performance defects in heterogeneous contexts.
Research visit	
Duration	January 2016 – April 2016 (4 months)
Host	Prof. Walid Maalej
Destination	Mobile Services and Software Engineering (MAST) Hamburg University Hamburg, Germany
Results	As a result of this visit, we developed and approach to automatically prevent crashes in mobile apps by deactivating UI features on the fly. This approach was empirically evaluated with a large group of mobile users and app developers.

of feedbacks from the crowd in app stores. Afterwards, Chapter 5 and Chapter 6 describe the approaches which constitute the App Store 2.0 before and after releasing apps, respectively. These approaches generate four types of actionable insights from crowdsourced information.

Part III: Evaluations. In this part, we present four empirical studies to validate the feasibility of the proposal. In particular, Chapter 7 reports on empirical evaluations to validate the approaches contained in the *in vitro* module. Whereas Chapter 8 validates the approaches contained in the *in vivo* module.

Part IV: Final Remarks. Finally, Chapter 9 presents the final conclusions and speculates future research directions to continue this research.

Part V: Appendixes. Additionally, we attach three appendixes. Appendix **??** describes the tools implemented to support the approaches presented in this thesis. Appendix A shows the design of the survey used to evaluate our proposal with users.

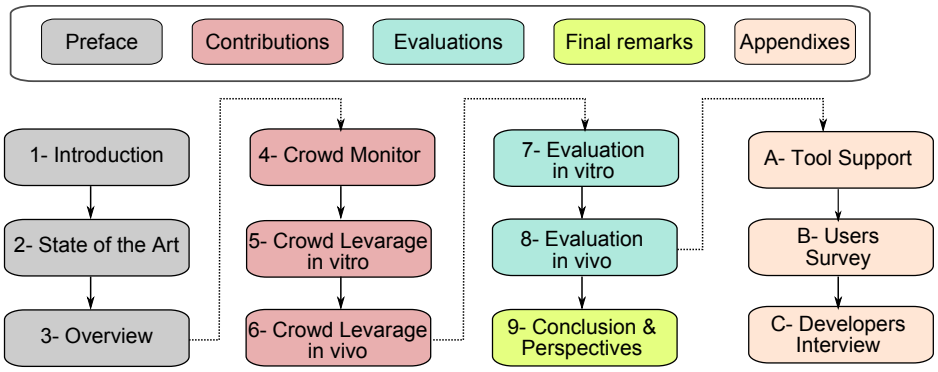


Fig. 1.2 Theis outline

Appendix B illustrates the design of the interview performed with developers as part of the evaluation of this thesis.

Chapter 2

State of the Art

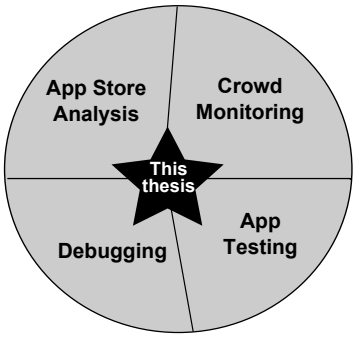


Fig. 2.1 Chapter 2 at a glance

The development of mobile apps is experiencing an unprecedented popularity. This chapter reviews the state of the art in the major research areas that are closely related to this thesis. This chapter aims to answer the following research questions:

q1: *Which approaches have been proposed?*

q2: *How to organize the proposed approaches?*

q3: *What are the limitations of current approaches?*

These research questions are answered along the chapter. We have identified four main research areas: *app store analysis*, *debugging*, *crowd monitoring*, and *mobile app testing*. To classify existing approaches, we consider different sub-areas within each area. Figure 2.2 shows a graphical presentation of the main research areas and subareas of this thesis.

This thesis fits in the intersection of the four bodies of knowledge. In the following sections, we analyze and discuss the most relevant approaches in these areas and their subareas.

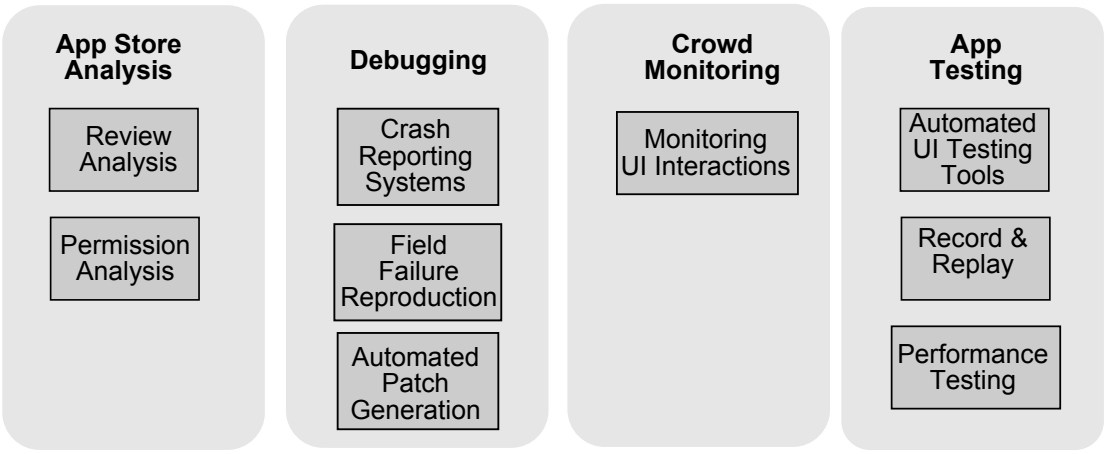


Fig. 2.2 Research Areas and subareas related to this thesis

The rest of the chapter is organized as follows. Sections 2.1 to 2.4 summarize related work to this research and outlines the deficiencies of existing approaches. Finally, Section 2.5 presents the conclusions.

2.1 App Store Analysis

The *app store analysis* area studies information regarding mobile apps mined from app stores. William et. al. [154] have identified seven key sub-fields of app store analysis: *API analysis*, *feature analysis*, *release engineering*, *review analysis*, *security analysis*, *store ecosystem comparison*, and *size and effort prediction*. We focus on analyzing the related work in the two sub-fields related to this thesis: **reviews** and **API analysis**. In particular, in the *API analysis* sub-field we focus on approaches which concern the study of the permissions of APIs.

2.1.1 Review Analysis

In this section, we discuss recent approaches that analyze app reviews posted by users on the Google Play Store. Table 2.2 summarizes prior work in the *Review Analysis* sub-area.

Ha *et al.* [82] manually analysed user reviews available on Google Play Store to understand what users write about apps. Performing this task manually becomes infeasible due to the large amount of available reviews.

Table 2.1 Prior work in the *Review Analysis* in Google Play Store

Approach	**#Apps**	**#Reviews**
Ha *et al.* [82]	59	556
Chen *et al.* [122]	4	169,097
Fu *et al.* [70]	171,493	13,286,706
Iacob and Harrison [93]	161	3,279
Maalej *et al.* [108]	80	146,057

Chen *et al.* [122] present AR-Miner, a tool for mining reviews from Google Play Store and extract user feedbacks. First, they filter reviews that contain useful information for developers to improve their apps. Then, they prioritize the most informative reviews before presenting the content of reviews. They use LDA to group the reviews discussing about the same thematics.

Fu *et al.* [70] propose WisCom, a system to analyze user reviews and ratings in order to identify the reasons why users like or dislike apps.

Iacob and Harrison [93] propose MARA, a tool for extracting feature requests from online reviews of apps. First, they use a set of predefined rules to identify sentences in online reviews which refer to feature requests. Then, they apply LDA for identifying the most common topics among the feature requests.

Maalej *et al.* [108] propose an approach to automatically classify reviews into for categories: bug reports, feature requests, user experiences, and ratings.

Similar to these studies we analyze reviews available in stores to extract informative feedback. Nevertheless, our work differs from these studies since we focus on reviews as an oracle of error-proneness. The reviews constitute a trigger to further analyze apps.

2.1.2 Android Permission Analysis

The *API analysis* sub-field studies the API usage extracted from apps. In Android, APIs are protected by a permission model. More specifically, we focus on the literature that analyzes the usage of permissions.

CHABADA [81] proposes an API-based approach to detect apps that misbehave regarding their descriptions. CHABADA clusters apps with similar descriptions and identifies API usage outliers in each cluster. These outliers point out potential

malware apps. In contrast, we analyze user reviews (and not on app descriptions) to automatically identify buggy apps (as opposed to identify malware).

Frank *et al.* [69] propose a probabilistic model to identify permission patterns in Android and Facebook apps. They found that permission patterns differ between high-reputation and low-reputation apps.

Barrera *et al.* [47] studied the permission requests made by apps in the different categories of the Google Play store by mapping apps to categories based on their set of requested permissions. They show that a small number of Android permissions are used very frequently while the rest are only used occasionally.

Jeon *et al.* [95] proposed a taxonomy that divides official Android permissions into four groups based on the permission behaviors (*e.g.*, access to sensors, access to structured user information). For each category, they propose new fine-grained variants of the permissions.

Chia *et al.* [54] performed a study on Android permissions to identify privacy risks on apps. They analyze the correlation between the number of permissions requested by apps and several signals, such as app popularity and community rating.

Our work differs from previous studies as we focus on permissions as a proxy for *bugginess*. Our taxonomy has a different goal, the aim of our classification is helping to identify error-sensitive permissions. Previous studies focus on official Android-specific permissions, we also analyze Google-defined, Vendor-defined and Developer-defined permissions.

Discussion

Despite the prolific research in the *app store analysis* area, none of previous approaches have proposed mechanisms to exploit the user feedback by the stores themselves automatically. We claim that app stores can automatically translate the crowd feedback into actionable insights.

2.2 Debugging

Debugging is the process of detecting, locating, and correcting faults in a computer program [131]. Debugging can take place during development, or after deployment when the software is in production. Zeller [163] identifies 7 steps in the debugging process:

1) Track the problem, 2) Reproduce the failure, 3) Automate, 4) Find infection origins, 5) Focus on likely origins, 6) Isolate the infection chain, and 7) Correct the defect.

This thesis aims to assist app developers to debug mobile apps after deployment. In particular, we focus on supporting steps 1, 2 and 7. Our goal is to assist developers to **detect**, **reproduce**, and **patch** failures. We review the literature in the subareas corresponding to each of these debugging steps. To track these problems, we review previous works in the *crash reporting systems* sub-area. To reproduce failures, we review the literature in the sub-area of *field failure reproduction*. Finally, to correct defects, we review existing approaches in the *automated patch generation* sub-area.

2.2.1 Crash Reporting Systems

Crash reporting systems collect information (such as stack traces and core dumps) when a program crashes and send it back to developers. In the mobile platform, there is a range of crash reporting tools, for example: CRASHLYTICS [cra], SPLUNK [splunk], ACRA [ACRA], and GOOGLE ANALYTICS [googleanalytics]).

These systems collect raw analytics on the execution of apps. Crash reports (collected by crash reporting systems) typically lack steps to reproduce failures [110]. Thus, developers face problems when reproducing failures experienced by users. Knowing the steps that lead to failures and the context under which failures arise, it is essential to reproduce failures [163]. However, none of previous crash reporting systems synthesize steps to reproduce crashes neither the context that induce the failures.

This thesis goes beyond current crash reporting systems by exploiting crowd feedback in a smarter way. The APP STORE 2.0 provides developers *reproducible scenarios*, which define the steps to reproduce crashes and the context that induce the failures. The *reproducible scenarios* are crowd-validated before delivery to developers.

2.2.2 Field Failure Reproduction

The *Field Failure Reproduction* sub-area studies techniques to assist developers to reproduce in-lab crashes that are faced by end-users after release.

BugRedux. Jin and Orso [96] introduce BUGREDUX, an approach that applies symbolic execution and constraint solving techniques to recreate field failures for desktop programs. BUGREDUX implements 4 reproduction strategies by taking as

input 4 different types of failure data: the crash location, the crash stack trace, the call sequence, or the complete execution trace. BugRedux is implemented for the C language.

BUGEX. Röβler et al. [133] propose the approach BUGEX that leverages test case generation to systematically isolate failures and characterize when and how the failure occurs. The input of BUGEX is a failing test case. BUGEX uses an evolutionary approach to generate executions which are identical to the input failing test. BUGEX is implemented for the Java language.

ReCrash. Artzi et al. introduce ReCrash [42], a technique to generate unit tests that reproduce failures in Java programs. ReCrash stores partial copies of method arguments in memory during program executions to create unit tests to reproduce failures. ReCrashJ is an implementation of ReCrash for Java.

STAR. Chen and Kim presents STAR [53], a framework to automatically reproduce crashes from crash stack traces of object-oriented programs. The input of STAR is the stack trace of a crash. STAR is implemented for the Java language.

Despite the prolific research in this area, none of the aforementioned approaches are available for mobile platforms. Mobile apps pose additional challenges for the reproduction task—*i.e.,* context-induced crashes that are invisible from the apps' code.

We identify the following reproduction approaches for the mobile platform:

Crashdroid. *Crashdroid* [152] is an approach to automatically generate steps to reproduce bugs in Android apps. *Crashdroid* takes as input the call stack from crash reports and generates a sequence of steps and a replayable script to reproduce the crash. Developers need to perform a preprocessing in the apps. In particular, they need to provide natural language descriptions of different scenarios of the apps under test. *Crashdroid* cannot characterize contexts under which crashes arise, consequently it cannot isolate device-specific crashes.

Table 2.2 summarizes the state of are in the *Field Failure Reproduction* sub-area.

This thesis complements existing approaches by providing a solution that can synthesize steps and characterize contexts to reproduce crashes, without any preprocessing from developers. Thus, we can deal with context-specific crashes which are the most challenging for developers.

Table 2.2 Summary of related work in the *Field Failure Reproduction* sub-area

Approach	Language	Technique	Mobile	Context
BugRedux [96] (2012)	C	Symbolic execution	No	-
BUGEX [133] (2012)	Java	Evolutionary approach	No	-
ReCrashJ [42] (2008)	Java	Record&Replay	No	-
STAR [53] (2014)	Java	Stack-trace based	No	-
Crashdroid [152] (2015)	Android	Stack-trace based	Yes	No

2.2.3 Automated Patch Generation

The last step of the debugging process is the correction of the defect to avoid their emergence. The *Automatic Software Repair* area explores approaches for *'automatically finding a solution to software bugs, without human intervention [115]'*. A *patch* is *'any modification to a source or object program'* [131]. The modification could require or not require recompiling the source program. In this section, we summarize the sub-area that concerns the automatic generation of patches.

Monperrus [115] presents a literature review on *Automatic Software Repair* approaches. A huge number of automatic repair approaches have been proposed for desktop programs. These approaches use techniques which span over *Genetic Programming* (*e.g.*, *GenProg* [102]), *Rollback and Checkpoint* (*e.g.*, *Assure* [141]), Probabilistic and Machine Learning techniques (*e.g.*, *ClearView* [129]), and Model-checking techniques (*e.g.*, *AutoFix-E* [128]) among others.

Other approaches focus on *automatic recovery* from failures as part of the repair. For example [52, 71] have proposed successful mechanisms to recover Java programs from failures (unhandled exceptions) at runtime. Pagano *et al.* [127] propose *FastFix*, a platform that remotely monitors web or desktop applications to identify symptoms of incorrect execution or performance degradation. However, these approaches are not available for the mobile platform.

Despite this research area has been extensively explored in desktop programs, the automated patch generation for mobile apps is still in its infancy.

In the mobile domain, Azim et al. [45] propose an approach based on patch construction and application restart to provide self-healing capabilities to recover Android apps from certain kinds of faults. Their patching strategy consists of rewriting the bytecode of apps to insert try/catch blocks to wrap methods that throw unhandled exceptions, and to restart the system to a safe GUI state after the app experiences a failure. They

focus on runtime recovery from crashes. Their approach is reactive—*i.e.*, it initiates after a crash happens. In addition, since their approach is based on binary rewriting, apps need to be reinstalled after the patch is applied.

Choi and Chang [55] propose a new component-level exception mechanism for Android apps to recover from unexpected exceptions.

All previous approaches share a similar goal to this thesis, make programs more resilient to faults. We focus on learning from failures to prevent more manifestation of crashes, while the developer fixes the app. In addition, our approach is fed by crowd feedbacks and continuously learns to improve its repair strategy.

Discussion

The debugging process has been extensively studied for desktop programs and currently there is a huge number of assisting tools. Nevertheless, there is a lack of approaches and tools to assist the post-deployment debugging for mobile platforms. The debugging process posses additional challenges in mobile environments due to high device fragmentation, rapid platform evolution (SDK, OS), and diverse operating context (*e.g.*, sensors).

2.3 Crowd Monitoring

Another family of approaches have monitored a multitude of devices in the wild with different purposes. Table 2.3 summarizes the most relevant approaches in the *Crowd Monitoring* area.

Table 2.3 Summary of related work in the *Crowd Monitoring* area

Approach	Type of bug	Platform	Goal
MobiBug [29] (2010)	Functional	Not implemented	Failure debugging
AppInsight [132] (2012)	Performance	Windows Phone	Performance analysis
Carat [124] (2013)	Energy	Android	Energy diagnosis

MobiBug. Agarwal et al. [29] propose MobiBug, a collaborative debugging framework that monitors a multitude of phones to obtain relevant information about failures. This information can be used by developers to manually reproduce and solve failures. This thesis synthesizes *reproducible scenarios* to enable developers to automatically reproduce crashes.

AppInsight. AppInsight [132] is a system to monitor app performance in the wild for the Windows Phone platform. AppInsight instruments mobile apps to automatically identify the critical path in user transactions, across asynchronous-call boundaries. However, they do not synthesize scenarios to reproduce crashes.

Carat. Oliner et al., [124] introduce Carat, a collaborative process in top of a crowd of mobiles devices for diagnosing energy bugs in mobile apps. Carat is implemented for Android apps.

Previous approaches have exploited the crowd to diagnose different types of bugs in mobile apps. Similar to these approaches, this thesis exploits the crowd to extract knowledge. This thesis also translates crowd knowledge into actionable insights automatically for developers. Finally, we exploit the crowd to asses the generated insights.

2.3.1 Monitoring User Interactions to Support Bug Reproduction

Previous research have monitored user interactions for testing and bug reproduction purposes in Web ([87]) and desktop applications (*e.g., FastFix* [136], [137]).

In the mobile domain, *MonkeyLab* [105] is an approach to mine GUI-based models based on recorded executions of Android apps. The extracted models can be used to generate actionable scenarios for both natural and unnatural sequences of events. However in *MonkeyLab*, apps are exercised by developers in lab. This thesis aims to synthesize realistic scenarios form user traces collected in the wild. In addition, our approach also deals with context information, since context is crucial to reproduce failures in mobile environments.

Discussion

There is a lack of crowdsourced solutions to assist developers to automatically detect, reproduce, and patch failures in mobile apps after their deployment in the wild.

2.4 Mobile App Testing

Currently, a wide range of testing tools for Android apps are available. These techniques focus on aiding developers to detect potential failures before the release of apps. This

section reviews three sub-areas of app testing related to this thesis: *Automated UI testing*, *Record and Replay* techniques, and *Performance Testing*.

2.4.1 Automated UI Testing Tools

First, there are several frameworks that support automatic testing of mobile apps. Android provides the tools *Espresso* [34], and *UI Automator* [36]. In addition, other popular app testing frameworks are: *Calabash* [Calabash], *Robotium* [rob], and *Selendroid* [selendroid]. These frameworks enable testing apps without having access to the source code. These frameworks require the developer to manually define the test scenarios to execute. Thus, they risk to lack unexplored code of the apps.

In addition, there is a broad body of research in automated input generation for mobile apps. These tools automatically explore apps with the aim of discovering faults and maximizing code coverage. These tools can be categorized in three main classes concerning the exploration strategy they implement [57]:

Random testing: These tools generate random UI events (such as clicks, touches) to stress apps to discover faults. The most popular tool in this category is *Monkey* [35], provided by Android. *Dynodroid* [112] is also a random tool which generates UI and system events.

Model-based testing: The second class of tools extract the GUI model of the app and generate events to traverse the states of the app. The GUI models are traversed following different strategies (*e.g.* breath or depth first search) with the goal of maximizing code coverage. Examples of tools in this category are: MobiGUITAR [31], PUMA [86], and ORBIT [56].

Systematic testing: The third class of tools use techniques, such as symbolic execution and evolutionary algorithms, to guide the exploration of the uncovered code. Examples of these tools are: Sapienz [114], EvoDroid [113], ACTEve [33], A^3E [46], and AppDoctor [90].

Despite the rich variety of approaches, previous tools lack support to test apps under different execution contexts—*i.e.,* networks, sensor states, device diversity, etc. Therefore, they cannot detect context-related bugs which are inherent to the mobile domain.

There exist cloud services to test an app on hundreds of devices simultaneously on different emulated contexts. Caiipa [103] is a cloud-based testing framework that applies the contextual fuzzing approach to test mobile apps over a full range of mobile operating contexts. In particular, they consider three types of contexts: *wireless network conditions*, *device heterogeneity*, and *sensor input*. However the concepts are generic to any mobile platform, current implementation of Caiipa is only available for Windows Phones. There are other cloud-based commercial solutions where developers can upload their apps to test them on hundreds of different devices simultaneously, such as Google Cloud Test Lab [Google], Xamarin Test Cloud [xamarin] and testdroid [tes].

Despite the prolific research in the automated UI testing sub-area, testing approaches cannot guarantee the absence of unexpected behaviors in the wild. This thesis aims to complement existing testing solutions, with a post-release solution to help developers to efficiently detect and debug failures faced by users.

2.4.2 Record and Replay

There is also a set of techniques that capture and replay user and system events for GUI test automation.

The *Android getevent tool* [getevent] runs on devices and records kernel input events. *RERAN* [73] is a tool to record and replay kernel event traces. *RERAN* uses the *getevent* tool to record and generate a replay script. Similarly, MobiPlay [165] is a record and replay technique based on remote execution.

The inputs recorded by these tools are based on screen coordinates. Thus, these tools only can replay the scripts in the same devices where they were recorded. To overcome the fragmentation problem, MOSAIC [85] and VALERA [91] extend RERAN to generate platform-independent scripts which can be recorded and replayed in different devices.

2.4.3 Performance Testing

In addition to functional testing, developers also need to test the performance of their apps.

Liu *et al.* [106] identify three types of performance issues in mobile apps: *UI lagging*, *memory bloat*, and *energy leak*. Additionally, they present PerfChecker, a static code analyzer to identify such performance bugs. Later, Liu *et al.* [106] present an approach for diagnosing energy and performance issues in mobile Internetware applications.

Other approaches have focused on detecting UI performance defects. Yang *et al.* [160] propose a test amplification approach to identify poor responsiveness defects in android apps. Ongkosit *et al.* [125] present a static analysis tool for Android apps to help developers to identify potential causes of unresponsiveness, focusing on long running operations.

The aforementioned approaches focus on stressing apps to expose UI responsiveness lags. Nevertheless, the performance of the app highly varies depending on the execution context. For example, one app can perform well on a set of devices, but exhibit performance issues in different devices or different OS version. There is a lack of automated testing approaches to detect UI performance issues that take into consideration different execution contexts.

2.4.3.1 Performance Regression Testing

When a new version of software is released (for example after fixing bugs), developers conduct *regression testing* [153]. *Regression testing* implies validating the new release against prior releases. This is typically achieved via execution of "regression tests" (tests used to validate prior releases). *Performance regressions* are defects caused by the degradation of the system performance compared to prior releases [67]. Thus, developers assess if the new release fulfills their expected performance goals—*i.e.*, preventing performance degradations, improving performance, etc.

Performance regression testing has been vastly studied in desktop and web application domains. Foo *et al.* [68] introduce an approach to automatically detect performance regressions in heterogeneous environments in the context of data centers. However, *performance regression testing* has received less research interest for mobile apps. The rapid evolution of the mobile ecosystem (OSs, APIs, devices, etc.) and the diversity of operating conditions (network, memory, etc.) make it difficult to guarantee the proper performance of mobile apps running on different environments. Thus, detecting performance degradations among different app versions and execution contexts is crucial to ensure high quality apps. This thesis aims to complement existing performance

testing solutions with a performance regression approach that takes into consideration the highly diverse execution context.

Discussion

Testing tools cannot ensure the absence of unexpected failures in the field. Even if apps are extensively tested before release, unexpected errors can arise due to the rapid platform and fragmentation evolution.

2.5 Conclusion

This chapter reviews the most relevant approaches in the research areas and subareas that are closely related to this thesis. This thesis falls in the intersection between: *app store analysis*, *debugging*, *crowd monitoring*, and *app testing*.

Among the prolific research in these areas, there is a lack of automated solutions for the mobile platform to debug apps in the wild which fully support:

- collection of realistic execution traces for debugging,
- generation and validation of scenarios to reproduce bugs in the field,
- generation and validation of patches that avoid bugs in different groups of devices and contexts,
- detection of UI performance defects in different groups of devices and contexts.

This thesis contributes a crowdsourced monitoring solution to help developers to debug mobile apps after release. This framework can be incorporated into app stores, which can provide actionable insights for the stakeholders of the mobile ecosystem.

Chapter 3

The Vision of App Store 2.0

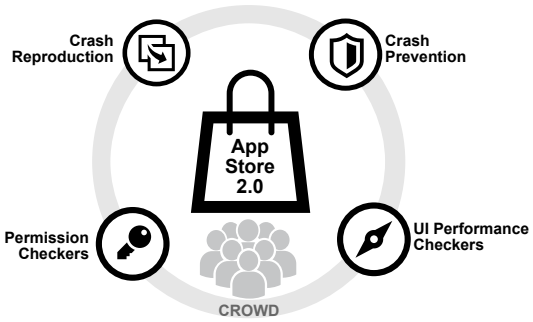


Fig. 3.1 Chapter 3 at a glance

The proliferation of smartphones is leading to a rapid growth in the development of mobile applications (apps). Currently, app developers can chose among a large ecosystem of app stores to publish their apps [lis]. Stores enable developers to run their apps on thousands of mobile devices. Neverhteless, existing app stores offer limited support for improving the quality of mobile apps. When an app has a bug, the store continues spreading the defective app across users and devices, until the developer uploads a fixed version. Unfortunately, the release time of a new app version can be long [117]. Thus, impacting negatively on the reputation of the app, the developer, and the store.

This thesis proposes leveraging the ***wisdom of the crowd*** to improve the quality of mobile apps. This research investigates a *crowd-based framework* to engineer a new generation of app stores—App Store 2.0. The App Store 2.0 enhances current app store services with new functionalities to assist developers to deal with different types of defects in mobile apps. The goal is to continuously capture the quality of experience of a crowd of mobile users and provide actionable insights to developers and users.

The reminder of this chapter presents the vision of App Store 2.0. Section 3.1 summarizes the proposed crowd-based approach to deliver actionable insights from app

stores. Section 3.2 describes the main building blocks which compose the approach. Finally, Section 3.3 concludes the chapter.

3.1 App Store 2.0 Overview

App stores have access to three huge crowds: **crowd of apps**, **crowd of devices**, and **crowd of users**. Nevertheless, current app stores do not exploit the diversity of available crowd sources to provide actionable insights to developers and users. Figure 3.2 reports on the different types of crowds and their volume [126]. The huge amount of possible permutations results in a complex ecosystem.

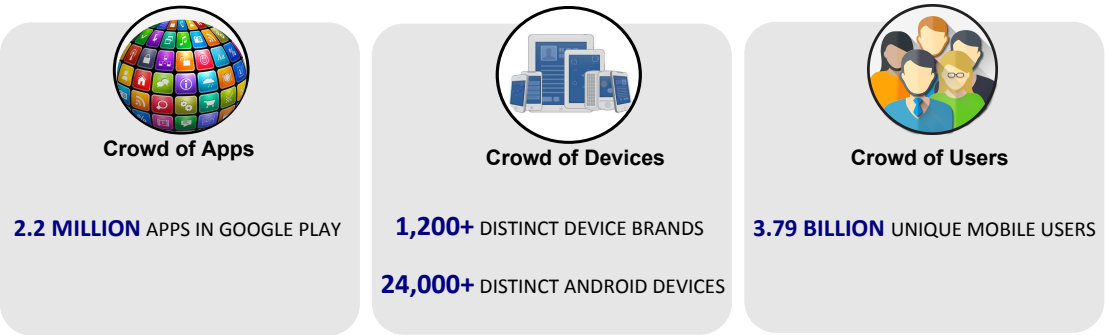


Fig. 3.2 Types and volume of crowds in app stores

Our key insight is that the wisdom of those crowds can be combined, and the resulting sum is more powerful than each one in isolation. The combination of these crowds results in 3 types of crowdsourced information—*app execution logs*, *app context*, and *user feedbacks*. App stores can leverage the value of this crowdsourced information to deliver new services.

This thesis provides a collection of approaches to generate actionable insights by using different types of crowdsourced information. The actionable insights aim to assist developers to deal with 2 types of bugs in mobile apps: *crashes* and *UI janks*.

The App Store 2.0 orchestrates a feedback loop that continuously analyses crowd feedbacks to detect and eventually fix defective apps. In the presence of defects, the App Store 2.0 takes self-initiated measures to generate actionable insights. In particular, the App Store 2.0 incorporates two types of approaches:

1. ***In vitro* approaches.** Upon submission of new apps, stores take preventive actions before making the apps publicly available to users. Preventive actions

refer to predictions of problems (through crowd feedback analysis) and attempts to avoid their occurrence in the field.

2. ***In vivo* approaches.** After users install and execute apps, if problems surface—such as crashes—the store immediately takes actions to assist the developer to fix and prevent such issues. These reactive approaches actuate when apps are in hands of end-users.

The different mechanisms are complementary and can be activated simultaneously. App Store 2.0 moderators decide upon which measure(s) to activate. Figure 3.3 shows an overview of the App Store 2.0.

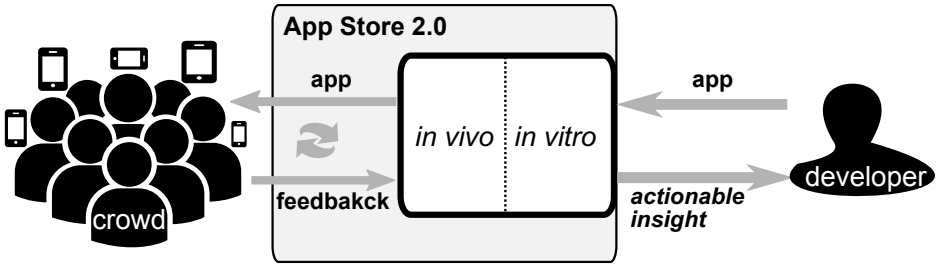


Fig. 3.3 Big picture of Smart Stores 2.0.

Three target audiences can benefit from the proposed approach:

- *App stores.* Currently there is a huge range of app stores available, all competing to attract customers (developers and users). App stores can implement the presented App Store 2.0 approach to enhance their services, improving the quality of apps and consequently the store's reputation.
- *App developers.* Developers want to deliver high quality apps to survive the market competition. By using this approach, they can increase the quality of their apps, thus contributing to improve their users' satisfaction and loyalty.
- *App users.* Users want high-quality apps that ensure a high user experience (*e.g.,* absence of crashes and responsive apps).

3.2 Main Building Blocks

The approach to engineer App Store 2.0 contains two main building blocks: `Monitoring the Crowd` and `Leveraging the Crowd`. At the `Monitoring Block`, crowd feedback

is continuously collected from different sources—*i.e., user feedbacks*, *app execution logs*, and *app context information*. The *crowd feedback* constitutes the knowledge to guide the autonomous decision-making process in the store. At the `Leveraging Block`, the crowd feedback is exploited to deliver actionable insights for the stakeholders of the mobile ecosystem. These actionable insights will assist developers during the process of detecting, reproducing, and fixing mobile app bugs. Figure 6.2 shows the main building blocks which compose the proposal. We envision 4 modules to generate 4

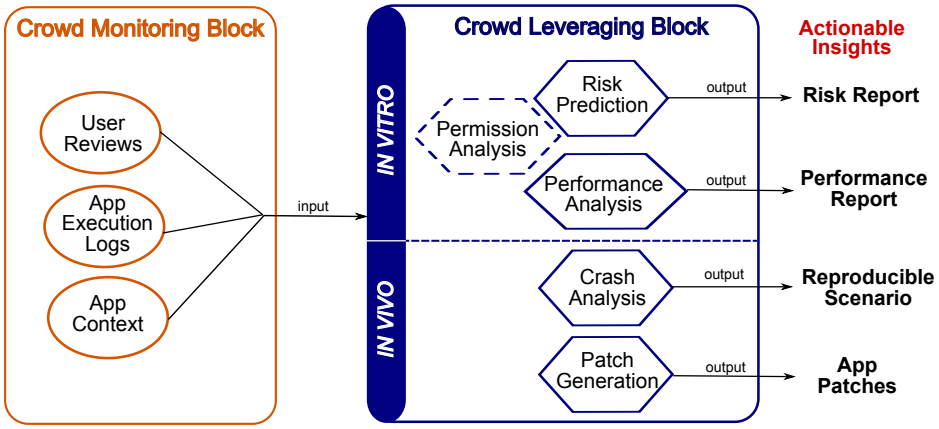


Fig. 3.4 Main building blocks of the proposal.

different types of actionable insights. These actionable insights span over *risk reports*, *performance reports*, *reproducible scenarios* and *app patches*.

Figure 3.5 illustrates the workflow of the App Store 2.0. An app developer uploads an app to the App Store 2.0 for distribution. Before making the new app publicly available to users, the store runs a `Risk Analysis` to predict potential crashes. The risk analysis is based on observations of user feedbacks reported by other apps executed by the crowd. If there is a risk of crash, the store sends a `risk report` to the store moderator, who can notify the developer or decide to publish the app anyway. When the developer uploads the new fixed release, the store runs a `Performance Analysis` to ensure that the new release does not perform worse than the previous release. A detailed `performance report` is provided to the developer, who can update the app to fix the performance defects prior to its publication.

When the app is downloaded and executed on the users' mobile devices, a `Crash Analysis` service listens for crash occurrences in the wild. In the presence of crashes, the App Store 2.0 learns crash and context (*i.e.*, software and hardware configurations) patterns, which are turned into a `reproducible scenario` to help developers to quickly reproduce the observed errors. Meanwhile, as the release process can be long, the store

generates patches to prevent that other users continue facing the same observed issues. Afterwards, the store keeps monitoring the information crowdsourced from devices and user feedback as an oracle for the autonomous improvement process.

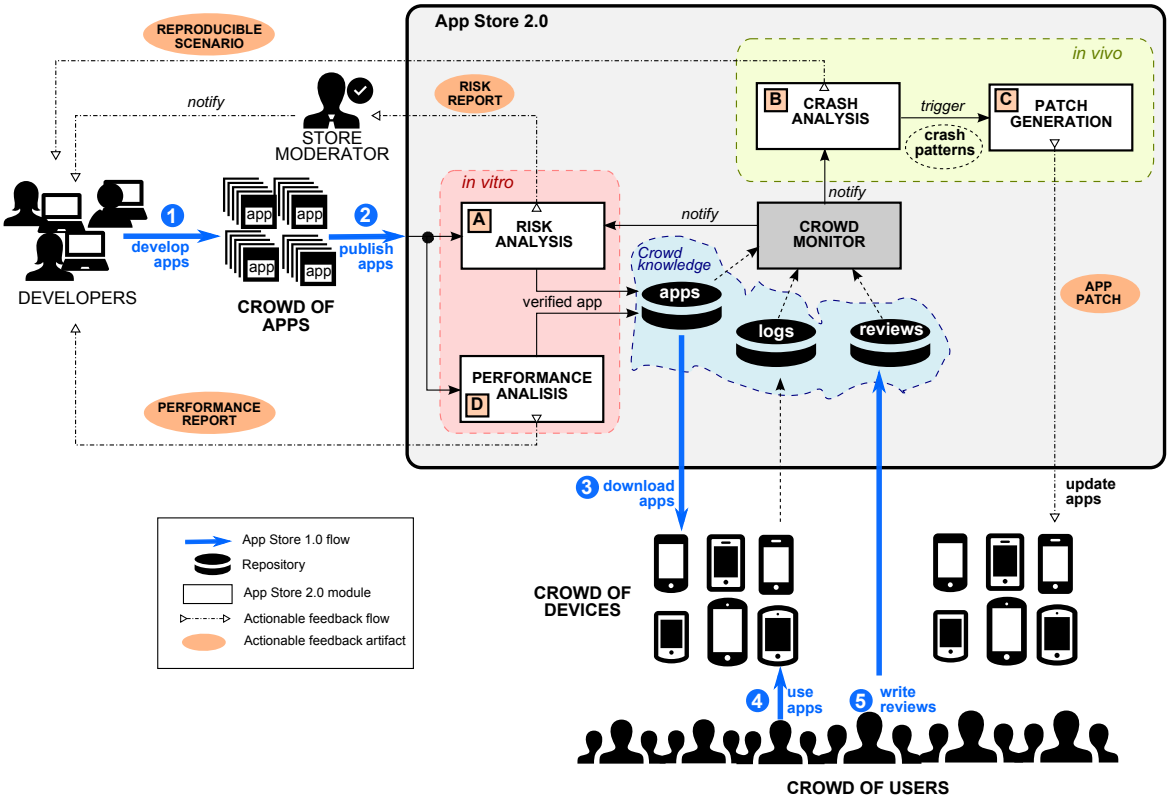


Fig. 3.5 App Store 2.0 workflow.

3.2.1 Crowd Monitoring Block

This building block is associated with a `crowd monitor` component, which continuously collects crowd information from different sources: *user feedbacks* posted in the store, *app execution logs*, and *app context information* derived from the execution of apps in devices by users. Table 3.1 illustrates different types of crowdsourced information and their volumes.

The collected crowd information is used *a posteriori* with two different purposes:

- *Defect Detection.* Detecting when users experience issues (*i.e.,* crashes, UI lags) with apps.

Table 3.1 Types and Volumes of Crowdsourced Information in App Stores.

Crowdsourced information		
Contexts	23	distinct Android OS API levels [126]
	1,411	distinct requested permissions [78]
Reviews	228+ million	user reviews in Google Play [41]
Logs	60+/day/app	apps are run by 280 million users [66]

- *Continuous Validation.* Validating the quality of the measures adopted by the store to continuously improve its strategy.

3.2.2 Crowd Leveraging Block

This building block is composed by 4 main modules to deliver 4 different types of actionable insights: *risk reports*, *performance reports*, *reproducible scenarios*, and *app patches*. The first two modules are executed *in vitro* before releasing apps. Whereas the two latter modules are executed *in vivo* once apps are in hands of end users.

3.2.2.1 *In vitro* approaches:

- **Risk Prediction.** The App Store 2.0 can build *crowd-based checkers* to rank the risk of a crash in newly submitted apps. The `Risk Prediction` module relies on an additional module: `Permission Analysis`. The crowd monitor component continuously supervises user reviews published in the store. Once the store identifies a cluster of crash-prone apps, the `Permission Analysis` component searches for recurring permission patterns (requested by the apps) that correlate with the crashes. The `Permission Analysis` component then creates a predictive machine-learning model to predict if a new app will crash based on the set of requested permissions. The resulting predictive model constitutes the basis of the *crowd-based checkers* to score the risk of crashes of new apps at submission time.

  Tool support: This thesis provides a `Recommender System of Buggy Permission Checkers` to predict crash-prone apps depending on the set of permissions requested. This system is built from an empirical study of 46,644 apps and their 1,402,717 user reviews from the Google Play Store.

- **Performance Analysis.** One important category of performance defects in Android apps is related to poor *GUI responsiveness*. The main research challenge of automatically identifying UI performance problems on mobile devices is that the performance of an app highly varies depending on its context—*i.e.,* the hardware and software configurations on which it runs. By leveraging the wisdom of the crowd of devices, the store can incorporate UI Performance Checkers to flag UI performance defects as outliers in metrics monitored across different app executions and heterogeneous contexts.

  Once an app developer uploads a new release of an app to the store, the `Performance Analysis` component compares the UI performance of the new app release and the previous releases to assess whether the new app release fulfills the expected performance goals—*i.e.*, to prevent performance degradations.

  Tool support: This thesis contributes the `DUNE` tool. `DUNE` is a context-aware approach to detect UI performance deviations between different app releases and contexts.

3.2.2.2 *In vivo* approaches:

- **Crash Analysis.** Once an app is downloaded and executed on the users' mobile devices, the store listens for the emergence of crashes. When crashes surface, the `Crash Analysis` component aggregates the crowdsourced execution logs (collected from a multitude of devices) into a weighted directed graph. This graph provides an aggregated view of interactions from a multitude of users in a defective app before a specific crash arises. The store uses the graph to identify repeating patterns of UI events and contexts that appear frequently among crashes. The patterns are translated into a *reproducible scenario* to automatically recreate the crashes faced by users.

  Tool support: This thesis provides the `MoTiF` tool. `MoTiF` uses machine learning techniques atop of data crowdsourced from real devices and users. MoTiF is composed of two parts: 1) a *mobile client library*, that runs on mobile devices, to monitor app executions and crashes; and 2) a *cloud service* to learn and generate reproducible scenarios.

- **Crash Prevention: Patch Generation.** While developers are working on fixing the apps, the app store generates temporary *patches* to prevent the occurrence

of the same crash for different users. The `Patch Generation` component creates two types of patches: *inserting try/catch blocks* in suspicious locations of code, and *deactivating the crash-triggering features* of crashes. When the store receives a download request for an app that has been previously flagged as defective, the store delivers an alternative patched release of the app. Afterwards, the store keeps monitoring crowdsourced information from devices and user feedback that run those patched apps to assess the effectiveness of the generated patches. If a patch generation technique fails (*e.g.*, the patched app keeps crashing), the store learns from these failures. The App Store 2.0 continuously monitors crowdsourced information to improve the patching process.

Tool support: To inject the patches, this thesis provides the `CrowdSeer` tool. The `CrowdSeer` tool implements two patching strategies. The former instruments the bytecode of Android apps to inject try/catch blocks. The latter deactivates the crash-triggering features of apps at run-time. While the first strategy, requires reinstall the apps. The second strategy update apps on the fly.

Chapter 5 and Chapter 6 provides further details regarding each of the modules.

3.3 Conclusions

This chapter outlines a crowd-based framework to engineer the App Store 2.0. The goal of the App Store 2.0 is to improve the quality of the delivered mobile apps. The main idea is that collecting information from a multitude of devices and users it is possible to extract relevant knowledge to assist app developers to automatically detect, reproduce and patch mobile app bugs.

The approach consists of two main building blocks: *monitoring* and *leveraging* the crowd. The proposed crowd-based approach reports the following benefits. First, it increases the crash tolerance of apps, thus enhancing user experience. Second, it improves reactivity to detect and avoid crashes, thus avoiding to harm the app reputation. Third, it automates the bug-fixing process, thus reducing human intervention to maintain mobile apps. Four, it updates apps on-the-fly, without requiring to release a new version of the app, thus reducing the time that users are exposed to crashes, and reducing the number of affected users.

Part II

Contributions

In part I, we introduced the vision of the App Store 2.0. In this part, we sketch our solution towards making it reality.

Chapter 4

Monitoring the Crowd

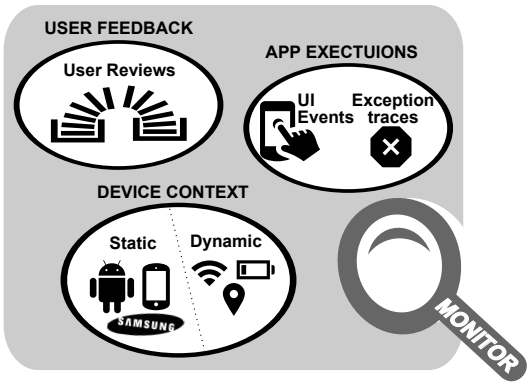


Fig. 4.1 Chapter 4 at a glance

Current app stores are software repositories that deliver apps upon user requests. The architecture of current app stores is composed of two parts: a) a *client app* that runs in mobile devices and enables to download and install apps; and b) a *centralized server side* that enables developers to upload apps for distribution and obtain feedback about their apps. Given this architecture, app stores have access to a huge crowd of devices, users, apps, and developers. However, current app stores serve as distributors of apps and do not exploit the available crowd sources. We claim that by leveraging crowdsourced information it is possible to assist developers to deal with bugs which affect mobile apps, such as crashes and UI janks. These two types of bugs have a negative impact on user experience.

This chapter describes the *monitoring block* which current app stores can incorporate to gather information from the crowd. In particular, three types of crowd information can be accessed from stores: *user feedbacks, app execution logs, and app context.*

The reminder of the chapter is organized as follows. We first describe the main types of bugs which affect mobile apps in Section 4.1. Afterwards, Section 4.2 discusses types of crowd-sourced information which are accessible from app stores, and presents

techniques to monitor such sources in app stores. Finally, Section 4.3 concludes the chapter.

4.1 Types of Bugs in Mobile Apps

We start by reviewing the literature to identify different types of bugs which affect mobile apps. Based on the types of bugs identified by Zaeem et al. [162] and Liu et al. [107], we divide mobile app bugs in two main groups: *functional bugs* and *performance bugs*. Table 4.1 shows different categories of app bugs identified by previous work.

Table 4.1 Types of Bugs in Mobile apps.

Functional bugs [162]	**Performance bugs** [107]
*Unhandled Exceptions**	*GUI lagging**
Rotation	Energy leak
Activity Lifecycle	Memory bloat
Gestures	
Input handling	
Third party lib	

In this thesis, we focus on *Unhandled Exceptions* (*i.e.*, crashes) and *GUI lagging.* (*i.e.*, janks) because these types of issues are directly perceivable by users and have a negative impact on user experience [88]. In the following subsections, we further describe these two types of defects. We claim that app stores can contribute to mitigate these issues by exploiting crowdsourced information.

4.1.1 App Crashes

If an app implements inefficient error-handling mechanisms in its source code, then the app may throw unhandled exceptions during execution and the operating system terminates the app. This behavior is commonly referred to as a *crash*. We focus on app failures which manifest with crashes, hence we first study which are the main causes of app crashes. Kechagia *et al.* [97] identify causes of Android app crashes within a dataset of stack traces collected from real devices. In addition, Liang *et al.* [103] identify *context-related bugs* in mobile apps, *i.e.*, *network conditions*, *device*

Table 4.2 Categories of Android app crashes.

Cause	**Sample app**	**App crashes**
Missing or corrupted resource	PocketTool	*If the* Minecraft *game is not installed on the device*
Indexing problem	Ermete SMS	*Deleting a phone number taken from the address book*
Insufficient permission	ACV	*Long-pressing a folder*
Memory exhaustion	Le Chti	*After some navigation steps in the app*
Race condition or deadlock	Titanium	*Clicking the back button during the app launch*
Invalid format or syntax	PasswdSafe	*Opening a password that contains Norwegian characters*
Network conditions	Wikipedia	*Attempting to save a page without network connectivity*
Device heterogeneity	Wikipedia	*Pressing the Menu button on LG Devices*
Sensor input	MyTracks	*When GPS is unavailable*

heterogeneity and *sensor input*. Table 4.2 summarizes these categories of Android app crashes, together with a sample app exhibiting such a type of crash.

In particular, the crashes that depend on context are more challenging to isolate and to reproduce by developers in the lab [29]. We aim to complement existing in-house testing solutions with a collaborative approach that monitors apps after their deployment in the wild.

4.1.2 UI Jank

In Android, a lag in the UI is called a *jank* [Tes], particularly when dealing with scrolling, swiping, or other forms of animation [64]. To ensure smooth animations, apps must render at *60 frames per second* (FPS), which is the rate that allows animations to be perceived by the human eye [60f]. Most Android devices refresh the screen every *16 ms* ($\frac{1\ sec}{60\ fps} = 16\ ms$ per frame). Consequently, any frame taking more than 16 ms will not be rendered by Android, and hence will be flagged as a jank. In this case, users will experience choppy animations and laggy GUI. Performing heavy computions in the UI thread of the app is one of the most common practice that leads to janks. Google therefore encourages developers to test the UI performance of apps to ensure that all frames are rendered within the 16 ms boundary [60f].

The main research challenge of automatically identifying janks on mobile devices is that the performance of an app highly varies depending on its context—*i.e.,* the hardware and software configurations on which it runs. For example, an app can perform well

on a set of devices, but it may exhibit janks in a different environment consisting of, amongst others, a different device model.

4.2 What Information to Monitor from the Crowd?

To support developers to deal with defects in mobile apps, different sources of information can be monitored from the crowd. The main component in the monitoring block is the `App monitor` which gathers crowd feedback from different sources. We have identified three types of crowdsourced information:

- **User feedbacks**. Reviews and ratings which mobile users post in the store regarding the apps.
- **App context**. Information related to the operating context where the app runs—*e.g.*, device model, SDK version, memory, and state of sensors.
- **App execution logs**. Information derived from the execution of apps—*e.g.*, user interaction traces, exception traces. An *execution log* contains a sequence of user interactions (such as clicks) and operating contexts observed during the app execution.

App stores can incorporate the `App monitor` component to gather crowd feedback.

4.2.1 Monitoring User Feedback

The `App monitor` component incorporates a listener which continuously supervises users' feedback published in the store (*i.e.*, reviews). This component identifies apps which accumulate *crash-related reviews*—*i.e.*, those which mainly discuss about buggy behavior, to alert the presence of bugs.

We propose a four-step approach to automatically identify crash-prone apps from reviews written by users in app stores. Figure 4.2 shows an overview of the proposed approach. The approach comprises the following steps:

1. **Mine topics.** To automatically identify online reviews which treat topics related to crashes and errors, we use unsupervised machine learning, specifically *topic mining*, which can be used to automatically identify topics discussed in user reviews. For this purpose, we apply the *Latent Dirichlet Allocation* (LDA)

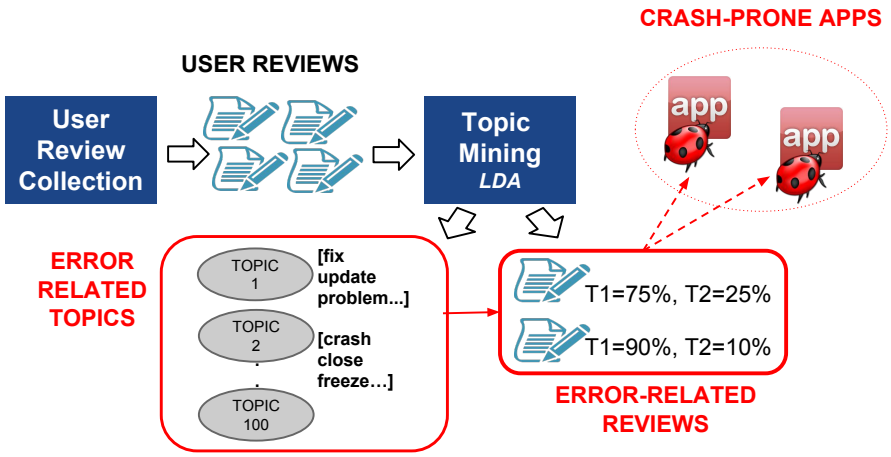


Fig. 4.2 Identifying Crash-prone Apps from User Reviews

algorithm [50]. LDA identifies topics discussed in an entire corpus of documents with unlabelled text. We consider each single review of apps as a document. A topic is a list of words which appear frequently together. For each document, LDA estimates a probability distribution over the mined topics. For example, one document may have a probability of 0.7 to relate to topic #1 and of 0.3 to belong to topic #2 (the sum is always 1). In this case, the document mostly belongs to topic #1.

The LDA model takes as an input parameter the number of topics to extract. In order to achieve a better precision, in a preliminary exploratory phase, we ran the LDA algorithm using different number of topics (*e.g.*, 20, 40, 60, and 100). Finally, we observe that selecting 100 topics the algorithm generates fine-grained topics (from user reviews) which help to identify bug-related issues. To reduce noise in the modeled topics, we filter out English stop words[1] from the entire corpus.

The rational of using topic models is to automatically (1) cluster reviews discussing about crashes, and (2) extract keywords that characterize error themes without sketching them beforehand. We are particularly interested in topic models which reveal bug-related keywords, which were unforeseen initially. This provides an oracle of error-proneness which the store will use as suspicion of problems affecting users. To implement this approach, we use MALLET (*MAchine Learning*

[1]We use the list of 524 common English stop words included in Mallet.

for LanguagE Toolkit) [116], a Java library that provides an implementation of LDA.

2. **Select error-related topics.** From the mined topics, the store filters the topics that are related to bugs and crashes. Fig. 4.3 shows three error-related topics extracted from a corpus of user reviews from Google Play Store:

> **topic1:** *fix update problem fixed bug issue crashes phone stars bugs plz pls time crashing problems working issues crash hope*
>
> **topic2:** *work doesn doesnt won didnt working open kit show load kat properly anymore sucks bad note worked android wont*
>
> **topic3:** *app crashes force time open fix close closes won crashing work crash start freezes working times constantly closing doesnt*

Fig. 4.3 Example of three error-related topics from user reviews

3. **Select crash-related reviews.** The next step consists of identifying reviews which are mainly composed by error-related topics. To select crash-related reviews, we analyze the probability distribution given by LDA for each review. Since a review can discuss several topics, we consider as crash-related reviews those with at least 5% of its probability related to a topic discussing buggy issues. Fig. 4.4 shows two examples of crash-related reviews obtained with our approach. The reviews belong to the app: *"This American Life"* (version 2.1.8) [2]:

> **Example review:** Astoundingly buggy. *Get ready for the app to crash when you want it to work, and for it to linger in your notification bar even after you force close. I have no idea how such a phenomenal show has such a disappointing app.*
>
> **Example review:** Crashes all the time. *I love TAL but this app is horrible. Actually it is a good app but it literally crashes every time I put it to use. Sometimes it will open up and I can start a show and listen to the whole thing and when it is done it crashes. Sometimes it crashes as it is opening.*

Fig. 4.4 Examples of crash-related reviews in Google Play Store

[2] https://play.google.com/store/apps/details?id=org.prx.talbot. The dates of the reviews are 13-12-2013 and 18-12-2013, respectively.

4. **Select crash-prone apps.** We consider as *error-prone apps*, the set of apps containing at least n crash-related reviews. The minimum number of error-related reviews is a parameter to be decided by app store moderators depending on the quality they want to ensure in their stores, and the confidence they have on the users reporting reviews. In this work, we have considered the minimum possible evidence of problems ($n > 1$ review).

   To sum up, by applying unsupervised machine learning on the reviews of apps, the store automatically flags each app as: *"Buggy"* or *"NonBuggy"*.

4.2.2 Monitoring App Context

To isolate bugs, it is essential to know the execution context under which bugs arise. As an illustration, users recently experienced crashes with the Android *Wikipedia* app[3], which crashed when the user pressed the menu button. However, this crash only emerged on LG devices running Android 4.1. Thus, app developers need to know the *user interactions* and the *execution context* (*i.e.*, software and hardware configuration) that led to crashes.

App stores can collect the following context information during the execution of apps:

- **Static context**. Properties that remain invariable during the whole app execution—*e.g.*, device manufacturer, device model, and SDK version.
- **Dynamic context**. Properties that change along app execution—*e.g.*, memory state, battery level, network state, and state of sensors.

4.2.3 Monitoring App Executions

Previous research have monitored user interactions for testing and bug reproduction purposes in different domains [136, 87]. To isolate bugs, when enabled, the store can log *user interaction events* during the execution of an app. An *execution log* contains a sequence of user interactions (such as clicks) and operating contexts (static and dynamic) observed during the app execution.

Android apps are UI-centric—*i.e.*, `View` is the base class for widgets. To intercept user interaction events with an app, Android provides the `View` class which provisions

[3]https://play.google.com/store/apps/details?id=org.wikipedia

different event listener interfaces that declare public event handler methods. The Android framework calls these event handler methods when the respective event occurs [4]. For example, when a view (such as a button) is clicked, the method `onClickEvent` is invoked on that object. Table 4.3 reports on a subset of available event handler methods.

Table 4.3 Examples of Android view types with their event listeners and handler methods

Type	**Event listener**	**Event handler**
View	OnClickListener	onClick
ActionMenuView	OnMenuItemClickListener	onMenuItemClick
AdapterView	OnItemClickListener	onItemClick
Navigation Tab	TabListener	onTabSelected
	OnTabChangedListener	onTabChange
Orientation	OrientationEventListener	onOrientationChanged

4.2.3.1 Logging Crash Traces

We propose a two-level monitoring strategy. Initially, the `Monitor` component only listens for unhandled exceptions (crashes) thrown during the execution of apps. When a certain percentage of users suffer from crashes with an app, the store flags the app as *defective*. Then, the system activates a lightweight monitoring mechanism to transitory gather execution logs from user interactions with the defective apps. App developers when submitting apps to the store, can set the threshold of affected users (*e.g.*, just after observing a single crash or a specific ratio) to start logging.

We define a *crash trace* (*ct*) as a sequence of events executed in an app before a crash arises—*i.e.*, $ct = \{e_1, e_2, ..., e_n\}$. Events can be of two types: *interaction* and *exception*. The last event of a logged trace (e_n) is always an exception event. Each time an event is executed, our approach records *event metadata*. If the app crashes during execution, we also collect *exception metadata*. In particular, the following metadata is recorded for each type of event:

- **Event metadata**: timestamp, method name, implementation class, thread id, and view unique id.
- **Exception metadata**: timestamp, location, and exception trace.

The static context is only reported in exception events, since it remains invariable along the whole app execution. In contrast, the dynamic context is reported for each user interaction event. Figure 4.5 depicts an example of a crash trace with two interaction events e_1, e_2, leading to an app crash $crash1$.

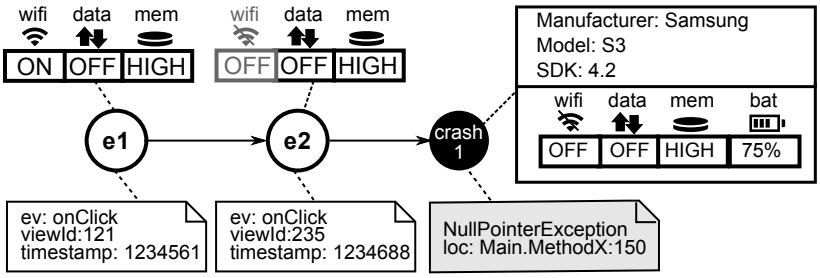


Fig. 4.5 Example of a crowdsourced crash trace.

During the execution of an app, the observed events are kept in memory. Only if the app crashes, the crash traces are temporarily stored in JSON files in the device memory, until the data is automatically flushed to the store for processing. Table 4.4 reports an example of a crash log in JSON format.

To minimize the impact on battery lifespan and the data subscription of end-users, devices only report the logs to the cloud server when the device is charging and connected to the Internet. Once uploaded, the synchronized traces are automatically removed from the local storage.

The monitoring takes place separately on different devices running the defective app. To avoid any accidental user's disturbance, only one app is monitored in each device and the monitoring is redistributed periodically among devices in the crowd.

4.2.3.2 Logging Performance Traces

During the execution of an app, the store can additionally log UI performance metrics (*i.e.*, frame rate) to estimate the UI performance of an app.

A *performance trace* (pt) logs a sequence of UI events during the execution of an app together with the following data:

1. *Performance metrics:* We focus on UI rendering related metrics, in particular the *number of frames rendered*, the *average time to render a frame*, and the *smooth ratio*. We calculate the *smooth ratio* as the ratio of rendered frames that are not

```
1   "App": "org.wikipedia",
2   "UserEvent": [{
3           "timestamp": "1440115138134",
4           "location": "<1> main org.wikipedia.search.SearchArticlesFragment$8
                onClick 262 3",
5           "Variables": {
6               "v": "instance of android.widget.LinearLayout(id=830028030432)",
7               "fields": {
8                   "v.mID": "2131099772"
9               }
10          },
11          "DynamicContext": {
12              "totalMem": "16.0MB",
13              "bluetooth": false,
14              "netType": "WIFI",
15              "maxMem": "64.0MB",
16              "nativeHeapAlloc": "4.4MB",
17              "battery": "100%",
18              "gps": true,
19              "freeMem": "0.8MB",
20              "network": "ON"
21          }
22      }],
23      "CrashEvent": {
24      "timestamp": "1440115139856",
25       "exception": "java.lang.RuntimeException:ExceptionEvent@org.wikipedia.
             concurrency.SaneAsyncTask:62 in thread <1> main",
26     "StaticContext": {
27      "androidSDK": "4.1.2",
28      "deviceModel": "Samsung Galaxy Nexus",
29      "hardware": "tuna",
30      "APIlevel": "16",
31      "manufacturer": "samsung",
32      "availableProcessors": "2"
33     }
34  }       }
```

Table 4.4 Excerpt of a *crash log*

janky:

$$\text{smooth\_ratio} = 1 - \frac{\#janky\ frames}{\#total\ frames}$$

We use the smooth ratio as a measure to determine the UI performance of an app. A higher ratio indicates better performance.

2. *Execution context characteristics:* The context under which apps are executed. In particular, we consider the following three major dimensions of execution context:

   - *Software profile:* Mobile SDK, API level, and version of the app under test;
   - *Hardware profile:* Device manufacturer, device model, CPU model, screen resolution;

- *Runtime profile:* Network type, network quality, battery level.

Note that additional information can be included in the model (*e.g.*, memory and sensor state). Developers can select the relevant characteristics they want to include in the model depending on the type of bugs they target.

Figure 4.6 depicts an example of the UI frame performance metrics collected during an app execution in a specific context. It shows the frames rendered along the app execution from the launch to the end. Each vertical bar represents one frame, and its height represents the time spent (in ms) to render the frame. The horizontal dashed line represents the speed limit (16 ms), in the sense that frames taking more time than this limit—*i.e.*, janky frames—will be skipped, leading to the app's lag. The different colors in the bars represent the time spent in each stage of the frame rendering process (execute, process, and draw). During the test run, 148 frames were rendered and 7 frames were flagged as janky.

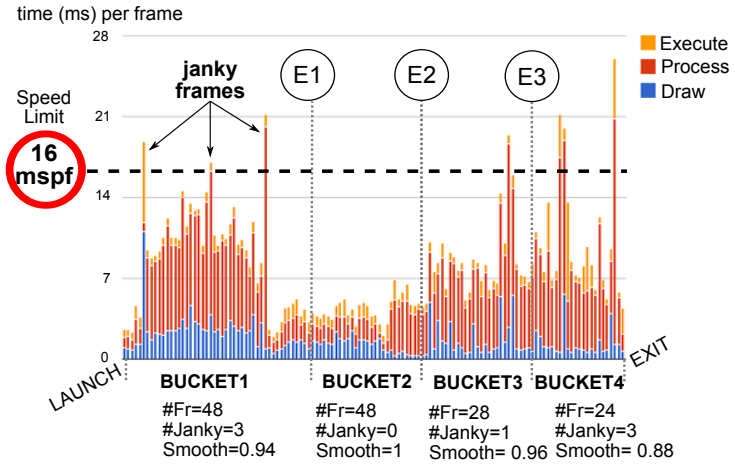


Fig. 4.6 Frame rendering metrics for an app execution divided in buckets per UI event

We propose two performance logging modes. The former is the *coarse-grained* mode which considers the metrics observed during the app execution as an aggregation. Thus, the *coarse-grained execution log* stores each performance trace (pt) as a vector: $pt = [c_1, ..., c_n, m_1, ..., m_n]$. Each c_i represents a context property, and each m_i contains a global performance metric (#janky frames, smooth ratio, rendering time) across the whole execution. For example, $pt1$ is stored as:

$$pt1_{coarse} = [v1, 4.1, samsung, S3, x86, wifi, 148, 0.95, 45]$$

In addition, the *fine-grained execution log* links the metrics to specific user input events that happened during the app execution and hence could be the trigger of janky frames. This *fine-grained* strategy aims to help developers to isolate the root cause of performance problems. With this goal, we distribute and aggregate the collected metrics into N different buckets, where N is the number of UI events plus 1. The first bucket contains all collected metrics before the first user event, the second bucket those between the first and second user event, etc.

In the example of Figure 4.6, there are 3 UI events ($E1$, $E2$, $E3$) during the app execution, which means that the frame metrics from app launch to finish are aggregated into 4 buckets. In particular, each bucket summarizes the number of frames, the number of janky frames, and the smooth ratio observed in that bucket. Thus, the *fine-grained* mode logs a performance trace as a vector that includes one dimension per bucket to store the metrics observed in those buckets. For example, $pt1$ is:

$pt1_{fine}$ =[v1, 4.1, samsung, S3, x86, wifi, [48,0.94,3], [48, 1,0], [28, 0.96,1], [24, 0.88,3]]

The metrics of each performance trace populate a repository of *historical executions* for different context configurations. This repository will compare the performance of the app with future app releases. More specifically, it will support the analysis of performance goals in future releases of the app.

4.3 Conclusions

This chapter has discussed the monitoring phase of the proposed crowd-based approach to engineer the App Store 2.0. First, we summarize different types of bugs which affect mobile apps. This thesis intends to tackle two particular bug types: crashes and janks. Then, we introduce the different types of crowd feedback available from stores: user feedbacks, context, app executions. This crowd sources constitute the knowledge to assist developers to deal with the bugs that affect their apps in the wild. Finally, we detail different approaches for collecting the available crowd sources from app stores.

Chapter 5

Leveraging the Crowd *in vitro*

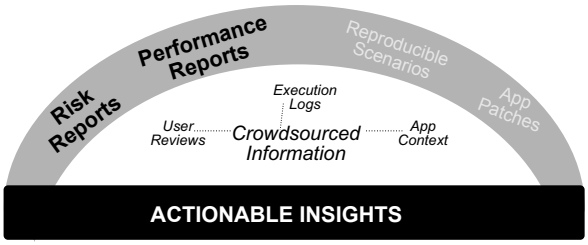


Fig. 5.1 Chapter 5 at a glance

Chapter 4 presented three different types of crowdsourced information (*i.e.*, user reviews, app context, and app execution logs) which can be accessed from app stores. As existing app stores offer limited support for improving the quality of mobile apps, we propose leveraging the existing crowdsourced information to deliver **actionable insights**. An actionable insight is defined as: *"a piece of information that enables an individual (or group) to make an informed decision. Actionable insights are typically derived by synthesizing vast amounts of data into succinct, concise statements"*[1].

This chapter presents two types of actionable insights that app stores can deliver *in vitro*, this means during the release phase. The release phase corresponds with the phase just after submitting apps to the store (deployment phase), when the apps are invisible to users [28].

We introduce three modules (`Risk Analysis`, `Permission Analysis`, and `Performance Analysis`), which exploit different types of crowdsourced information to generate two types of actionable insights: *risk reports* and *performance reports*. For each module, we describe the approach to produce an actionable insight from crowdsourced information.

[1] http://www.answers.com/Q/What_is_the_meaning_of_give_an_actionable_insight_to_someone

The reminder of this chapter is structured as follows. Section 5.1 introduces the approach to generate risk reports to predict app crashes before publishing the app. Section 5.2 details the approach to detect performance degradations between different releases of an app. Finally, Section 5.3 concludes the chapter.

5.1 Reporting Risky Apps a priori

The App Store 2.0 can build *crowd-based checkers* to rank the risk of a crash in newly submitted apps—*i.e.*, prior to publication. The basic idea is identifying a set of apps that tend to crash in hands of end-users and learning common permission patterns among these apps. Then these permission patterns enable to predict, with certain probability, when a new app will crash by observing its set of requested permissions.

To build the checkers, the `Risk Analysis` component relies on 2 types of crowdsourced information: *user reviews* and *app metadata* (*i.e.*, app permissions). The approach to generate risk reports from crowdsourced information consists of five steps, illustrated in Figure 5.2, together with the techniques and tools used. (1) The store starts by identifying apps for which users reported crashes in user reviews (cf. `Monitor` component, Section 4.2.1). (2) The second step consists in analyzing the existing permissions declared by apps. We define a taxonomy to characterize the types of permissions which Android apps can request (cf. Section 5.1.2). (3) Taking as input the crash-prone apps in step 1 and the permission classification performed in step 2, we automatically mine permission patterns which correlate with crashes (cf. Section 5.1.3). These patterns constitute the knowledge to construct checkers to predict if an app will potentially crash. App store moderators can activate these app checkers to score the quality of new submitted apps. Then, the store can notify developers about the existence of potential bugs in the app before making it publicly available to users.

5.1.1 Empirical Study of Google Play Store

In this section, we describe the dataset we built to perform an empirical study to generate the crowd-based checkers.

We built a dataset that contains a random sample of all the mobile apps available on the Google Play Store (as of January 2014). The apps belong to the 27 different

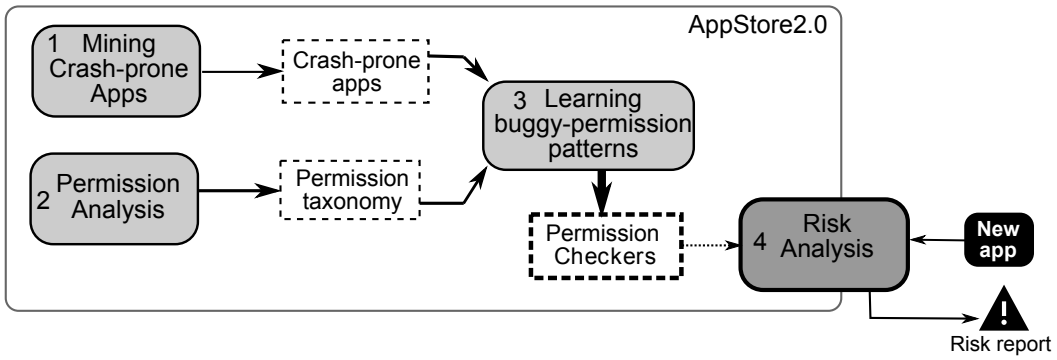


Fig. 5.2 Overview of the `Risk Analysis` component

categories[2] defined by Google, and the 4 predefined subcategories (free, paid, new free, and new paid). For each category-subcategory pair (`tools-free`, `tools-paid`, `sports-new_free`, etc.), we collect a maximum of 500 samples[3], resulting in a median number of apps per category of 1,978. For each app, we retrieve the following metadata: name, package, creator, version code, version name, number of downloads, size, upload date, star rating, star counting, and the set of permission requests. The resulting dataset contains contains 38,781 apps requesting 7,826 different permissions.

In addition, for each app, we collect up to a maximum of the latest 500 reviews posted by users in the Google Play Store. For each review, we retrieve its metadata: *title*, *description*, *device*, and *version* of the app. None of these fields are mandatory, thus several reviews lack some of these details. As app updates might fix bugs from previous releases, some errors reported in the version 1.1 of a given app may be fixed in the new release 1.2. Therefore, from all the reviews attached to an app, we only consider the reviews associated with the latest version of the app —*i.e.*, we discard unversioned and old-versioned reviews. Thus, resulting in a corpus of 1,402,717 reviews.

In our dataset, we identify 10,658 error-suspicious apps (27.48%) by following the process described in Chapter 4 (cf. Section 4.2.1) to identify crash-prone apps from user reviews. Figure 5.3 shows the number of error-suspicious apps identified in each category of the Google Play Store. For each category, we illustrate the distribution of error-suspicious apps according to the number of error-related reviews published by users. With 68.90% of apps having error-related reviews, the category `GAME` contains

[2]Books&Reference, Business, Comics, Communication, Education, Entertainment, Finance, Games, Health&Fitness, Libraries&Demo, Lifestyle, Live Wallpaper, Media&Video, Medical, Music&Audio, News&Magazines, Personalization, Photography, Productivity, Shopping, Social, Sports, Tools, Transportation, Travel&Local, Weather, Widgets.

[3]This number is a constraint enforced by the Google Play Store.

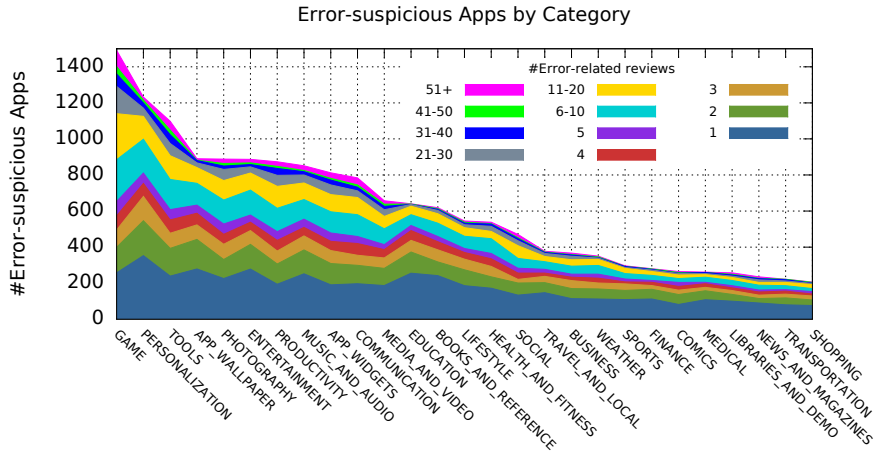


Fig. 5.3 Distribution of error-suspicious apps by categories in Google Play.

the largest number of error-suspicious apps. These data are in line with the results of the study performed by *Crittercism* based on observations in real devices, showing that gaming apps have the highest crash rate [59].

5.1.2 Analyzing App Permissions

To find correlations between permission and crash-proneness, we start by proposing a taxonomy to classify the types of permissions that Android apps can request. This classification will later support the identification of error-sensitive permissions to build app permission checkers.

5.1.2.1 Background on Android Permissions

Android apps run in an isolated area of the system without access to system resources by default. Android provides a set of APIs to enable apps to access to data, resources, and privileged operations. The APIs are protected by a permission model. Apps must explicitly request the required permissions to access protected resources and third-party libraries. Every app has a manifest file (`AndroidManifest.xml`) that summarizes information about the app. In particular, the manifest declares the *permissions* that the app requires for its execution, and a *package name* that serves as unique identifier for the app. When an app attempts to access a resource protected by a permission, the system checks the content of the manifest at runtime.

App developers are solely responsible for identifying the set of permissions required by their apps. Sometimes, developers fail assigning the adequate permissions, often because the available documentation of the Android permission system is incomplete [44, 65, 149]. Indeed, a permission error often results in an exception (*e.g.*, `SecurityException`) being thrown back to the app during execution [Android-System Permissions], often leading to the app crash. In addition, the use of buggy or obsolete APIs can lead to crashes in the apps.

As a matter of clarification, we distinguish between two terms used along the chapter:

- *permission requests* refer to permissions declared by apps in their manifest file;
- *permission types* group available permissions that any app can request.

5.1.2.2 Taxonomy Creation

To create the taxonomy of permission types in Android apps, we follow a grounded theory approach [148]. We first review the literature and define a draft taxonomy based on observations made by prior works [47, 65]. As a matter of example, Felt et al. [65] state that they do not consider neither *developer-defined permissions* nor *Google-defined* ones in their analysis. In addition, they reveal the existence of a set of *non-official* Android APIs. We include those types of permissions as categories in the taxonomy and we investigate them in depth. Another example is Barrera *et al.* [47], who manually observed that some apps in their dataset request *non-existent* permissions and *deprecated* permissions. We also consider information provided in the official Android documentation for developers[4] and in Android community forums (*e.g.*, Android Developers Blog[5]).

5.1.2.3 The Taxonomy of Permission Types in Android Apps

Figure 5.4 shows the resulting taxonomy of permission types.

The taxonomy classifies permissions into four main *categories*:

[4]http://developer.android.com
[5]http://android-developers.blogspot.com

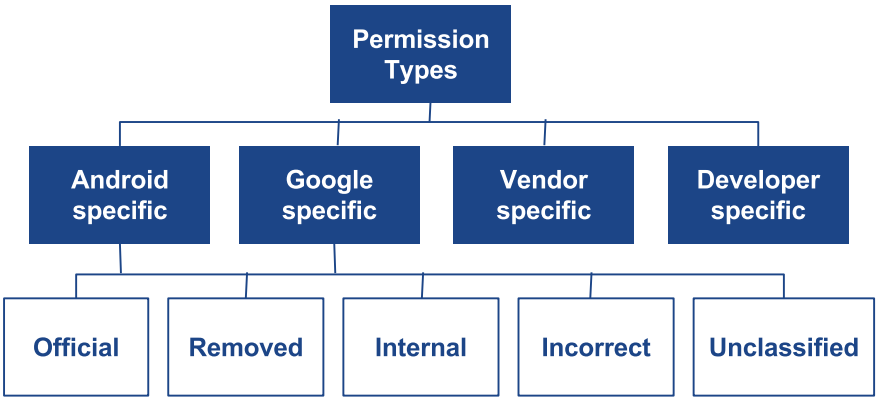


Fig. 5.4 Taxonomy of Permission Types in Android apps

1. *Android-specific permissions* refers to permission types to use the official APIs provided by the Android SDK[6]. For example, `android.permission.READ_CONTACTS` allows an app to read the user's contacts data.

2. *Google-specific permissions* groups the permission types to use the APIs provided by the Google Play Services SDK[7], which include services such as: *Google Play Licensing, Google Play In-app Billing, Google Maps, Google Cloud Messaging (GCM)*, etc. For example, the permission `com.android.vending.BILLING` is required to use the In-app Billing service that enables to sell products from inside an app.

3. *Vendor-specific permissions* encloses permission types to use the APIs included in SDKs provided by specific mobile device vendors (*e.g.*, Samsung, HTC) to create apps specialized for their devices. As an illustration, Samsung provides the *AllShare Framework SDK* that includes APIs[8] to implement convergence services (*e.g.*, media sharing, screen sharing). Apps using AllShare APIs should request the following permission: `com.sec.android.permission.PERSONAL_MEDIA`[9]. Another example is the permission `com.sonymobile.permission.CAMERA_ADDON` that is part of the *Sony Add-on SDK*[10] (Camera Add-on API) required to develop apps that can be launched from the native Xperia camera.

4. *Developer-specific permissions* finally refers to permission types defined by third-party developers. Android enables apps to define their own permissions to

[6]http://developer.android.com/sdk
[7]http://developer.android.com/google
[8]Samsung documentation for developers: http://developer.samsung.com/develop
[9]"sec" comes from: *Samsung Electronics Co*
[10]Sony documentation for developers: http://developer.sonymobile.com

protect the functionality they expose to other apps. For example, `android.webkit.permission.PLUGIN` from *Adobe Flash plugin*.

We group permissions within each category around 5 *classes* of permissions:

(a) *Official permissions* groups permissions that are available in the public SDK's documentations for developers.

(b) *Removed permissions* refers to permissions that were available in previous versions of APIs and do not exist anymore.

(c) *Internal permissions* encloses permissions that are intended for internal use by the system and system apps.

(d) *Incorrect permissions* groups permissions defined erroneously by developers (*e.g.*, misspelled permissions).

(e) *Unclassified permissions* groups permissions that do not fit into any of the previous classes.

5.1.2.4 Taxonomy Exploitation

To automatically catalog permissions under categories, we propose heuristics based on regular expression matching and string analysis:

Heuristic 1 (Android-specific permissions): First, to identify permissions belonging to the Android-specific category, we filter permissions which follow the patterns:

`android.permission.[*]`
`com.android.[*].permission.[*]`

From these permissions, we tag as *official* the permissions that match the list provided in the public Android documentation for developers according to Android 4.4 (API level 19)[11].

Heuristic 2 (Google-specific permissions): To identify permissions belonging to the Google-specific category, we filter permissions which follow the patterns:

`com.google.android.[*].permission.[*]`
`com.android.[*]`

[11]http://developer.android.com/reference/android/Manifest.permission.html

The last pattern excludes the word *"permission"* as it matches the Android-specific pattern. Examples of Google-specific permissions that follow this pattern are:

```
com.android.vending.BILLING
com.android.vending.CHECK_LICENSE
```

to use the billing and licensing services, respectively. From these permissions, we define the class of *internal Google permissions* with the permissions defined by Google apps (*e.g.*, Voice Search, GMail). These permissions follow the patterns:

```
com.google.android.apps.[GoogleAppName].[*]
com.google.android.googleapps.permission.[*]
```

For example, `com.google.android.voicesearch.SHORTCUTS_ACCESS` in the *Voice Search* app.

Heuristic 3 (Vendor-specific permissions): To identify permissions belonging to the Vendor-specific category, we filter permissions following the patterns:

```
com.[vendor].[*].permission.[*]
android.permission.[vendor].[*]
```

We compile a predefined list of available vendor names: sonymobile, htc, huawei, sec (Samsung Electronics Co), dell, and motorola.

Heuristic 4 (Classes refinement): We build lists of permissions to refine the permissions contained in each category under specific classes: official, removed, and internal. These lists are manually created by collecting documentation available in different sources. As a matter of example, the Android SDK contains a set of internal APIs that are intended for exclusive use by the system and system apps. However, they are available from the Android source code and third-party apps can access internal APIs using Java reflection [65]. The internal APIs reside in the Android source code in the package `com.android.internal`, and in the other packages with the annotation `@hide`.

Heuristic 5 (Identifying incorrect permissions): We select all the permissions within a category that are considered as unclassifiable in previous classes. To automatically identify incorrect permissions, we compute the *Damerau-Levenshtein distance* [61] to measure the similarity between two input strings. We normalize the distance to the

range $[0,1]$ by dividing the distance by the length of the longest string, thus defined as:

$$dist_{NDL}(s1,s2) = \frac{dist_{DL}(s1,s2)}{\max(|s1|,|s2|)} \tag{5.1}$$

We compute the normalized Damerau-Levenshtein distance between each unclassified permission P and each official permission O_i. Finally, we set the similarity score of an unclassified permission as the minimum distance of all the obtained normalized Damerau-Levenshtein distances:

$$score(P) = \min\{dist_{NDL}(P,O_i)\}, i = \{0...n\} \tag{5.2}$$

Considering different similarity ranges, we observed 5 types of permission request mistakes in the Google Play Store:

Misspell. For example, requesting `android.permission.READ_CONCACTS` instead of `android.permission.READ_CONTACTS`.

Wrong prefix. Each permission is identified by a unique label. Typically, the label is defined by a string starting with a prefix (*e.g.*, `android.permission.`) followed by a constant in capital letters. We have identified a set of permissions defining incorrect prefixes, specifically two common mistakes:

Prefix absence. For example, `INTERNET` is missing the prefix `android.permission`. The rational for this mistake seems to come from the official Android documentation on permissions, which first provides a table with the permissions showing only the constant part.

Prefix confusion. For example, requests to `android.permission.SET_ALARM`, instead of `com.android.alarm.permission.SET_ALARM`.

Misuse. It is a special case of prefix misuse. Apps request as permissions other Android elements (*e.g.*, libraries, features). For example: `android.hardware.CAMERA`.

Lacking. Finally, we also observed some permissions that look like being incomplete: `android.permission.`

In addition to these mistakes, we build a cluster of *unclassified* permissions: all the permissions within a category that do not fit in any of the defined classes. Finally, the *developer-specific permissions* category covers permissions that do not fit into any of the previous categories.

Table 5.1 synthesizes the results of measuring the abundance of each class of the permission taxonomy in our dataset from Google Play Store. For each category and class of the taxonomy, we show the number of permissions and permission requests. We observed that official Android-specific permissions are the most commonly requested ones among apps. Removed permissions is the second most popular class among the dataset.

	Official		**Removed**		**Internal**		**Incorrect**		**Unclassified**	
	#Perm	#Req	#Perm	#Req	#Perm	#Req	#Perm	#Req	#Perm	#Req
Android	137	245,274	19	1,699	50	1,040	144	561	152	1,240
Google	5	18,246	837	1,107	51	200	39	61	3	22
Vendor	184	1,514	-	-	-	-	-	-	-	-
Samsung	88	302	-	-	-	-	-	-	-	-
Sony	38	291	-	-	-	-	-	-	-	-
HTC	28	251	-	-	-	-	-	-	-	-
Dell	14	14	-	-	-	-	-	-	-	-
Motorola	12	649	-	-	-	-	-	-	-	-
Huawei	4	7	-	-	-	-	-	-	-	-
Developer	1,085	2,875	-	-	-	-	-	-	-	-
TOTAL	**1,411**	**267,909**	**856**	**2,806**	**101**	**1,240**	**183**	**622**	**155**	**1,262**

Table 5.1 Summary of applying the permission taxonomy to our Google Play Store dataset

We manually validated the resulting permission classification. We took a random sample of 300 permissions. Our manual inspection found 5.33% of false positives. Some false positives derive from the interpretation of permissions. For example, the permission `android.permission.SEND_MMS`[12] was tagged as *Misspelled Android* permission while it is obviously not. The nearest match is the permission `android.permission.SEND_SMS`. However, we consider the developer is trying to access to a functionality related with MMS instead of SMS, and it is not a misspelling.

We define as *error-sensitive permissions* those permissions that are suspected to induce bugs. We consider as *error-sensitive permissions* the set of permissions belonging to the classes *removed*, *internal*, *incorrect* and *unclassified*, since the use of *non-official* permissions can be the source of problems. To clarify, not all the bugs are related to the suspicious permissions themselves (*e.g.*, missing permission, wrong permission due to a typo), but rather to the fact that some APIs for which the permission is requested are buggy or obsolete [104]. We study the correlation between permission requests and bugginess without claiming the underlying causes of bugs.

[12]This permission does not exist in the official Android documentation.

5.1.3 Generating Risk Reports

We now study potential correlations between apps that use error-sensitive permissions and those reported as crash-prone by end-users. We use the knowledge inferred from this study to propose crowd-based checkers that could be embedded in app stores to anticipate the emergence of app crashes in the wild.

With the objective of mining common permissions in error-suspicious apps, we use supervised machine learning, and specifically a classifier where independent variables are permissions and the dependent variable is crash-proneness.

5.1.3.1 Mining Crash-prone Permission Patterns

To identify permission patterns that correlate with bugs, we use a predictive machine-learning model, in particular the *J48 Decision Tree* algorithm (a Weka[13] implementation of C4.5 [130]). J48 predicts the value of a dependent variable based on the value of various attributes (independent variables) of the data. In our setup, the independent variables are permissions, and the dependent variable is crash-proneness. We choose J48 because it enables the direct extraction of rules to predict a label. From the resulting decision tree model, we extract permission patterns that lead to the label *'Buggy'*. We only consider presence of permissions in the patterns. However the absence of a permission also causes the app to crash. This kind of bug is partially handled by our approach, when the permission is missing because of an incorrect or incomplete declared permission. Furthermore, there already exists some developer support [48] to automatically check if apps declare all the permissions required to run.

Step 1: Dataset Preprocessing. Let D be a dataset of apps, we represent each app (A) as a binary vector: $A = [p_1, ..., p_n, c_1, ..., c_n, L]$, where $p_i \in \{0,1\}$, $c_i \in \{0,1\}$, and $L \in \{Buggy, NonBuggy\}$. Each p_i depicts a permission, c_i represents a class of permission according to our taxonomy (*e.g.*, `INCORRECT` class), and L represents the class label learnt from its user reviews (*i.e.* `'Buggy'` or `'NotBuggy'`). The value $p_i = 1$ indicates that the app requests the permission p_i, and p_i=0 indicates the absence of the permission. Similarly, $c_i = 1$ indicates that the app requests a permission that belongs to the class c_i in our taxonomy.

[13]Weka is an open-source Java library that provides implementations of several data mining algorithms: http://www.cs.waikato.ac.nz/ml/weka

For the purpose of this analysis, we group all the classes of infrequent single permissions (*e.g.*, the misspelled ones) within a common abstract class. Indeed, a specific misspelled permission usually only appears in one app, but we are rather interested in knowing if the group of incorrect permissions is frequent in error-suspicious apps. There are also some permission names that are customized in each app. For example, apps requesting the GCM Google service must include a specific permission for receiving messages: `[appPackage].permission.C2D_MESSAGE`. The permission name must exactly match the pattern, but each app substitutes `[appPackage]` by its own package name in the manifest. This permission prevents other apps from registering and receiving their messages. Therefore, each specific GCM permission is only requested once at maximum. Moreover, we grouped all the official Android permissions in a single dimension[14]. We notice that the 10 most requested official Android permissions in our dataset are the same top requested permissions observed by other studies [Android Observatory, 69][15].

Finally, the 4 classes of permissions considered are: `ANDROID-OFFICIAL, INCORRECT, GOOGLE-GCM,` and `GOOGLE-REMOVED` permissions. Each app is represented by a 1,563-dimensions vector, where the first $1,558$ dimensions represent single permissions (p_i), the following 4 dimensions refer to classes of permissions (c_i), and the last dimension is the assigned class label (L).

Step 2: Model Construction. The J48 algorithm takes as input parameters a *confidence factor* and a *minimum number of objects* ($minNumObj$). The $confidencefactor$ limits the prediction error. For example, a confidence factor of 0.25 indicates that a permission pattern fails as maximum in the 25% of predictions. Thus, the lower the confidence factor, the more accurate the classifier. The $minNumObj$ parameter sets the minimum number of instances that reach a leaf of the tree model. In our case it represents the minimum number of apps that must exhibit a permission pattern to be considered as a predictor of the class.

We train a J48 decision tree model using the dataset which contains $10,658$ apps labeled as `Buggy` and $11,539$ labeled as `NonBuggy` (cf. Section 5.1.1). Varying the $confidence$ and $minNumObj$ thresholds impacts the number of permission checkers

[14]The request of Android permissions should not make the app crash, and we are rather interested in revealing the existence of patterns in error-sensitive permissions. First, we run the experiment considering all the official Android single permissions, but the most requested permissions appeared in many checkers, leading to high amount of irrelevant patterns.

[15]The most requested permissions in our dataset are: `INTERNET, ACCESS_NETWORK_STATE, READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, READ_PHONE_STATE, ACCESS_WIFI_STATE, WAKE_LOCK, ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, and VIBRATE`

obtained and their respective performance. We run the algorithm several times for different input values in order to identify the best calibration. We set cross-validation 10 folds. Then, we train the model with different confidence factors (ranging from 0.05 to 0.50 by increments of 0.05) with three different *minNumObj* limits—*i.e.*, 100, 50 and 20 apps. Figure 5.5 reports on the results of the sensitivity analysis performed. The resulting family of checkers ranges from 4 to 12 different permission checkers.

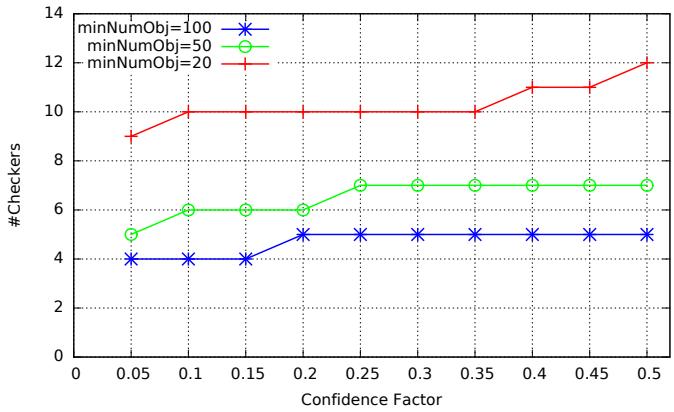


Fig. 5.5 Calibration of *confidence factor* and *minNumObj* parameters.

Table 5.2 details 4 families of permission checkers (F1–F4) obtained varying the $confidence$ factor (c) and $minNumObj$ (m) values. For example, the family $F2$ includes the 4 permissions pointed by $F1$ and one additional permission. In the family $F1$, we observe two official Google permissions: `CHECK_LICENSE` and `BILLING`. Contrary to our expectations, these official Google permissions are involved in some bugs. After searching in online forums for Android developers, we realize that there are many Android developers complaining because they have experienced crashes (due to security exceptions) in their apps after the update of Google Play services 4.3 (March 2014).

Note that the buggy permission checkers are not meant to be permanent. The app ecosystem is continuously evolving, and app store moderators can use our approach regularly (say weekly) for updating existing checkers, discovering new ones, and discarding outdated ones. The proposed system reveals interesting insights for isolating bugs in real devices. Nevertheless, we do not claim causality, but rather we suggest permissions that correlate with bugs.

The family of permission checkers obtained forms the knowledge of the recommender system. Taking into consideration the performance of the different families obtained, thus app store moderators can make informed decisions on which checkers to enable

Table 5.2 App Permission Checkers

	c	m		Error-sensitive Permission Checker
F1	0.1	100		android.permission.WRITE_INTERNAL_STORAGE android.permission.ACCESS_SUPERUSER com.android.vending.CHECK_LICENSE com.android.vending.BILLING
F2	0.2	100	F1	com.android.launcher.permission.READ_SETTINGS
F3	0.2	50	F1	android.permission.WRITE_OWNER_DATA com.sonyericsson.extras.liveware.aef.EXTENSION_PERMISSION
F4	0.2	20	F3	com.google.android.googleapps.permission.GOOGLE_AUTH android.permission.STORAGE com.android.email.permission.READ_ATTACHMENT com.google.android.gm.permission.READ_CONTENT_PROVIDER

regarding their performance, in order to predict potential buggy-apps before they are published in the store.

5.1.4 Implementation Details

This section provides details of the implementation that supports the generation of the crowd-based permission checkers.

To generate the checkers we started by collecting a dataset of apps from Google Play Store. To store the dataset, we created a *graph database* with *Neo4J*[16]. This dataset therefore consists of a graph describing the apps as nodes and edges. Graph databases provide a powerful and scalable data modelling and querying technique capable of representing any kind of data in a highly accessible way [134]. We chose a graph database because the graph visualization helps to identify connections among data (*e.g.*, clusters of apps sharing similar sets of permission requests). In particular, our dataset graph contains five types of nodes: `APP` nodes grouping the details of each app, `PERMISSION` nodes describing permission types, `CATEGORY` nodes describing app categories, `SUBCATEGORY` nodes describing app subcategories, and `REVIEW` nodes storing user reviews. Furthermore, there are four types of relationships between `APP` nodes and each of the remaining nodes *e.g.*, `USES_PERMISSION` relationships between APP and PERMISSION nodes. In total, our graph contains $1,449,361$ nodes and $1,901,703$ relationships among them. To extract statistics from the dataset, we can query the graph database using one of the available graph query languages (*e.g.*,

[16]http://www.neo4j.org

Cypher, SPARQL, Gremlin). We chose *Cypher* [Cypher], which is a widely used pattern matching language.

Figure 5.6 shows an excerpt of the generate Google Play Store graph database in Neo4j. This dataset is available online[17].

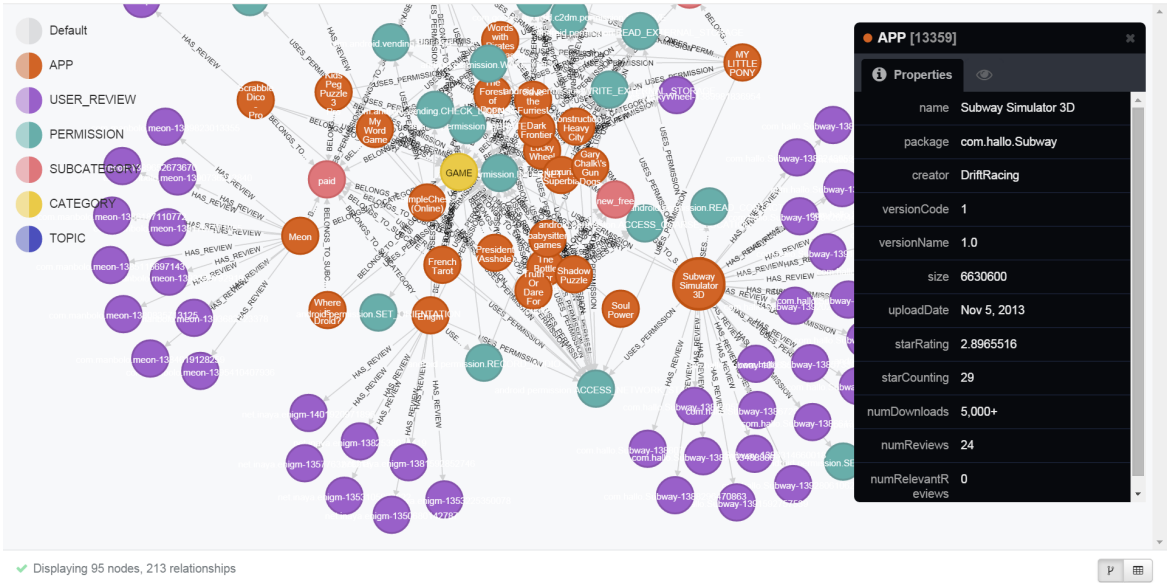


Fig. 5.6 Excerpt of the Google Play Store dataset in Neo4j

To identify apps which accumulate reviews related with errors and crashes we use topic modelling, in particular the *Latent Dirichlet Allocation* algorithm available in the MALLET library [116].

Finally, to learn the permissions patterns which constitute the checkers, we use the *J.48 algorithm* provided in the Weka library.

5.2 Reporting on Performance Degradations

When the app developer uploads a new app release to the store (for example after bug fixing), the App Store 2.0 runs a performance analysis to identify potential UI performance degradations. The `Performance Analysis` component receives as input a *scenario or test* and the *historic run repository* that are associated with the previous app release. The *historic run repository* contains performance metrics collected during the execution of the *scenario or test* in previous versions of an app.

[17]https://sites.google.com/site/androidbuggyappcheckers/

To assess whether the new app release fulfills the expected performance goals—*i.e.*, to prevent performance degradations and to improve performance on low-end devices—the system first repeats the scenario with the new app release on different devices of the crowd while collecting UI performance metrics. Then, it compares the newly collected metrics with the previous metrics available in the *historic run repository* to automatically flag performance deviations.

We propose an approach called DUNE[18] to detect *UI performance degradations* among different versions of an app and heterogeneous contexts. The `Performance Analysis` component implements the DUNE approach. In this section, we describe the DUNE approach in detail. Before jumping into the details of the approach, we discuss three major challenges associated with the current practices to identify *UI performance regressions* in mobile apps:

Challenge #1: Automatic detection of UI performance degradations. Android provides tools to profile the GPU rendering of apps in order to identify janky frames in an app [GPU]. The current UI performance testing consists of having a human tester which performs a set of user operations on the target app and either visually look for janks, or spend a large amount of time using the GPU profiler to identify janky frames. Hence, this *ad hoc* analysis is time consuming, tedious, and error prone. In addition, such a process relies on the human tester's ability to perceive frame rate changes, which could vary from person to person.

Challenge #2: Triaging UI performance root causes. Once a jank is identified, developers rely on other tools, like Android Systrace [Sys], to profile concrete app executions in order to find the source of the jank. In particular, they need to filter the full data collected from a test execution in order to identify the specific sequence of user events that triggered the slow-down. This manual investigation again requires trial-and-error, which is even harder as the size of the app increases.

Challenge #3: Environment heterogeneity. The app performance varies depending on the device on which it runs. For example, an app can perform well on one set of devices, or one combination of hardware and software, but it can present performance bottlenecks when running in lower-end devices or in a different execution context (*e.g.*, using different network types). There is a lack of tools to identify UI performance deviations between different app versions and contexts.

[18] DUNE stands for *Detecting Ui performaNce bottlEnecks.*

Figure 5.7 shows an overview of the approach to detect performance degradations between app versions and contexts. The approach has three main phases:

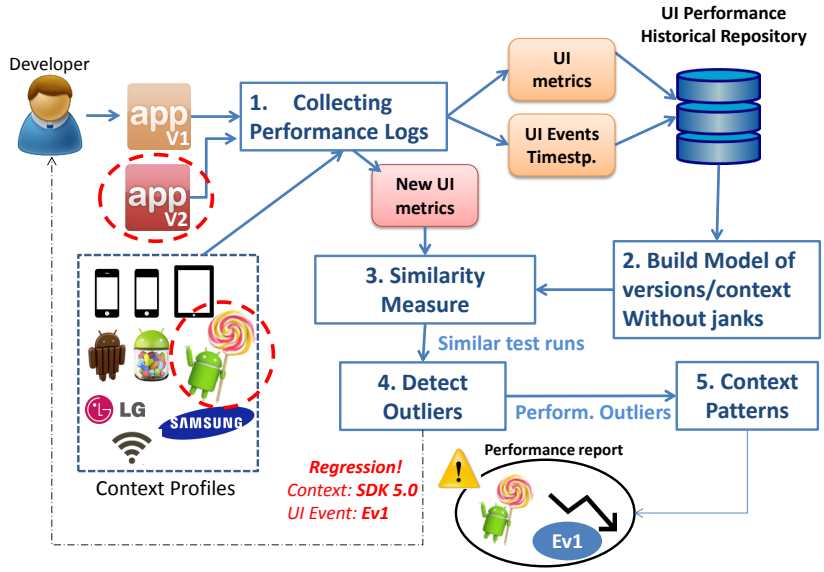


Fig. 5.7 Overview of the *Performance Analysis* module

1. *Aggregating Performance Logs.* To compare a new scenario execution with the historical executions, we first build a model which aggregates all the performance logs corresponding with a previous app release.

2. *Identifying Performance Deviations.* The `Performance Analysis` component calculates context similarity and applies a statistical technique (in particular Interquartile Range) to flag a performance regression when the distance between the metrics of the new and old executions in similar context is larger than a given threshold.

3. *Generating Performance Reports.* If the new app release is flagged as an outlier, the system identifies the device configurations and specific UI events that trigger the performance deviations. Then, a detailed report is sent to the developer who can fix the app before publication.

5.2.1 Aggregating Performance Logs

The *historic run repository* stores performance traces resulting from running a specific *scenario* on different devices (cf. Chapter 4 section 4.2.3.2). First, the `Performance Analysis` builds a *batch model* that aggregates the collected metrics and the execution context across all the runs in the repository for an specific app and scenario.

Figure 5.8 (left) depicts an example of a batch model created from a historic repository with three performance traces (T_1, T_2, T_3) in version $V1$ of an app. Each trace captures its *execution context* (*i.e.*, SDK, device manufacturer, device model, CPU chip, and network type); and the run's *UI performance metrics* (*e.g.*, #frames rendered, #janky frames, and smooth ratio).

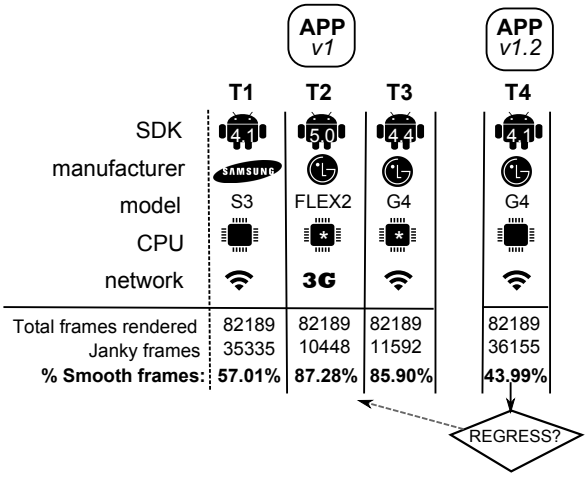


Fig. 5.8 Example of a *Batch Model* from a *historic runs* repository

The *batch model* will be used to detect deviations from the expected app performance. When the developer releases a new version of the app (*e.g.,* version $V1.2$), the batch model is used to flag performance deviations.

5.2.2 Identifying Performance Deviations

When there is a new version of an app or a new execution context (such as a new SDK released by Android), the app developer needs to guarantee that the new release (or the app on a new device) meets the expected performance requirements to fulfill the user expectations. Then the store executes the scenario in the new context and compares the metrics with the historic runs. The `Performance Analysis` component automatically

compares the new performance trace (T_4) with the historical runs repository (T_1,T_2,T_3) and flags potential performance degradations at specific locations in the app. Remark that the comparisons are only done between test runs replicating the same scenario in the app, executing exactly the same functionalities and features.

The process to identify performance degradations consists of 3 steps.

5.2.2.1 Step 1: Calculating Context Similarity

Since the app performance varies depending on the device on which it runs, we rank previous performance traces according to the context similarity with the new trace [68]. Thus, a previous trace which was run on a more similar environment receives more weight and will be more relevant for the comparison than a trace on a completely different environment. To calculate *context similarity* between two traces T_1 and T_4 (cf. Figure 5.8), first we extract from each trace the sub-vector that contains the context dimensions. For example, the context of the traces in Figure 5.8 are represented by the sub-vectors:

$$T_1 = [v1,\ \underline{4.1}, samsung, S3, \underline{x86}, \underline{wifi}\ ]$$
$$T_4 = [v1.2,\ \underline{4.1}, lg, G4, \underline{x86}, \underline{wifi}\ ]$$

Then, we generate a binary *similarity vector* to capture the context properties that are common in the two runs. A value of 1 indicates that the two runs share the context property. Conversely, the value 0 indicates that the context property differs between the two runs. The similarity vector between traces T_1 and T_4 is: $Sim(T_1, T_4)$=[0,1,0,0,1,1], which means that both T_1 and T_4 share SDK version 4.1, CPU chip $x86$ and $wifi$.

Finally, we measure the *similarity degree* ($simD$) between the new and the past trace as the Cartesian length of the similarity vector. The Cartesian length is calculated as $\sqrt{N}$, with N the number of 1s in the similarity vector. For example, the similarity degree between T_1 and T_4 is $\sqrt{3} = 1.73$. The higher the degree, the higher the similarity between the context of the traces.

5.2.2.2 Step 2: Ranking Previous Tests

Using the similarity degree, we can assign a *weight* (w) to each previous trace to describe the importance of that trace to analyze the new scenario run. Since we expect

that an app behaves similar in similar contexts, traces having similar or identical contexts as the new trace, should indeed have the largest weights.

Hence, to rank traces, we follow the approach proposed by Foo et al. [68], which calculates the weight of a previous test as $w(T_i) = \frac{simD(T_i)}{\sum_{j=0}^{n} simD(T_j)}$. Thus, the sum of all weights is 1. Table 5.3 reports on the similarities and weights between the new trace T_4 and the historical performance traces. For example, the weight of T_1 is $w(T_1) = \frac{1.73}{1.73+1+1.73} = 0.39$.

Table 5.3 Context similarities between the new performance trace (T_4) and the historical traces (T_1, T_2 and T_3).

$test_N$	$test_H$	**simD**($Test_N$,$Test_H$)	**w**($Test_H$)
T_4	T_1	1.73	0.39
	T_2	1	0.22
	T_3	1.73	0.39

We use the similarity weight to rank the previous traces, and cluster them according to the similarity degree with the new trace. Each cluster groups the traces that have the same degree (and hence weight). In the example, the traces result in two clusters: the first cluster contains the traces T_1 and T_3, while the second cluster only trace T_2. Since the first cluster contains the set of traces most similar to the new trace, it will be used to identify performance deviations.

5.2.2.3 Step 3: Identifying Performance Deviations

The third step aims to identify performance outliers by comparing the metrics collected in the new trace with the set of traces obtained in the previous step. To detect such outliers, we apply the *Interquartile Range* (IQR) statistical technique. This technique filters outliers and extreme values based on interquartile ranges—*i.e.*, a metric (x) is considered as an outlier if:

$$(1) \quad x < Q1 - OF \times (Q3 - Q1)$$
$$(2) \quad x > Q3 + OF \times (Q3 - Q1)$$

where $Q1$ is the 25% quartile, $Q3$ is the 75% quartile, and OF is the outlier factor, which is an input parameter of the technique. We use $OF = 1.5$ because it is the default value provided by the Weka library (used in our implementation) to identify

outliers with this technique. However, OF is a configuration parameter that can be overridden by the DUNE user. With a lower OF factor, metrics are flagged faster (*i.e.*, with slighter difference) as outliers.

The outliers flagged by *equation (1)* are values that deviate below the normal performance (with negative offset), whereas outliers flagged by *equation (2)* are values above the normal performance (with positive offset). Thus, the `Performance Analysis` component can detect performance deviations that can be categorized as either performance regressions or optimizations with respect to previous performance traces. Note that the interpretation of outliers as regression or optimization depends on the type of the metric x. For example, for the metric *number of frames rendered*, an outlier with negative offset is a performance regression. If the new test renders less frames means that some frames were skipped during the execution, thus resulting in laggy animations perceivable by users.

By default, the outlier detection considers the set of metrics of a scenario execution together (coarse-grained, cf. Section 4.2.3.2), then determines if the resulting trace is an outlier in comparison with the history repository. As illustration, consider a historic repository of an app with five performance traces that recorded the following performance metrics (#Fr, Rat, T):

$$
\begin{array}{rcccc}
 & (\#Fr) & (Rat) & (T) & \\
T_1 = [& 448, & 0.95, & 45.3 &] \\
T_2 = [& 453, & 0.94, & 44.5 &] \\
T_3 = [& 450, & 0.93, & 44.9 &] \\
T_4 = [& 457, & 0.80, & 50.3 &] \\
T_5 = [& 430, & 0.85, & 45.7 &]
\end{array}
$$

Recently, Android has released a new SDK version (Android Marshmallow 6.0.0) and the app developer wants to ensure that his app has a good performance in this new context. Thus, he runs the same scenario in a new device with Android 6.0.0, which records the following metrics:

$$T_6 = [\ 270,\ 0.40,\ 40.3\ ]$$

To automatically detect any performance deviation, we apply the IQR filtering to the data available in the history repository (T_1, T_2, T_3, T_4, and T_5) and the new performance trace (T_6). If we assume all 5 traces in the repository to be clustered together based on similarity degree, a new trace is flagged as outlier if at least one of its metrics is flagged as outlier compared to these 5 runs.

For example, the first metric of T_6 ($x_1 = 270$) is an outlier in comparison with previous observations of such metric: {448, 453, 450, 457, 430, 270}, according to *equation (1)*:

$$Q1 = 390, \quad Q3 = 454$$
$$270 \ < 390 \ - 1.5(64)$$

As a result, T_6 is flagged as an outlier.

Finally, the `Performance Analysis` component adds a class label ($Outlier_+/Outlier_-/N$) to the analyzed test to indicate if it is a performance optimization, regression or normal execution.

Furthermore, if the *fine-grained* mode is activated, DUNE can spot specific UI events associated with the outliers. In this case, the performance trace contains a list of sets of metrics, in particular as many sets of metrics as UI events in the scenario (cf. Figure 5.9 top). The outlier detection process follows the same IQR filtering strategy, but in this case, the system flags individual UI events as outlier.

Figure 5.9 illustrates the automated outlier detection process in the *fine-grained* mode. The historic repository contains five historical traces (T_1 to T_5) and a new trace T_6 to analyze. We apply the IQR technique to identify outliers per attribute. As a result, the store flags T_6 as an outlier and reports the event $E2$ as the location of the outlier. Then DUNE adds the class label $E2Outlier_-$ to the trace T_6, to indicate that the event $E2$ exhibits a performance degradation. If several UI events are flagged as outliers, then several class labels are added to the trace.

5.2.3 Generating Performance Reports

If the set of metrics of the new run are flagged as an outlier, the store generates a performance report providing information about the violated metric and the location of the offending UI event.

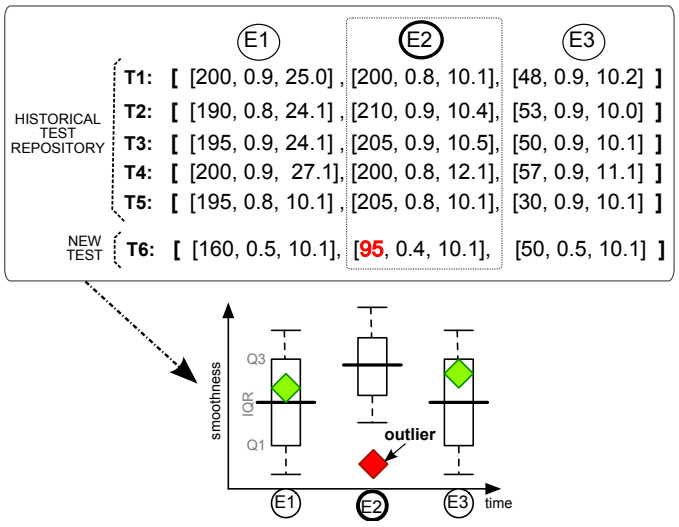


Fig. 5.9 Automated detection of performance deviations through Interquartile Ranges with fine-grained (UI) metrics.

5.2.3.1 Frequent Context Mining

In the presence of performance outliers, the `Performance Analysis` component characterizes the context under which the conflicts arise. To identify common context patterns that induce outliers, we use *Association Rule Mining*. An association rule is an implication of the form $X \implies Y$ that states "when X occurs, Y occurs" with a certain probability. The support of such a rule is the frequency of the rule in the dataset, while the confidence of a rule indicates the percentage of instances of X for which Y occurs. In our case, X is a context property, and Y is the class label (*e.g.*, $Outlier\_$) learned in the previous step. To build a context rule, we require a confidence value of 90%, which is the default parameter value provided by Weka. However, this configuration parameter can be overridden by the users of DUNE.

Some examples of rules are:
$\{sdk = 4.1\} \implies Outlier\_$
$\{v = 1.2, dev = LG\} \implies E2Outlier\_$

These context rules help the developer to narrow down the different contexts under which performance bottlenecks appear.

Finally, a performance report is sent to the app developer who can fix the app before making the app publicly available to users.

5.2.4 Implementation Details

This section provides details about our proof-of-concept implementation of Dune. Dune consists of two parts. The first component is in charge of executing tests on devices and collecting metrics during the execution, while the second component is in charge of detecting performance deviations.

Figure 5.10 shows an overview of our tool implementation in charge of collecting performance metrics during execution.

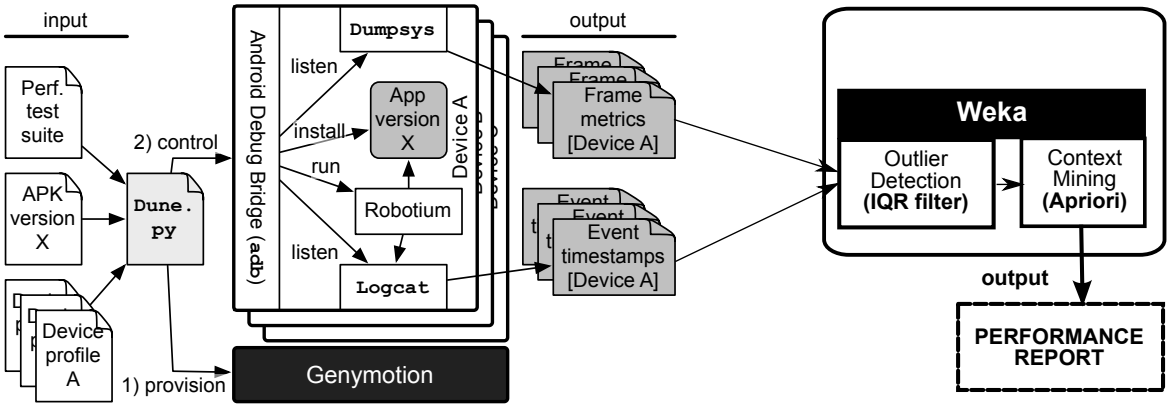


Fig. 5.10 Dune implementation. Metrics acquisition component (left). Report generation component (right).

This component is implemented as a Python script that installs and runs tests on mobile devices and collects metrics during the test executions. To communicate with devices, our implementation relies on the *Android Debug Bridge* (`adb`) [adb]. `adb` is a command line tool included in the Android SDK that acts as a middleman between a host and an Android device. We use `adb` to install/uninstall apps and tests, log UI events, and profile GPU rendering.

To track UI events, the tests write the timestamps of the events in the *Android Logging system* (`logcat`) [log]. In Android, the system collects debug information from apps and from the system into logs, which can be viewed and filtered by `logcat`. We have implemented a listener that monitors `logcat` and subscribes to the UI event messages.

To profile GPU rendering, we use the Android `dumpsys` tool, which runs on a device and dumps relevant information about the status of system services [dum]. Since `dumpsys` only provides information about the last 120 frames rendered, our tool read the frames rendered every second, to ensure that no frame information is missed. To

ensure a low overhead, this component is built on top of low-overhead tools provided by Android.

Instead of requiring tests to be run on physical devices, we leverage the fast third-party Android emulator Genymotion [gen], which allows to generate emulators with specific context configurations, on-the-fly. Following the concepts of Infrastructure-as-a-Service [92], Genymotion just requires a textual specification, after which generation, deployment and execution of the emulators is fully automated. In particular, one can configure the emulated device model, SDK, network, etc. Nevertheless, this component can also work whit physical devices.

The second component of our proof-of-concept implementation is a Java application that interfaces with Weka [84]. Weka is an open-source Java library that implements several data mining algorithms. In particular, we use the *InterquartileRange* filter implemented in Weka to identify outliers, and the *Apriori* algorithm to mine association rules to point out offending contexts.

5.3 Conclusions

This chapter presents two engineering approaches to automatically generate two different types of actionable insights (*risk reports* and *performance reports*) by analyzing three types of crowdsourced artefacts (*user reviews*, *execution logs*, and *app context*) accessible from app stores. The actionable insights aim to assist app developers and store moderators to anticipate the rise of issues in apps before making the apps publicly available to users. As a result, the quality of apps increases, then the user experience and satisfaction with apps.

Chapter 6

Leveraging the Crowd *in Vivo*

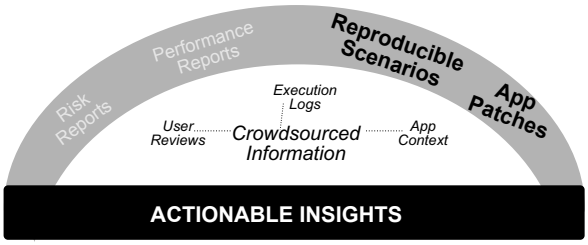


Fig. 6.1 Chapter 6 at a glance

After the release phase (cf. Chapter 5), the store makes apps visible to users. Once apps are installed and executed by users, the App Store 2.0 initiates the *in vivo* measures.

This chapter presents two types of actionable insights that app stores can deliver *in vivo*: *reproducible scenarios* and *app patches*. We present two modules (`Crash Analysis` and `Patch Generation`) to generate such insights respectively. The *reproducible scenarios* aid developers to automatically reproduce crashes experienced by users. The *app patches* prevent that users face a previously observed crash.

The rest of the chapter is organized as follows. Section 6.1 describes the approach to identify recurrent crash patterns and generate *reproducible scenarios* to support developers in the crash reproduction task. Section 6.2 presents the module to generate hot patches to mute crashes. Section 6.3 concludes the chapter.

6.1 Reproducing Crash Scenarios a posteriori

Any software developer knows that faithfully reproducing crashes experienced by users is a major challenge. From previous research we know that a single failure report contains insufficient information to isolate a bug [166]. To assist developers in isolating

and reproducing crashes in an automatic and effective manner, the APP STORE 2.0 incorporates a `Crash Analysis` component. We propose an approach called MOTIF to translate a collection of crash logs into a *reproducible scenario*. The `Crash Analysis` implements the MOTIF approach to support developers to reproduce crashes experienced by users.

Figure 6.2 summarizes the MOTIF approach. Figure 6.2 (left side) shows three crowdsourced crash logs collected from three different users when using the Wikipedia app. For example, there is a log that reports a crash after the user performs the following sequence of actions: *A-'Click on Menu'*, and *B-'Click on Save Page'*. At event *A*, the network was *ON*. Whereas the network was *OFF* at event *B*. We described the monitoring process to collect crash logs in detail in Chapter 4 (cf. Section 4.2.3).

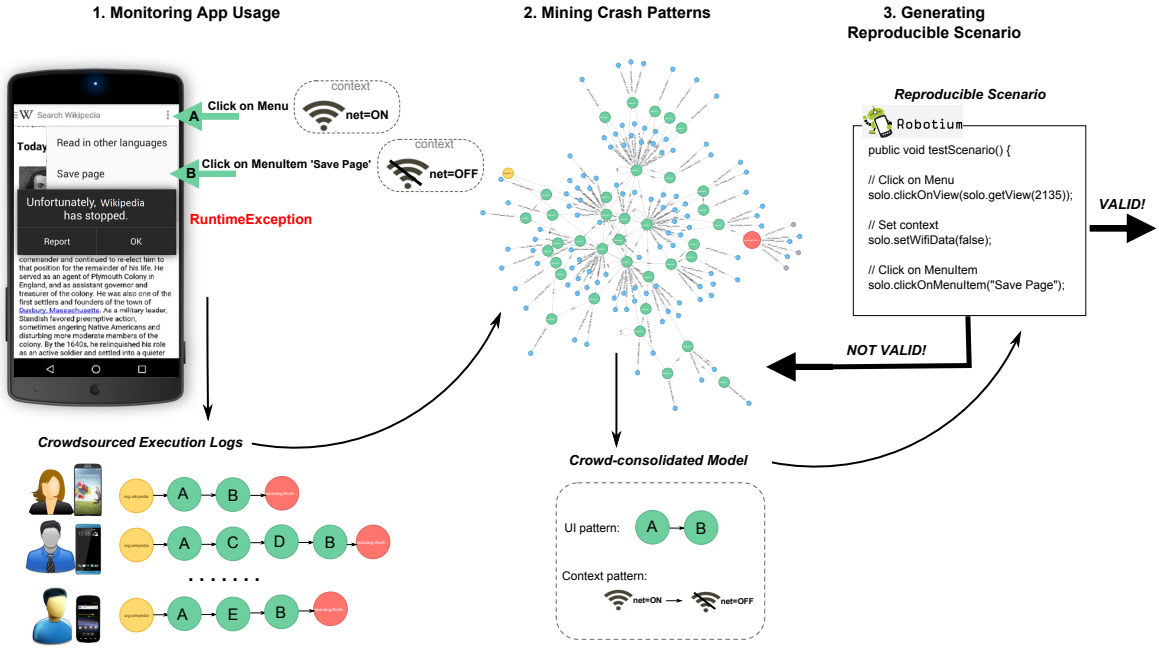


Fig. 6.2 Overview of the `Crash Analysis` module in the APP STORE 2.0.

The approach to generate *reproducible scenarios* from crowdsourced crash logs has three main phases:

1. *Aggregating crowdsourced crash logs.* The first step consists of aggregating the set of crash logs collected from a multitude of devices in a compact representation.

2. *Identifying crash patterns.* First, we identify common *crash patterns* among the collection of app execution traces observed in the wild. These patterns will be used to automatically extract the minimum sequence of steps to reproduce

crashes and characterize the operating conditions (*i.e.*, execution context) under which failures arise (cf. Section 6.1.2).

3. *Generating reproducible scenarios.* The crash and context patterns identified in step 2 are translated into a *reproducible scenario* to reproduce crashes experienced by users. This scenario will automatically replay a sequence of user interactions that led to a crash of the app (cf. Section 6.1.3). The store uses the crowd of devices to validate the generated scenarios. Once the scenario is validated, the store notifies the app developer.

6.1.1 Aggregating Crowdsourced Crash Logs

The `Crash Analysis` component aggregates the crowdsourced crash logs into a weighted directed graph that we denote as *Crowd Crash Graph*. The *Crowd Crash Graph* represents an aggregated view of all the interactions that users performed in a given app before a crash arises, with their frequencies. The graph enables to induce 1) the minimum sequence of steps to recreate a crash; and 2) the context under which crashes arise.

6.1.1.1 Definition: Crowd Crash Graph

The crowd crash graph (CCG) consists of a collection of directed graphs:
CCG=$\{G_1, G_2, ..., G_n\}$, where each G_i is a *crash graph* for a different type of crash. Such a *crash graph* aggregates all crash traces that lead to the same exception. It is based on a Markov chain (1st order Markov model), which is a widely accepted formalism to capture sequential dependencies [123]. In our *crash graph*, nodes represent user interaction events, and edges represent sequential flows between events. Nodes and edges have attributes to describe event metadata and transition probabilities, respectively. The transition probability between two events (e_i, e_j) measures how often, when e_i is fired, it is followed immediately by e_j. In each node, the probability to execute the next event only depends on the current state, and does not take into consideration previous events.

Our crash graphs are based on the idea of Kim et. al. [100] to aggregate multiple crashes together in a graph. However, our crash graphs capture a different kind of information. Whereas the nodes of Kim et al. represent functions and edges represent call relationships between functions (extracted from crash reports); our nodes represent

Table 6.1 Example of *crash traces* and single steps split. In bracket, occurrences of each step.

Crash traces	**Single trace steps**
e1→e2→crash1	e1→e2(2) e2→crash1(3)
e1→e4→e5→e2→crash1	e1→e4(1) e4→e5(1)
e3→e1→e2→crash1	e5→e2(1) e3→e1(1)
e1→e5→crash2	e1→e5(1) e5→crash2(2)
e1→e6→e5→crash2	e1→e6(1) e6→e5(1)

events, and our edges represent sequential user interaction flow. Our nodes and edges also store event and context metadata, and the graph forms a Markov model. In addition, we use crash graphs with a different purpose: to synthesize the most likely sequence of steps to reproduce a crash.

6.1.1.2 Building the Crowd Crash Graph

As illustration, we consider a version of the Wikipedia app (v2.0-alpha) that contained a crash-inducing bug—*i.e.*, the app crashes when the user tries to save a page and the network connection is unavailable. Table 6.1 (left) shows an example of five traces generated with the subject app.

Given a set of traces collected from a multitude of devices, the `Crash Analysis` component first aggregates the traces in a graph. The process to build the Crowd Crash Graph comprises the following two steps:

Step 1: Clustering traces by type of failure. First, we cluster the traces leading to the same exception. To identify similar exceptions, different heuristics can be implemented. For example, Dang et al. [62] propose a method for clustering crash reports based on call stack similarity. We use a heuristic that considers two exceptions to be the same if they have the same *type* (*e.g.*, `java.lang.NullPointerException`) and message, and they are thrown from the same location—*i.e.*, same *class* and *line* number. For example, in Table 6.1 (first column), we identify two clusters of traces. The first cluster contains three traces leading to $crash1$, and the second cluster contains two traces leading to $crash2$.

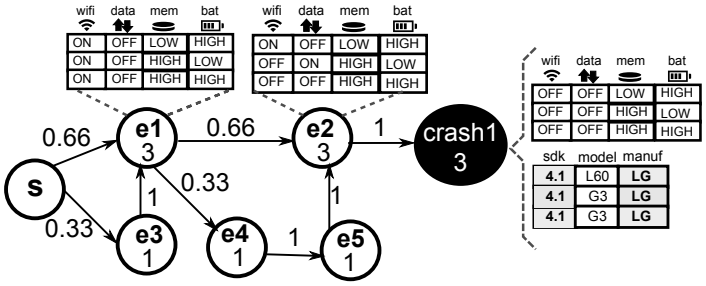


Fig. 6.3 Crash Graph derived from Table 6.1.

Step 2: Merging traces in a crash graph. Next, for each cluster of traces, we form a *crash graph* following the graph construction technique proposed by Kim et al. [100]. First, we decompose each trace into single steps—*i.e.*, pairs of events executed in sequence in a trace (cf. Table 6.1 right). The trace $e_1 \rightarrow e_2 \rightarrow crash1$ contains two steps: $e_1 \rightarrow e_2$ and $e_2 \rightarrow crash1$.

Then, for each event in a step, we create a node in the graph. If the node already exists, we update its weight. For the same step, we then add a directed edge to connect the step's two events. If the edge already exists, we update its weight. In addition, we create a start node (S) that represents the launch of the app, and add edges to connect the start node with the first event node of each trace. Finally, we add the context metadata associated with each event as attributes to the corresponding event nodes. Figure 6.3 shows the resulting *crash graph* from the cluster of traces leading to $crash1$ in Table 6.1.

For each step in the graph, we then calculate the transition probabilities. For example, after executing event $e1$, the event $e2$ is executed 2 times; and the event $e3$ is executed 1 time. Therefore, the transition probabilities from node $e1$ to $e2$ and $e3$ are: $P_{e1-e2} = 0.66$ and $P_{e1-e3} = 0.33$. We label each edge with the transition probabilities. In addition, each node contains a weight indicating the number of occurrences of the event. The event $e2$ was executed 3 times.

Finally, the set of crash graphs (one for each type of exception) is stored in a graph database to form the *Crowd Crash Graph* of a given app. This model provides a consolidated view of the most frequent actions among users before a crash arises, together with the observed execution contexts.

6.1.2 Identifying Crash Patterns

We assume that the most frequent events are the most relevant ones. Thus, the `Crash Analysis` component traverses the *Crowd Crash Graph* to identify repeating patterns of UI events and contexts that appear frequently among crashes. While several data mining techniques can be used, we use *Path Analysis*, *Sequential Patterns*, and *Set Operations* to induce the minimal sequence of steps that reproduce a crash as well as the context under which this crash occurs.

6.1.2.1 Synthesizing Steps to Reproduce Crashes

MoTiF applies graph traversal algorithms in order to effectively induce the shortest sequence of steps to reproduce a crash. Some of the app execution logs can be long and contain irrelevant events to reproduce the crash. For example, the trace e1→e4→e5→e2→crash1 in Table 6.1 includes four steps to crash the app. However, there is a two-step trace, e1→e2→crash1, that results in the same crash. By exploiting the *Crowd Crash Graph*, we reduce the size of traces and filter out the irrelevant steps.

The goal of this phase is therefore to find the shortest path from the starting node (S) to an exception node (e) that maximizes the Markov probability of the traversal. For this purpose, the `Crash Analysis` component implements the *Dijkstra*'s algorithm [63], which is a widely known algorithm to find the shortest path between nodes in a graph. Whereas *Dijkstra* aims to minimize the weights over the paths, our goal is to find the path that maximizes the transition probabilities over the S-e path. Given that *Dijkstra* does not work with negative weights, we reduce the problem to a standard shortest-path problem by replacing the transition probability (P_i) of each edge by $-\log P_i$. Since log is a monotonic function, maximizing P_i is equivalent to minimizing $\log P_i$. In addition, $P_i \in [0,1] \Rightarrow \log P_i \leq 0 \Rightarrow -\log P_i \geq 0$, hence all weights are positive. For example, in the crash graph of Figure 6.3, we convert the transition probability between the nodes $e1$ and $e2$ as: $P_{e1-e2} = 0.66 \Rightarrow P_{e1-e2}^{Dijkstra} = -\log 0.66 = 0.18$.

Therefore, the shortest S-e path is the maximum probability path, which we call the *consolidated trace* and is promoted as the candidate trace to reproduce the crash. In our example, Dijkstra's algorithm starts at node S and would select node $e1$, since has the minimum weight (0.18), which corresponds to the edge with the highest transition probability (0.66). After $e1$, it selects event $e2$ since it again has the minimum weight

(0.18 or probability of 0.66). Therefore, the *crowd-consolidated trace* to reproduce *crash*1 is *e1→e2→crash1*.

The algorithm can return N different traces ordered by descending probability. If the trace does not reproduce the crash, then the store tries with the next one. Since the graph contains the traces of all crashes observed in practice, at least one of the traces is guaranteed to reproduce the crash.

6.1.2.2 Learning Crash-prone Execution Contexts

As previously mentioned, not all the devices suffer from the same bugs and some crashes only arise under specific execution contexts—*e.g.*, network unavailable or high CPU load. Hence, the `Crash Analysis` component searches for recurrent context patterns within a consolidated trace, which help to 1) reproduce context-sensitive crashes, 2) select the candidate devices to assess the generated reproducible scenarios, and 3) select devices to check that future fixes do not produce any side effects. The store learns both *dynamic* and *static* context patterns.

Dynamic Context. To learn frequent dynamic contexts from a trace, we use *Sequential Pattern Mining*, which is a data mining technique to discover frequent subsequences in a sequence database [111]. A sequence is a list of itemsets, where each itemset is an unordered set of items. We concatenate the context properties reported in each step of the consolidated trace. Fig. 6.4 shows 3 sequences of context properties along the consolidated trace (extracted from Fig. 6.3). We concatenate the properties observed for each step of the consolidated trace in the different crowdsourced logs. Each of these sequences contains 4 itemsets, one for each of the events in the trace, and each item maps to a context property.

In particular, we mine *frequent closed sequential patterns*—*i.e.*, the longest subsequence with a given support. The support of a sequential pattern is the percentage of sequences where the pattern occurs. To ensure that the context truly induces the crash, MoTiF searches for closed sequential patterns with support 100%—*i.e.*, patterns that appear in all the observed traces. Among the available algorithms to mine closed sequential patterns (*e.g.*, BIDE+, CloSpan, ClasSP), we choose *BIDE+* because of its efficiency in terms of execution time and memory usage [150]. In particular, we use the implementation of BIDE+ available in the SPMF tool [spmf].

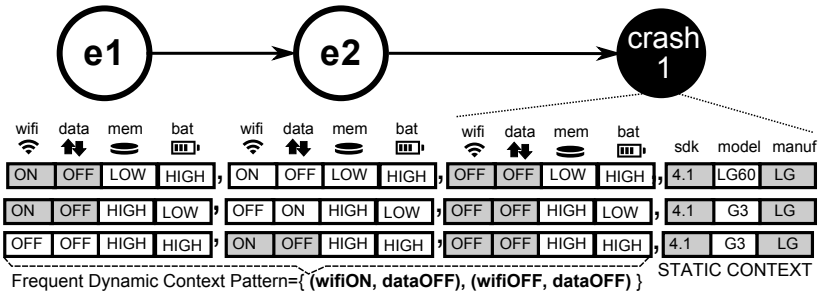


Fig. 6.4 Learning the crash-prone context from a candidate trace.

In Fig. 6.4, the algorithm identifies the following frequent context pattern: {(`wifiON`, `dataOFF`), (`wifiOFF`, `dataOFF`)}. This means that the properties (`wifiON`, `dataOFF`) are observed at the same time, and eventually are followed by the properties (`wifiOFF`, `dataOFF`) appearing together. Note that the itemsets appear in sequence—*i.e.*, the second itemset is always observed after the first itemset—but it does not need to be occurring at the same time in each trace. In the example, the pattern reveals that the crash arises when the network is being disconnected (`wifiOFF`).

Static Context. The example above shows the power of sequential pattern mining compared to simple intersection operations, since the former enables to capture relevant context changes, for example running out of memory (*e.g.*, the sequence {`memHIGH`, `memLOW`}), a network disconnection, or an empty battery.

However, to identify relevant static contexts, which do not evolve over time, we can just use *set operations*. For example, in Fig. 6.4, the union set across all traces for the *sdk* property is {`4.1`}, for the *manufacturer* property is {`LG`}, and for the *model* property is {`LG60, G3`}. In other words, the crash affects LG devices that run Android 4.1, and has been observed in LG60 and G3 models. The resulting sets are the relevant static contexts and will be used to select the devices to reproduce the crashes.

6.1.3 Generating Reproducible Scenarios

To help developers to reproduce crashes faced by users in the wild, the App Store 2.0 generates *reproducible scenarios* to automatically recreate crashes. The `Crash Analysis` component translates the *consolidated trace* and its *crash-prone execution context* (cf. Section 6.1.2.2) into a reproducible scenario.

The scenarios are encoded as black-box UI tests to automatically reproduce the sequence of user interactions that lead to the crash of the app. To implement the scenarios we use *Robotium*, which is a test automation framework for automatic black-box UI tests of Android applications [rob]. We chose *Robotium* because supports native and hybrid applications, does not require the source code of the application under test, and provides fast test case execution.

We propose mapping rules between the Android event handler methods (Section 4.2.3) and the methods provided by the Robotium API [Robotium API]. For example, the Android event `onClick` in a view of type `Button` is mapped to the *Robotium* method `clickOnButton`. Table 6.2 shows a subset of the mapping rules identified. These rules guide the automatic generation of reproducible scenarios from app execution logs. Using the mapping rules, we translate each event in the consolidated trace into a Robotium method invocation.

Table 6.2 Examples of mappings between Android event handler methods and Robotium methods.

Element	**Android method**	**Robotium method**
View	onLongClick	clickLongOnView
Button	onClick	clickOnButton
TextField	setText	typeText
ActionMenu	onMenuItemClick	clickOnActionBarItem
Orientation	onOrientationChange	setActivityOrientation

We define a base template for a Robotium scenario (Figure 6.5). The process to generate a Robotium scenario from a consolidated trace is as follows. First, the `Crash Analysis` component adds the crash-prone context as an annotation in the test case (`A`). Second, it sets the launcher activity of the subject app (`B`). Finally, it generates a test method to recreate the steps of the consolidated trace (`C`).

Figure 6.5 shows the Robotium scenario generated for the Wikipedia app. The test method `testRun` recreates the consolidated trace. Lines 2 and 6 correspond to the events $e1$ and $e2$ in the trace, respectively. Lines 1 and 3 represent delays between events. We calculate the delay between two events as the average of all the observed delays between those events. Finally, lines 4 and 5 set the network context. Network-related contexts can be automatically induced in the test cases because Robotium provides dedicated methods (`setWiFiData`, `setMobileData`) for this purpose. For other context properties, the observed context is added as an annotation in the test case to help developers to isolate the cause of failures (*e.g.*, $@Device(`LG')$).

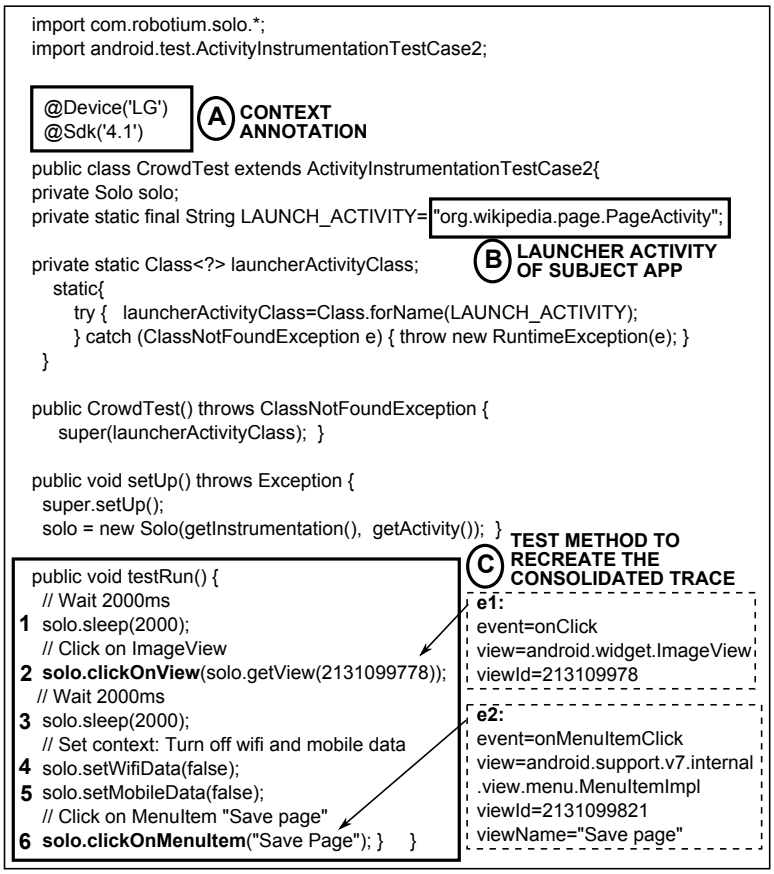


Fig. 6.5 Generated reproducible scenario for the Wikipedia app

In addition, a natural language description of each scenario is provided to developers.

6.1.3.1 Crowd-validation of Reproducible Scenarios

Before providing the generated reproducible scenarios to developers, the App Store 2.0 executes the tests in the crowd of real devices to assess whether or not they truly reproduce the observed crashes in the proper context.

First, the `Crash Analysis` component uses the static context to select a sample of devices that match the context profile (*e.g.*, LG devices), then it checks if the scenario reproduces the crash in those devices. We define the following heuristic to assess scenarios: the scenario execution should fail and collect the same exception trace as the original wild crash.

Later, the `Crash Analysis` component selects a random sample of devices that do not match the context profile, and tests whether they reproduce the crash. If the scenario

indeed reproduces the crash in a different context, the store concludes that the learned context is not discriminative enough. In this case, we add the context in the scenario as a note to developers, mainly informing him about the devices most frequently running their apps. If on the contrary, the scenario only reproduces the failure on the consolidated context, that context will be included as a *critical* annotation in the test. Note that, to avoid any user disturbance, the store executes the tests for validation only during periods of phone inactivity, *e.g.*, during the night, and when the device is charging.

During the execution of the scenarios for validation, the store could additionally log UI performance metrics (cf. Section 4.2.3.2) to populate a repository of *historical executions* for different context configurations in realistic scenarios. These realistic scenarios and repositories could be used as input for the `Performance Analaysis` component presented in chapter 5 (cf. Section 5.2). Thus, combining performance analysis *in vitro* (with tests generated by developers) and *vivo* (with realistic scenarios generated by users).

6.1.4 Implementation Details

This section provides details about the infrastructure that supports MoTiF.

Figure 6.6 shows an overview of the proof-of-concept implementation.

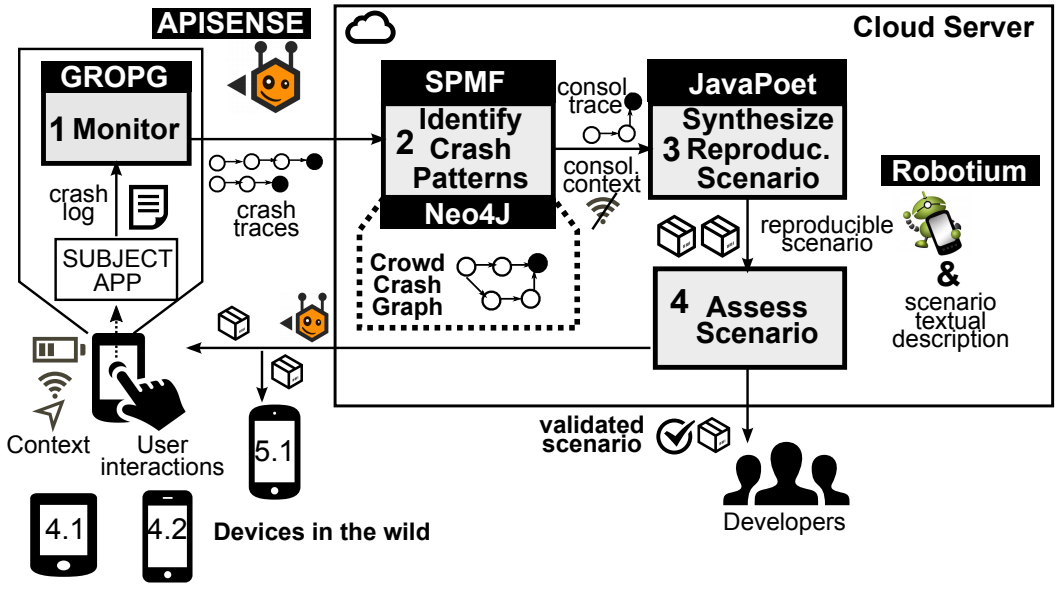


Fig. 6.6 MoTiF implementation

Our prototype implementation includes two parts: an Android client library that runs on the mobile device and a cloud service component. MoTiF can monitor any *debuggable* app[1] running on a mobile device, without requiring access to its source code. Our approach is transparent to users, who can keep on using their apps as usual. They only have to give their consent to automatically report debugging information when an app crashes in their devices, just like current error reporting systems do.

6.1.4.1 Android Client Library

The Android virtual machine (named *Dalvik*) implements two debugging interfaces: the *Java Debug Interface (JDI)* and the *Java Debug Wire Protocol (JDWP)*, which are part of the *Java Platform Debugger Architecture (JPDA)* [Java]. This technology allows tools such as the `adb` tool (Android Debug Bridge) to communicate with a virtual machine. MoTiF's client app runs `adb` on the device and communicates with Dalvik via `adb` and the standard debugging interfaces JDWP and JDI over a socket. For this, our tool extends and reuses part of the implementation provided by *GROPG* [121], an on-phone debugger. The *GROPG* implementation ensures low memory overhead and fast execution, and it enables to monitor apps and to intercept user interaction and exception events.

6.1.4.2 Cloud Service

MoTiF sends the data collected in devices to a cloud service for aggregation and analysis using APISENSE [apisense]. APISENSE provides a distributed crowd-sensing platform to design and execute data collection experiments in mobile devices [83]. APISENSE enables the comunication (sending and receiving data) between devices and the cloud.

To store and aggregate the crash traces collected from the crowd and the crowd crash graphs, MoTiF creates a *graph database* with *Neo4J* [Neo4J]. Graph databases provide a powerful and scalable data modelling and querying technique capable of representing any kind of data in a highly accessible way [134].

We can then query the graph database using the Cypher graph query language [Cypher], which is a widely used pattern matching language. Figure 6.7 shows an excerpt of the Crowd Crash Graph in Neo4J of the *Wikipedia* app. To extract the *consolidated traces*

[1] These are apps that have the `android:debuggable` attribute in their manifest.

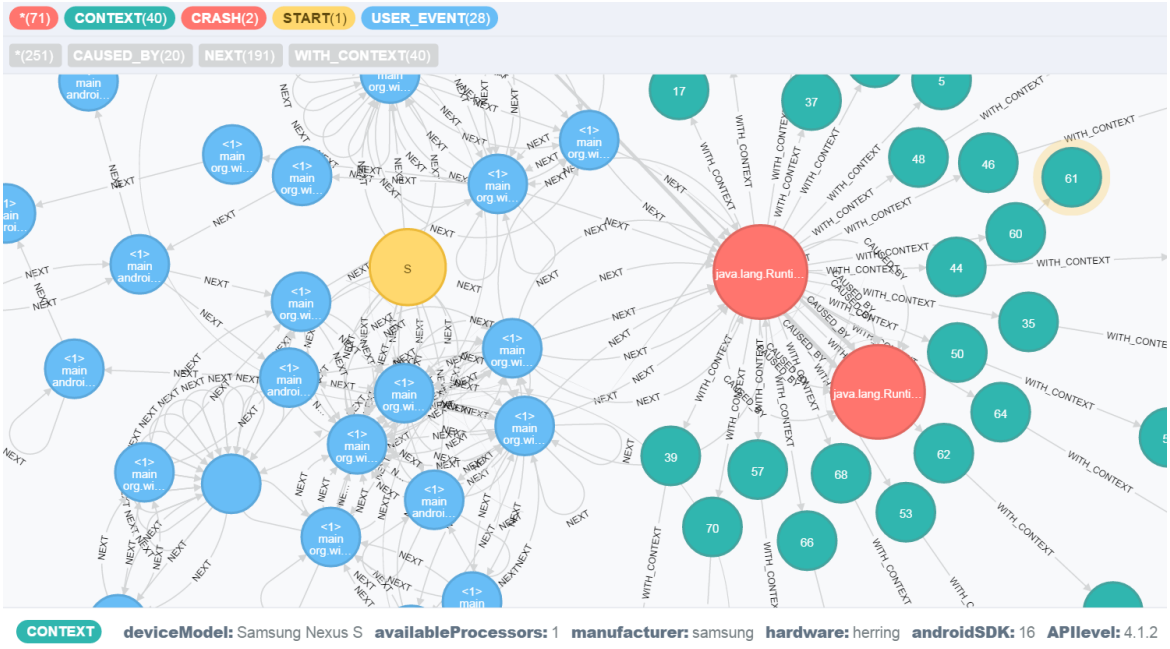


Fig. 6.7 Crowd crash graph in Neo4J

from the Neo4J graph database, we have implemented the *Dijkstra* algorithm as a Cypher query. We can then directly query the graph to extract the consolidated traces from the aggregated crowd data. Figure 6.8 shows the Dijkstra's implementation in our Neo4J database to extract consolidated traces. To learn context patterns along the

```
MATCH (from: APP), (to:CRASH_EVENT),
paths = allShortestPaths( (from)-[:NEXT*]->(to) )
WITH REDUCE( dist=0, rel in rels(paths) | dist+rel.pN )
AS distance, paths
RETURN paths, distance
```

Fig. 6.8 Dijkstra algorithm implementation in Cypher

consolidated trace we use the BIDE+ algorithm provided by the SPMF library [spmf].

Finally, to generate reproducible scenarios we translate the consolidated trace and context into a Robotium test [rob] and a textual description of the scenario. To generate the java code of the Robotium tests, we use *JavaPoet* library which enables generating .java source files.

6.2 Patching Defective Apps in the Wild

Prior research has demonstrated that users who encounter app crashes are likely to stop using the app [88]. Preventing and quickly fixing crashes have therefore become a major goal of app developers.

The App Store 2.0 can contribute to automatically prevent crashes in mobile apps by exploiting crowd feedback. When a user experiences a crash, we expect others to suffer from a similar situation. Thus, by mining such crashes reported by the crowd of users, we can learn the conditions triggering such a crash and prevent it. We propose Crowdseer, an automated crash prevention mechanism, which learns from crashes observed in a crowd of mobile devices to anticipate and prevent the emergence of the same crashes for other users.

While the developer investigates and releases a sustainable fix (which might take several days [117]), the `Patch Generation` component can generate temporary patches to prevent the occurrence of an observed crash for different users. The `Patch Generation` implements the Crowdseer approach. Crowdseer proposes two patching strategies to prevent app crashes from app stores. The first strategy consist of muting unhandled exceptions raised during the execution of apps. The second strategy consist of deactivating UI features (*e.g.*, a menu option, a button) triggering crashes. The two strategies are complementary and can be activated simultaneously.

The patching process starts when the App Store 2.0 flags an app as defective (cf. Chapter 4, Section 4.2.3). The `Patch Generation` component receives as input the crash patterns (*i.e.*, consolidated trace and context) extracted by the `Crash Analysis` component (cf. Section 6.1.3) and generates candidate patches to prevent the crashes.

6.2.1 Patch strategy 1: Muting unhandled exceptions

The first strategy synthesizes candidate patches to avoid crashes during the execution of apps. This strategy generates a new version of the app for each patch that synthesized. Thus apps need to be re-installed after patching.

This patch generation process is as follows. First, the `Patch Generation` component extracts the exception trace of a crash log. Figure 6.9 shows an example of an exception trace thrown by a defective app.

```
    FATAL EXCEPTION: main
    java.lang.RuntimeException: Unable to start activity ComponentInfo
    {com.snowbound.pockettool.free/com.snowbound.pockettool.free.LevelSelector}:
    java.lang.NullPointerException
1:  at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2059)
      ...
2:  at com.android.internal.os.ZygoteInit.main(Native Method)
3:  at dalvik.system.NativeStart.main(Native Method)
4:  Caused by: java.lang.NullPointerException
5:  at com.snowbound.pockettool.free.LevelSelector.getWorldList(LevelSelector.java:78)
6:  at com.snowbound.pockettool.free.LevelSelector.onCreate(LevelSelector.java:43)
7:  at android.app.Activity.performCreate(Activity.java:5008)
      ...
```

Fig. 6.9 Example of exception trace thrown by a defective app

Second, the component extracts the n frames that refer to the defective app. In the sample app, the 2 frames that start with the package name of the app (*com.snowbound.pockettool.free*). From each suspicious frame, we extract the name of the suspicious method and the class that implements it. In our example there are 2 suspicious methods (cf. Fig. 6.9, lines 5 and 6): *getWorldList* and *onCreate*, defined in the class *LevelSelector*. The patch wraps the code defined inside the suspicious methods with a `try/catch` block to capture the runtime exceptions that are not handled by the methods.

Next, the `Patch Generation` component creates n different patches, where n is the number of suspicious methods. For each synthesized patch, the store creates a new release of the defective app which includes the patch. In the sample app, the component creates 2 patched versions, where each patch wraps a different suspicious method.

Current app stores only have access to the bytecode of apps. To inject the patches, the store instruments the bytecode of the apps. Nevertheless, the App Store 2.0 could also enable developers to upload the source code of their apps to allow for more powerful repairing techniques. Finally, the different patched app versions are deployed on different user devices available in the crowd.

When the store receives a download request for an app that has been previously flagged as defective, the store delivers an alternative patched release of the app. Afterwards, the store keeps monitoring crowdsourced information from devices and user feedback that run those patched apps to assess the effectiveness of the generated patches. If a patch generation technique fails (*e.g.*, the patched app still crashes), the store learns from these failures. If a delivery strategy distributes patches to the wrong set of devices (*e.g.*, not all the devices suffer from the same bug), the store detects it. The App Store 2.0 continuously monitors crowdsourced information to improve the patching process.

In the sample app, the patch applied in the *getWorldList* method (Patch1) continues throwing the exception, whereas the patch applied in the `onCreate` method (Patch2) avoids the crash. Therefore, the store learns that Patch1 is ineffective and automatically discards it.

6.2.2 Patch strategy 2: Deactivating UI features

Industrial experience have stated that asking users to re-download apps make to lose a certain percentage of users, who never do it [27]. Thus, it is necessary to count with solutions that can prevent crashes without requiring users to re-download apps.

The second patch dynamically and contextually disables app features to limit the access to functionalities that have been previously observed as buggy by other users. In particular, we adopt the concept of **interaction feature** defined by Zaeem *et al.* [162]: *"An interaction feature is an action supported by the mobile platform, which enables a human user to interact with a mobile app, using the mobile device and the graphical user-interface (GUI) of the app".*

As illustration, users recently experienced crashes with a version of the *Wikipedia* app (v2.0-alpha), which contains a crash-inducing bug. The app crashes when the user requests to save a page when the network connection is unavailable. After observing several instances of this crash, the App Store 2.0 disables automatically the *'Save page'* menu in different devices to prevent the crash. Based on the crash patterns and context patterns identified by the `Crash Analysis` component (cf. Section 6.1.2), the `Patch Generation` component generates a preventive patch to disable the interaction feature that triggers the crash depending on context. The patch automatically disables the trigger of the crash (*e.g.*, 'Save page' on Wikipedia app example), thus avoiding the crash to emerge and to disrupt the user experience.

This patch generation process takes as input a *crowd-consolidated trace* (from the `Crash Analysis` Component) and proceeds as follows.

1. Patch Generation. First, the `Patch Generation` component processes the *crowd-consolidated trace* and selects the last event before the crash. From that event, the `Patch Generation` component extracts the feature involved (UI element) and such feature is promoted as the candidate feature to be disabled. From the consolidated trace of the Wikipedia app, the option 'Save Page' will be disabled, thus preventing other users to access such functionality to crash their app. The option will only be

disabled when the network connection is unavailable. Additionally, a message informs the users about the temporary unavailability of the feature. Apart from disabling UI features, the patch can also disable the rotation of specific Activities when an orientation change is observed previous to a crash. In Android, many crashes manifest when the device is rotated from landscape to portrait orientation or vice versa due to a deficient implementation of the Activities life-cycle [162].

Finally, the store automatically deploys the preventive patch in all the instances of the app which run in different devices in the crowd. The store uses the context patterns to select the set of users which have a similar context profile. The store updates all the apps on-the-fly, without requiring the developers to publish a new version of the app.

2. Continuous Patch Validation. After applying the preventive patch, the store continues monitoring the app to validate the effectiveness of the preventive patch. If, after disabling a feature, the app continues exhibiting the same crash, then the store reverts the patch and mines for a different crash-triggering feature to disable. In this case, the `Patch Generation` component tries with the previous feature in the consolidated trace. In the particular case of the Wikipedia app, the deactivation of the 'Save page' menu option effectively avoids the crash when network is disabled. But if after disabling this menu option the app still crashes, then component will try with the previous feature in the trace—*i.e.*, the full Menu would be disabled. The store continuously monitors the crowd and supervises its own preventive actions to make corrections if necessary.

The App Store 2.0 quickly identifies issues and provides immediate patches which prevent other users to experience same issues while the developer is fixing the app and publishes a new release in the store. This is especially important when considering that publishing a new version of the app can take from hours to several days, due to the validation process of the app stores. This strategy reduces the time that users are exposed to crashes as well as it reduces the number of affected users.

6.2.3 Implementation Details

This section provides details about the infrastructure that supports the Crowdseer module. Crowdseer implements two strategies, injecting try/catch blocks to mute exceptions and deactivating crash-triggering UI features on the fly. The following sections present the implementations which support each strategy.

6.2.3.1 Dealing with Try/Catch Injection

For the first patching strategy, we have implemented a Java program, which instruments the bytecode of Android apps using *Dexpler* [49]. This programs takes as input an .apk file (of the Android app to) and two input parameters: *class name* and *method name* where the patch has to be injected. Then, it returns as output a new .apk file which includes the patch. The instrumentation program is available online[2].

6.2.3.2 Dealing with Feature Deactivations at Run-time

In this section we describe the implementation of the second patching strategy proposed in this thesis. Following the concepts of *Infrastructure-as-a-Service* [92], our software implementation is composed of two parts: 1) a mobile client library to embed in apps; and a 2) a cloud service to update apps on-the-fly. A short demo of the infrastructure is available online[3].

Crowdseer Client Library. We have implemented an Android library, to be integrated in an app, to incorporate the automated crash prevention mechanism. This library enables apps to communicate with the cloud service to upload crash logs and receive preventive patches on-the-fly.

Android apps are *user-interface* (UI) centric. The graphical user interface of Android apps consists of components called `Activity`, which renders a screen to interact with users [2]. Each *Activity* corresponds to a core function of the app. The user interface for an activity contains a hierarchy of views. A `View` is an object that models a user interaction modality [7], such as `Button`, `Menu`, etc.

The Crowdseer library implements a *"UI facade"* that exposes a series of methods to disable GUI components of the app dynamically. For example, the method `disableViewById(int id)` disables the view (such as a button) with the `id` provided as a parameter. The *facade* uses *Aspect-Oriented Programing* (AOP) to update the UI. For this purpose, we uses the *AspectJ* library available for Android [aspectj].

Crowdseer Cloud Service. Crowdseer sends the crash logs collected by mobile devices to a cloud service for aggregation and analysis using APISENSE [apisense].

[2]https://www.dropbox.com/sh/u3ffy1lw85opww8/AACBLu2zcTCUNgXAFh7dpDbma
[3]https://www.youtube.com/watch?v=o7PnOpgoeJ8.

```
var ui = require('UI');
var wifi = require('wifi');

wifi.onStateChanged(function(wifiEvent) {
        if (wifi.isState(wifi.DISCONNECTED)) {
                ui.disableMenuItemById('@+id/menu_save_page');
                ui.showToast('The menu 'Save page' is temporarily unavailable');
        } else
                ui.enableMenuItemById('@+id/menu_save_page');
});
```

Fig. 6.10 Example of a CROWDSEER script

APISENSE is a distributed crowd-sensing platform to design and execute data collection experiments in mobile devices [83]. CROWDSEER uses the RESTful interface of APISENSE to update apps on-the-fly. The patch to be remotely applied in the apps are specified in JavaScript language and posted by CROWDSEER to APISENSE for deployment. APISENSE also bridges the Android SDK with a scripting engine so that all the methods defined by the *"'UI facade"* are made accessible from scripts. Thus, the preventive patches to disable features are generated by CROWDSEER as JavaScript scripts. Figure 6.10 shows an example of the preventive patch for the *Wikipedia* app is defined as follows:

This script indicates that when there is change in the network state, if the state is `DISCONNECTED`, then it disables a menu item (whose *id* is '@+idmenu_save_page') and shows a toast message to inform the user. When the network connection is available, then it enables the item again.

6.3 Conclusions

This chapter presents two engineering approaches to automatically generate *reproducible scenarios* and *app patches* by analyzing crowdsourced crash logs and app contexts.

The *reproducible scenarios* aim to assist app developers to reproduce *in vitro* crashes that were faced by users in the field. The *app patches* prevent future manifestations of an observed crash in the same user and other users. As a result, the user experience and satisfaction with apps increases. The generated actionable insights save effort and time for app developers during the bug-fixing process which is a crucial factor for succeeding in the highly competitive app market.

Part III

Empirical Evaluations

To demonstrate the feasibility of the App Store 2.0, we have implemented a set of supporting tools.
This part presents empirical studies to evaluate their capabilities.

Chapter 7

Evaluation of *in-vitro Approaches*

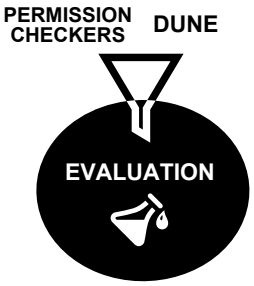


Fig. 7.1 Chapter 7 at a glance

In Chapter 5 we have presented two modules (`Risk Analysis` and `Performance Analysis`) to distill actionable insights from crowdsourced artifacts. We have implemented a set of supporting prototype tools for Android apps (cf. Appendix **??** for implementation details).

In this chapter, we apply the proposed approaches in practice and report on empirical studies to evaluate their capabilities. For each module we present the motivation, evaluation protocol, results, and threats to validity.

The rest of the chapter is structure as follows. Section 7.1 evaluates the accuracy of the generated crowdsourced checkers. Section 7.2 evaluates the DUNE approach. Finally, section 7.3 concludes the chapter.

7.1 Evaluation of Crowdsourced Checkers

In this section, we evaluate the accuracy of the generated crowdsourced checkers to predict buggy apps before release.

7.1.1 Empirical Study Design

To assess the effectiveness of our approach, we investigate three main research questions:
RQ1: *What is the accuracy of the inferred checkers?*
RQ2: *To what extent checkers are able to flag new apps as buggy?*
RQ3: *What is the effect of removing error-sensitive permissions on error reviews?*

In the following, we describe the studies we performed to answer these research questions.

7.1.2 Dataset

To assess the approach, we started by collecting apps from the *Google Play Store*. Since the Google Play Store is continuously evolving (adding, removing and/or updating apps), we performed three snapshots of the store in November 2013 (D_0), January 2014 (D_1), and March 2014 (D_2). Figure 7.2 depicts the evolution of our dataset along time in terms of additions, removals and updates of apps and permission types. D_1 contains 38,781 apps requesting 7,826 different permissions, while D_2 contains 46,644 apps and 9,319 different permission requests. The observed evolution tends to add new apps, more than updating or removing existing ones. In January, 15,023 new apps appeared, 9,001 apps (33,13%) were updated, and 3,411 apps (12,55%) were removed. Similarly in March, 12,543 new apps appeared, 8,970 apps (23,13%) were updated and 4,680 apps (12,07%) disappeared.

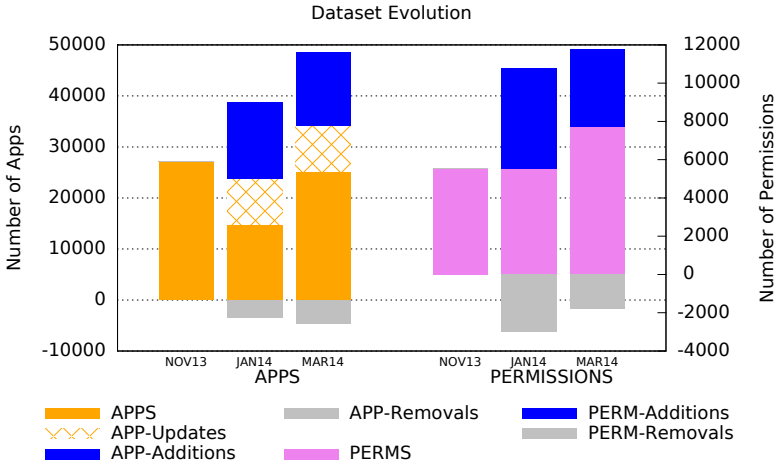


Fig. 7.2 Evolution of apps and permissions in the Google Play Store (between November 2013 and March 2014).

We observe that the app updates tend to rather add permissions than remove existing ones. Out of the 3,411 apps that updated to a new version in January 2014, there are 2,677 apps (78.48%) that update their set of permission requests. Specifically, there are 3,007 permission request removals and 5,225 permission request additions. This confirms that the trend of adding more and more permissions pointed out by previous studies [151] still holds. We have also observed that some apps are adding unnecessary permission requests when updating to a new version. For example, there are 74 apps adding the deprecated permission `[appPackage].permission.MAPS_RECEIVE` previously required to use the Google Maps API v1. The update of Google Play Services 3.1.59 (in July 2013) made this permission useless[1]. Furthermore, apps request 6 permissions by median, and at maximum, request 83 different permissions. In contrast, there are 2,592 apps that request 0 permissions.

7.1.3 Empirical Study Results

We now present the experiments we performed to answer the three formulated research questions and the results we obtained.

7.1.3.1 RQ1. Evaluating the Accuracy of the Inferred Checkers

Checkers often have false positives, which in our case means that the app is flagged as buggy while still works fine. To answer RQ1, we evaluate the accuracy of the inferred permission checkers (cf. Chapter 5, Section 5.1.3).

To evaluate the accuracy we use the *Laplace expected error estimate* [58], which is computed as follows:

$$LaplaceAccuracy = (n_c + 1)/(n_{tot} + k) \tag{7.1}$$

where k is the number of classes in the domain, *e.g. Buggy/NonBuggy* ($k = 2$). n_{tot} is the total number of examples covered by the checker. In our case, is the total number of apps requesting the permissions captured by the checker. n_c is the number of examples in the predicted class by the checker. In our case, is the number of *Buggy* apps that request the permissions captured by the checker.

[1]Google Maps Android API v2 Release Notes: https://developers.google.com/maps/documentation/android/releases?hl=en#july_2013

We set cross-validation 10-folds and we perform a sensitivity analysis of the accuracy of the checkers for different values of input parameters. Figure 7.3 shows the accuracy of the recommender system. The family of checkers exhibits an accuracy that ranges from 61.42% to 61.96%. We observe that the reported accuracy does not change significantly for different values of the parameters. Table 5.2 shows the *accuracy* (acc.) values for

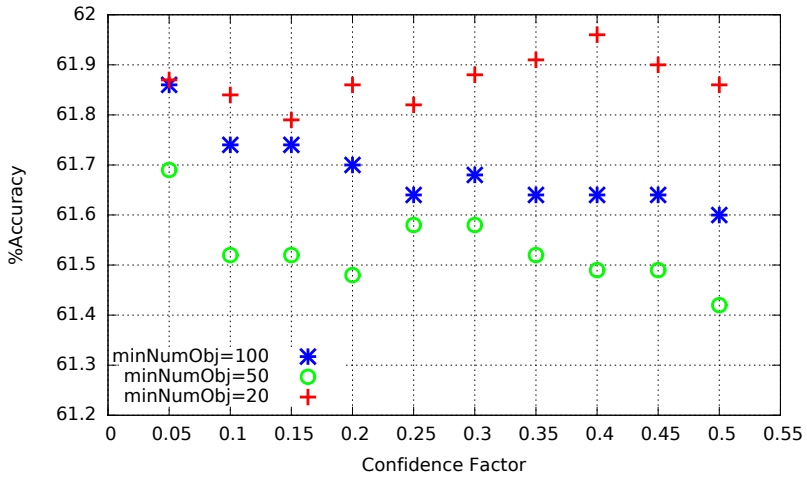


Fig. 7.3 Evaluation of the accuracy of the permission checkers.

some of the permission checkers identified. We also report the number of *Buggy* (B) and *NonBuggy* (NB) apps in the testing set, which request the permissions captured by the checkers.

7.1.3.2 RQ2. Evaluating Checkers with New Apps

For answering RQ2, we use the dataset D_2 updated in March 2014 (cf. Section 7.1.2). We select the subset of new apps that appeared in D_2 and did not exist in D_1. D_2 contains 6,783 new apps (with reviews). We use the user reviews as our ground-truth for determining if an app is buggy or not. Thus, we check if the D_2-version apps flagged as suspicious by our checkers have been reported as buggy by end users (after the publication of the version of D_2). Our system flagged 1,896 apps as error-suspicious. Out of 1,896 suspicious apps, 56% of apps were also reported as buggy by end users.

> The 56% of new apps flagged as suspicious by our checkers were also reported as buggy by end users *a posteriori*.

We further evaluate the performance of our approach by comparing it with alternatives. First, we compare our checkers built from permission patterns against: 1) a classifier that flags as suspicious apps the apps that request some single error-sensitive permissions without learning permission patterns; and 2) a random classifier that flags suspicious apps randomly. As Table 7.1 shows, our approach learning permission patterns performs better than the others.

Table 7.1 Comparison to alternatives

	Pattern-based	**Single perm.**	**Random**
Flagged apps	1,896	814	1,718
Bug-reported apps	**1,062 (56%)**	404 (50%)	815 (47%)

Remark that this measure is only an approximation, since we build an oracle of app bugginess from user reviews without performing in-depth analysis of apps. In fact, the lack of bug-related reviews does not necessarily imply that the app is bug-free, since an error can exist without being reported. Although this oracle is incomplete, it is a useful tool that helps store moderators to make informed decisions about app bugginess considering only information sources—*i.e.*, reviews—available on stores.

7.1.3.3 RQ3. Impact of Removing Error-sensitive Permissions

To answer RQ3, we investigate if the apps that remove error-sensitive permissions (captured by the checkers) in the update get less error-related reviews. We noticed 30 apps (with error-related reviews) that remove error-sensitive permissions after updating. For these apps, we compute the percentage rate of error-related reviews before and after the update. Figure 7.4 illustrates the evolution of the reviews in these apps. We observe that after removing error-sensitive permissions, 22 apps (out of 30) remove error-related reviews. If the checkers had been enabled in the app store, all those apps would have been flagged as suspicious before publication.

7.1.4 Threats to Validity

We focus on permission requests and user reviews to build a family of buggy app checkers that can help app store moderators to score the quality of a submitted app. One threat to the *internal validity* of our study is that we have not analyzed the source or binary code of apps to ensure that the declared permissions are actually used.

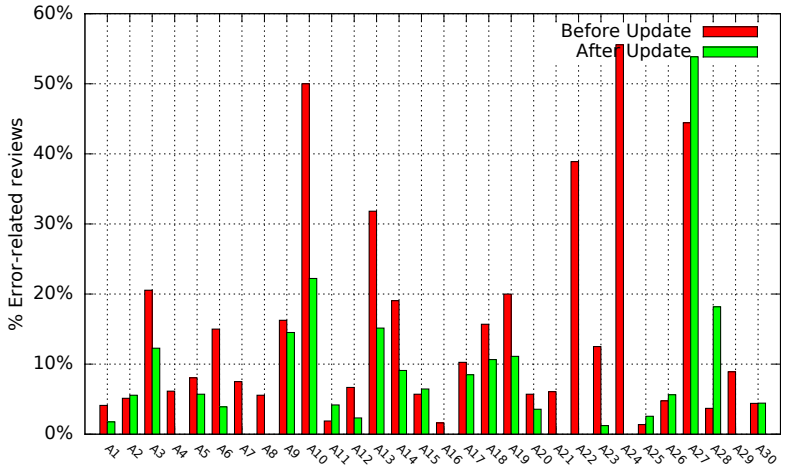


Fig. 7.4 Reviews evolution after removing error-sensitive permissions.

We consider user reviews as a ground truth of bugginess, but we are aware that this measure is only an approximation. Users do not always report crashes when faced, and some bugs can be reported later in time. In addition, apps can crash for many reasons, not only due to permission-related issues. To address this threat, the *in vivo* approaches (in particular the `Crash Analysis` module, cf. Chapter 6) complement this pre-release approach. The *in vivo* approaches immediately actuate after users experience crashes.

The conceptual foundations of our approach are independent of Android. We only need an oracle of bugginess and some observation features (in our case, the user reviews and the requested permissions, respectively). To gain in confidence in the *external validity* of our evaluation, more evaluations are needed on other platforms, using other oracles of bugginess and features.

Regarding sensitivity, the proposed approach is based on predefined thresholds specified by various parameters. Table 7.2 summarizes these parameters together with the reasonable values used in the experiments reported in this thesis (cf. Section 4.2.1). The calibration of these parameters can impact the results and constitute a potential threat to validity. We have performed a sensitivity analysis for different values of these parameters. The reported accuracy does not change significantly for different values.

Table 7.2 Summary of the approach's parameters

Parameter	**Value**
LDA model number of *topics* threshold	100
Error-related reviews threshold in suspicious apps	> 1
Composition threshold in error-related reviews	0.05
J48 model $confidence$ factor	0.4
J48 model $minNumObjects$	20

7.2 Evaluation of DUNE

In this section, we perform experiments to evaluate the effectiveness of DUNE to detect *UI performance regressions and optimizations* in apps executed on different devices with different hardware and software configurations.

7.2.1 Empirical Study Design

To evaluate the effectiveness of the DUNE approach we investigate four research questions:

- **RQ1:** *What is the effectiveness of the approach to identify new test runs with performance degradation*?
- **RQ2:** *What is the effectiveness of the approach to isolate contextual conditions under which performance degradations arise*?
- **RQ3:** *What is the effectiveness of the approach to identify performance optimizations*?
- **RQ4:** *What is the effectiveness of the approach to identify performance degradations at a fine granularity*—i.e., *UI event level*?

7.2.2 Dataset

We describe the *apps under test* (AUTs), the devices used, the tests developed, and the data collection procedure to set up the experiments.

We first mine real UI performance defects that are documented in issue tracker discussions and commit messages in software repositories of Android apps. In particular, we study DUNE on real UI performance issues reported in two Android apps: the *K-9*

Mail client, and the *Space Blaster* game. To further evaluate the approach, we use an additional app, the *ExoPlayer* Android media player. In this app, we manually inject janks in the context of our evaluation process. We provide a description of each AUT, and the UI performance issues that were reported.

- **K-9 Mail**. *K-9* is an open-source e-mail client for Android [k9]. The K-9 app (version 5.006) lags when opening a message from the list of messages. This issue happens after upgrading devices to Android Lollipop (V 5.∗) [k9-].
- **Space Blaster**. *Space Blaster* is a space shooter game that originated on the Nintendo and Sega consoles, and later has been ported to Android [spa]. Users experience performance degradations when playing on Android 5.0 devices.
- **ExoPlayer**. *ExoPlayer* is an open source media player for Android developed by Google [11]. *ExoPlayer* provides functionality to play audio and video both locally and over the Internet. It is currently used inside Google's popular *YouTube* and *Play Movies* applications [12].

We chose these apps because UI animations are prime (*e.g.,* scroll mails, game animations and play video). Thus, the UI performance is crucial and any performance degradation quickly impacts user's experience.

7.2.2.1 Building the Device Lab

To automatically run different versions of the AUTs in heterogeneous contexts in our experiments, we leverage the fast third-party Android emulator Genymotion [gen] (cf. Appendix **??**). *Genymotion* provides a wide range of images of real devices (with distinct hardware and software profiles), and it allows the developer to simulate different execution environments—*i.e.*, network types.

We select the images of 20 different Android devices with different hardware and software profiles to emulate a heterogeneous context. These context profiles correspond to common configurations of popular devices at the time of performing our experiments. The selected devices run the Android SDKs: 4.1.1, 4.2.2, 4.3 and 4.4.4. We call this lab the *Device lab*.

We then build an additional lab that contains 5 additional images of Android devices running the Android SDKs: 5.0, 5.1, and 6.0.0. The latter forms the *Upgraded device lab*. These two device labs will be used across different experiments. Table 8.2 shows

the set of the virtual devices we use in the study, with their hardware and software characteristics.

Table 7.3 Sample of the Device Lab used in the experiment.

Device	Brand	Model	SDK	API	CPU	Screen
D1	LG	Optimus L3	4.1.1	16	x86	240x320
D2	Google	Galaxy Nexus	4.2.2	17	x86	720x1280
D3	Motorola	Moto X	4.2.2	17	x86	1080x1920
D4	Samsung	Galaxy S4	4.3	18	x86	1080x1920
D5	Samsung	Galaxy Note 3	4.3	18	x86	1080x1920
D6	HTC	Evo	4.3	18	x86	720x1280
D7	Sony	Xperia Z	4.3	18	x86	1080x1920
D8	Google	Nexus 5	4.4.4	19	x86	1080x1920
D9	Google	Nexus 9	5.0.0	21	x86	1536x2040
D10	Google	Nexus 5X	6.0.0	23	x86	1080x1920
D11	Google	Nexus 5	5.1.0	22	x86	1080x1920
D12	Samsung	Galaxy S6	5.1.0	22	x86	1440x2560
D13	Google	Nexus 4	5.1.0	17	x86	768x1280
D14	Google	Nexus 4	4.4.4	19	x86	1080x1920
D15	Sony	Xperia Tablet Z	4.1.1	16	x86	1920x1200
D16	HTC	One	4.3	18	x86	1080x1920
D17	Motorola	Moto X	4.3	18	x86	720x1280
D18	Samsung	Galaxy S3	4.2.2	17	x86	720x1280
D19	Google	Nexus 7	4.3	18	x86	800x1280
D20	Sony	Xperia Z	4.2.2	23	x86	1080x1920

7.2.2.2 Setting Up Tests to Collect Data

We then define, for each AUT, user interaction scenarios to mimic the real usage scenarios that were reported in the issue trackers to reproduce the performance bugs. We specify 3 test scenarios, one for each AUT.

To automatically run and repeat these scenarios, we use *Robotium*, which is a test automation framework for automatic black-box UI tests for Android apps [rob]. *Robotium* executes a sequence of user interactions—*i.e.*, clicks, scrolls, navigations, typing text, etc.—on an app without requiring the source code, and it is able to do this fast. During such a test execution, we log the timestamp of each executed *UI event*, and we profile the *GPU rendering* of the app.

7.2.2.3 Benchmark Suite

To evaluate the performance of DUNE, we run each test scenario on the 20 emulated devices in the *Device lab*. To attenuate noise and obtain more accurate metric values, we repeat the same test 10 times and accumulate the results as a mean value. We repeat this process twice on each device with 2 different types of networks—*i.e.,* WiFi and 3G. This results in $20 \times 2 = 40$ different test runs for each app, which constitutes the *historical test repository* of an app. We consider all test runs in the historical test repositories as having acceptable performance.

Finally, we run the same tests in the 5 devices in the *Upgraded device lab*. This results in 5 new test runs per app.

7.2.3 Empirical Study Results

Using the repositories presented above, we performed a series of 4 experiments to evaluate DUNE. We perform an experiment for each research question. For each experiment, we provide a motivation, present the approach followed and our results.

7.2.3.1 RQ1. Performance Degradation

Motivation: The first goal of DUNE is detecting performance deviations in a new test run in comparison with previous test runs. In this experiment, we study the coarse-grained filtering performance of DUNE to flag tests that contain performance outliers.

Approach: To perform this experiment, we select two real context-related performance bugs reported in the *K-9* and *Space Blaster* apps. Both apps reported UI performance issues after upgrading devices to Android Lollipop ($v5.*$). The app developers confirmed these issues. While the most recent version of the *K-9* app fixes the problem, the bug remains in the *Space Blaster* app.

In order to simulate the device upgrades reported in the issues for the AUTs, we select, for each app, the 5 test runs which were executed in the *Upgraded device lab* (cf. Section 7.2.2.3). These test runs exhibit performance degradation. We then compare each new test with the repository of historical tests, thus we perform 10 of such comparisons.

Results: **Dune correctly flags all performance regressions in our dataset.** Out of 10 new tests (5 per app), DUNE flags the 10 tests as containing performance degradations.

Figure 7.5 compares the frames rendered during the execution of the test scenario in the *Space Blaster* app in the historical repository (left) and in an updated device (right). In particular, we illustrate the test runs in devices Google Nexus 4 with Android 4.3 and 5.0.0, respectively. In the historical test, the average time to render each frame is 8 *ms* which is under the 16 *ms* threshold. On the contrary, the average rendering time per frame is 144 *ms*. In addition, while the historical test execution rendered 2,145 frames and only 6 frames were janky (the smooth ratio is 99%), the new test only rendered 176 frames, thus showing that more than the 90% of frames were skipped and resulting in laggy animations perceivable by users.

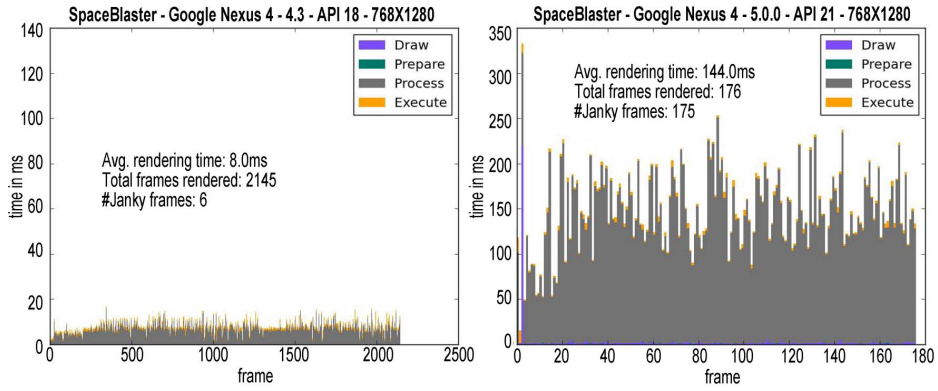


Fig. 7.5 Frames rendered during the test scenario execution in the *Space Blaster* app in the historical dataset (left) and in the new context (right).

Similarly, Figure 7.6 shows the frames rendered during the execution of the test scenario in the *K-9* app on a Google Nexus 7 device with Android 4.3 in the historical test repository (left) and in the same type of device with Android 5.1.0 (right).

The task to manually determine if a new test induces a performance degradation along its execution is challenging and time-consuming. This task becomes harder as the size of the historical test repository augments and the number of considered metrics increases. Nevertheless, DUNE is able to correctly identify each performance outlier in ~ 1 *ms* in our datasets.

Figure 7.7 shows the outliers flagged in the new test runs in comparison with the historical test runs in the *K-9* app for the metrics: *number of frames*, *smooth ratio*, and *average time to render each frame*.

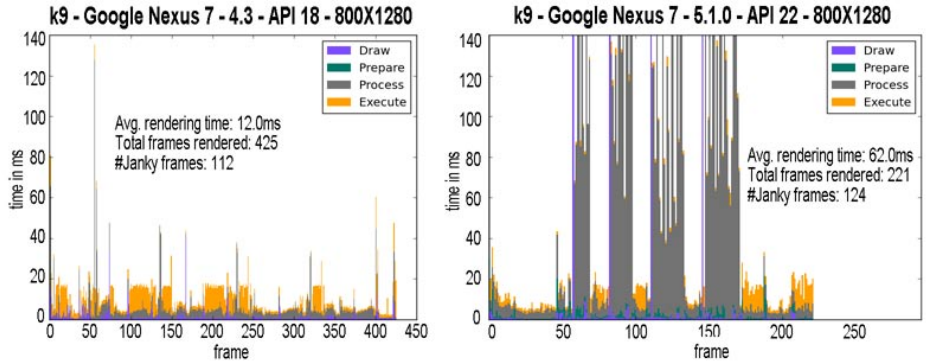


Fig. 7.6 Frames rendered during the test scenario execution in the *K-9* app in the historical dataset (left) and in the new context (right).

7.2.3.2 RQ2. Context Isolation

Motivation: The second goal of DUNE is to pinpoint potential contexts that induce janks. This experiment aims to assess this capability.

Approach: To perform this evaluation, we use the test repositories generated in *Experiment A* for the *K-9* app. We then evaluate the effectiveness of DUNE to identify the conflictive contexts that were reported as causing the performance issues in the *K-9* app. We let DUNE learn context rules with minimum support 0.01 and minimum confidence 0.9. We set minimum support 1%, since the proportion of new test runs is small in comparison with the number of test runs in the historical dataset. For example, when a new test run includes a new context property that was not available in the historical tests, *e.g.,* a new SDK version is released, these properties have a low frequency. By setting a low minimum support value we ensure that the rules capture these new infrequent properties as they emerge. On the contrary, we set confidence to 90%, since we want to ensure that the context properties are highly correlated with outliers. This is the default value provided in the Weka library used in the implementation.

Results: **Dune effectively learns context rules where janks arise.** As a result, DUNE identifies 7 context rules (after pruning redundant rules). All these rules have a confidence of 100%. Figure 7.8 shows a graph-based visualization of the context rules identified in the *K-9* app. In the graph, nodes represent items (or itemsets) and edges indicate relationship in rules. The size of the nodes represents the support of the rule. In particular the contexts are the SDKs: 5.0.0, 5.1.0, and 6.0.0, and their associated API levels: 21, 22, and 23, respectively. Thus, DUNE effectively

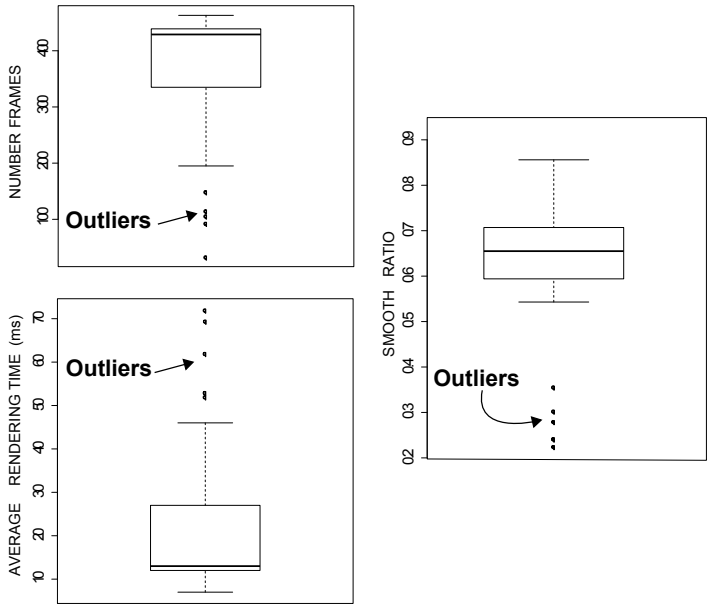


Fig. 7.7 Outliers flagged in the new test runs in the *K-9* app in comparison with the historical test runs.

identifies the contexts that were reported as problematic in the issue tracker system. Furthermore, even though the SDK 6.0.0 was not reported in the apps' issue reports[2], we manually repeated the scenario on a device with 6.0.0 and we could perceive UI lags as described. This finding underlines the potential of the technique to identify performance regressions across contexts. In addition, the *Google Nexus 5X* device appears in a rule, since it is the unique device in the repository that runs the version 6.0.0.

To analyze the sensitivity of the parameters in this experiment, we increase the minimum support threshold with deltas of 0.01 and explore the rules obtained. When the minimum support reaches 0.03, the number of context rules found is 5. In particular, the contexts `SDK=6.0.0` and `device=Google Nexus 5X` are pruned. Whereas the remaining 5 rules are the same as those previously identified, which effectively reproduce the performance regressions. Moreover, when we set the minimum support to 0.07, no rules are found. In this experiment, all the rules exhibit a confidence of 1, which is the highest possible value. Developers can calibrate the parameters of DUNE (minimum support and confidence) according to their needs. If they are too restrictive, they risk to miss some defects, and harm the experience of their users. On the contrary, if they are too permissive, they can obtain context rules that are not discriminative enough.

[2]At the time of writing, Android 6.0.0 (Marshmallow) is one of the latest Android version released and it is few widespread among devices.

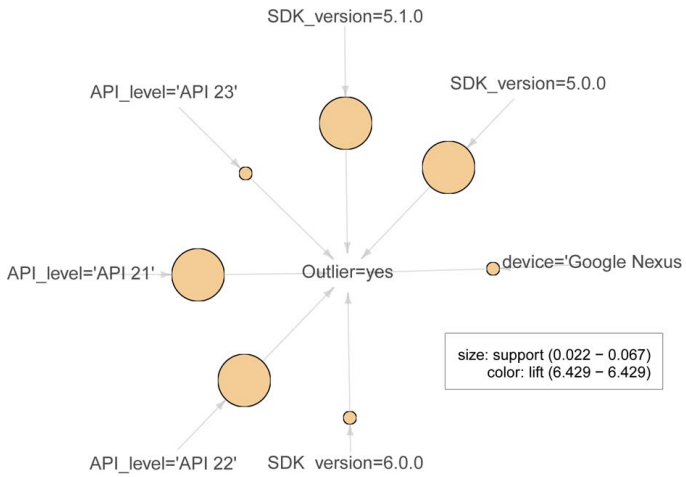


Fig. 7.8 Graph visualization of the context rules identified in the *K-9* app.

7.2.3.3 RQ3. Performance Optimization

Motivation: In addition to detecting performance degradations, improving app performance (for example in low-end devices) is another significant performance goal which app developers persecute. Hence, developers can use DUNE to assess performance optimizations. The goal of the third experiment is to assess this capability.

Approach: We select the 10 new tests that were generated in *Experiment A* for both apps *K-9* and *Space Blaster*. Then, we randomly select one of the tests in each app from the *historical* test repository—*i.e.*, we swap the new tests and one of the elements of the test repository. Finally, we check if DUNE correctly flags the historical test as an outlier (optimization) in comparison with the new tests. In this case, the outlier should have a positive offset, hence indicating a performance optimization.

Results: **The approach effectively reports the historical test as an outlier with positive offset**, hence indicating that this run is better than previous tests. This result confirms the utility of DUNE, which can spot both regressions and optimizations effectively.

7.2.3.4 RQ4. UI Event Filtering

Motivation: The last experiment aims to evaluate the capability of DUNE to filter out individual events that trigger janks. Hence, in this experiment, we evaluate the fine-grained filtering performance to identify performance degradations at UI event

level. We check if DUNE correctly points out the events in a test that trigger a jank. The more irrelevant UI events DUNE can discard, the less effort developers require to find the root cause of problems.

Approach: In this evaluation, we use the *Exoplayer* app with manually injected janks. Since we know the location where the janks were injected, we can assess the effectiveness of the approach.

Google states that most of the performance issues in apps come from performing heavy operations in the main UI thread. Thus, we inject a synthetic jank, extracted from Sillars et al. [64], in the *Exoplayer* app to cause UI performance degradation. The jank consists of inserting a heavy computation in the UI thread. We inject the jank in two locations of the app—*i.e.*, when the user selects the option *Logging* during video playback and when the user touches the screen during video playback. We then define a realistic user interaction scenario for the AUT. At a high level, the scenario consists of selecting and playing two videos from the Internet, and change different configuration settings while the video is playing. In total, the scenario is composed of 20 user interactions and takes 102 seconds to execute on average. Across the execution of the test scenario the jank manifests 4 times after events: *#3, #7, #16, #17*.

We then run the test scenario with the mutated version of the *Exoplayer* app on 15 devices randomly selected in our *Device lab*. This results in a *mutated test repository* with 15 test runs that exhibit performance degradations. Remark that all the test executions traverse the injected faults.

We measure the effectiveness of DUNE in terms of *Precision* and *Recall* which are computed as follows [68]:

$$\text{Precision} = 1 - \frac{\#false\ positives}{total\#\ of\ events\ flagged}$$

$$\text{Recall} = \frac{\#janky\ events\ detected}{\#expected\ janky\ events}$$

False positives are events flagged as janky, while we did not include a jank in such a position. Note that the precision metric is only an approximation, since we cannot ensure that the false positives really are false alarms. The app could experience performance degradations at some points due to external factors, such as network variations, device state (*e.g.*, device running out of battery), etc. The less false positives, the less locations the developer will need to explore to fix the janks. On the contrary,

Table 7.4 Summary of the performance of DUNE to spot UI events.

	Janky events:**#3, #7, #16, #17**		
Test	**Flagged Events**	**Precision**	**Recall**
$T1$	**3**,5,**7**,**16**	0.75	0.75
$T2$	2,**3**,5,**7**,**16**,**17**	0.67	1
$T3$	**3**,5,**7**,**16**	0.75	0.75
$T4$	**3**,6,11,**16**,**17**	0.60	0.75
$T5$	**3**,11,**16**,**17**	0.75	0.75
$T6$	**3**, 6,11,**16**,**17**	0.60	0.75
$T7$	1,6,**7**,8,9,12,14,**16**,**17**,18	0.30	0.75
$T8$	1,2,**3**,**7**,9,11,12,**17**,18	0.33	0.75
$T9$	1,2,**3**,6,**7**,8,9,11,12,**16**,**17**,18	0.33	1
$T10$	**3**,4,6,**7**,8,**16**,18	0.43	0.75
$T11$	**3**,4,6,**7**,8,**16**,18	0.43	0.75
$T12$	**3**,6,**7**,8,**16**,**17**	0.67	1
$T13$	2,**3**,4,6,**7**,8,**16**,**17**,18	0.44	1
$T14$	1,2,**3**,4,6,**7**,8,**16**,**17**,18	0.40	1
$T15$	2,**3**,4,6,**7**,8,**16**,**17**,18	0.44	1
	average:	**0.53**	**0.83**

high recall means that the approach detects most performance degradations at specific events. The recall metric is the most important in this setting, since a high recall means that DUNE finds most of the janks.

Results: **The approach obtains an average precision and recall of 53% and 83%, respectively.** Table 7.4 reports the results of the experiment. For each mutated test, it shows the events that were flagged (bold events were flagged correctly), together with the approximate precision and the recall. Recall is the most important metric for DUNE, since it aims to find all occurrences of janks. The low precision means that several events were flagged without any jank being injected. After manual inspection of these incorrectly flagged events, we observe that many performance deviations are induced by the effect of the previous jank. For example, we injected a jank in event #7. We can observe that when event #7 is flagged in a test, the next event #8 is also flagged in most of the tests, since the app has not recovered yet. Although event #8 is considered as a false positive in the evaluation, it is observed as janky. In addition, we observe that in some test runs (*e.g.*, $T7$, $T8$, $T9$) the number of falsely flagged events is higher. The rational of this is that such test runs correspond with lower-end devices which make the app to perform worse in general along the whole execution. Finally, some external factors such as network variations, can be responsible to introduce lags at some points of the execution. Nonetheless, DUNE is a powerful tool which can filter out the most relevant events, thus saving precious time to developers.

7.2.4 Threats to Validity

Our results show that the approach can be successfully used to spot UI performance deviations at a fine granularity. Nevertheless, further analyses are necessary to evaluate the efficiency of this approach on different types of apps. For example, we should consider different app categories and sizes. The setting to evaluate the approach (number of scenarios and test runs) is similar to the settings used in previous works [68]) that successfully tackled a similar problem.

In order to better capture the user experience, the performance test scenarios that are used by DUNE are expected to provide a good coverage of the app functionalities and be representative of app usages. With regard to this threat, we could use user-realistic scenarios collected from the crowd, as the ones generated by the MOTIF module (cf. Section 6.1).

In addition, one important threat to construct validity is the choice of devices used in the experiment. Similar to user scenarios, the selected device profiles should be representative of the devices owned by users. One alternative technique to user monitoring is to select random profiles during the test execution and to rely on the similarity measure to better approximate the performance bottlenecks.

For the purpose of our experimentations, we use Android emulators, although the performance of an app could vary when running on a emulator compared to on a real device. Nevertheless, in practice, app developers use emulators to test their apps due to budget and time constraints. Given the large mobile fragmentation, developers use both emulators and real devices. To address this threat, we plan to extend our experiments in a crowd of real devices and perform performance analysis *in vivo*.

The proposed approach is based on predefined thresholds specified by various parameters. The first parameter is the *outlier factor* used to detect performance outliers (cf. subsubsection 5.2.3.1). The second parameter is the *confidence factor* used to learn context rules (cf. subsubsection 5.2.2.3). The calibration of these parameters can impact the results and constitute a potential threat to validity.

Finally, other ecosystems than the Android one should be explored. The conceptual foundations of our approach are independent of Android and the type of performance defect. We only need an oracle of performance and some observation features (in our case, context characteristics). To gain confidence in the external validity of our

approach, more evaluations are needed on other platforms, using different performance metrics.

7.3 Conclusion

This chapter has presented different empirical studies to evaluate the applicability and feasibility of the App Store 2.0 in practice. In particular, this chapter focuses on evaluating the *in vitro* approaches: *crowdsourced checkers* and Dune. The approaches have been studied with real app and real bugs. Overall, the application of our proposed approaches reveals positive results regarding its applicability and feasibility.

Chapter 8

Evaluation of *in-vivo Approaches*

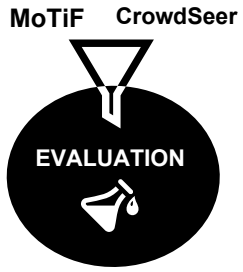


Fig. 8.1 Chapter 8 at a glance

This chapter reports on empirical studies to evaluate the *in vivo* approaches (MoTiF and Crowdseer) in the App Store 2.0. We have implemented supporting tools for Android apps (cf. Appendix **??** for details). In this section, we apply MoTiF and Crowdseer with real app bugs reported by the crowd to evaluate their applicability and feasibility. For each module we present the motivation, evaluation protocol, results, and threats to validity.

The rest of the chapter is structured as follows. Section 8.1 evaluates MoTiF. Section 8.2 evaluates Crowdseer. Section 8.3 concludes the chapter.

8.1 Evaluation of MoTiF

In this section, we report on empirical experiments that we performed to evaluate the applicability and performance of MoTiF.

8.1.1 Empirical Study Design

To evaluate MoTiF we address the following three research questions:

- **RQ1:** *What is the overhead of* MoTiF*?*
- **RQ2:** *Can* MoTiF *identify crash patterns effectively?*
- **RQ3:** *Can* MoTiF *synthesize scenarios that reproduce crashes effectively?*

8.1.2 Dataset

We start by collecting a dataset of real buggy apps. These apps were selected based on their popularity, app category, different size and complexity of functionality, mixture of open-source and proprietary, and the types of crashes that occur. Table 8.1 lists the apps used in the study, with their version, category, size, and type.

Table 8.1 Statistics of the Android apps used in our experiments.

Android App	**Category**	**Size in Kb**	**Type**	**#Users**	**Avg. rating**	**#Traces (#unique)**	**Crash type**
Google I/O 2014	Books	15,060	open	500k–1M	4.3	11 (2)	Device heterogeneity Invalid format
Wikipedia (2.0α)	Books	5650	open	10M–50M	4.4	4 (1)	Network conditions
OpenSudoku (1.1.5)	Games	536	open	1M–5M	4.6	5 (1)	NullPointerException
Bites (1.3)	Lifestyle	208	open	10k–50k	3.2	16 (1)	Invalid format
PocketTool (1.6.12)	Tools	1410	closed	N/A	N/A	16 (1)	Missing resource

To perform the experiments, we use two datasets—*i.e.*, the Chimp and CrowdSourced datasets. Both are based on the 5 Android apps that experience crashes. The Chimp dataset is obtained from the 5 mobile apps using the *Monkey* testing tool (provided by Android). This tool generates pseudo-random user events, such as clicks, touches, gestures as well as system-level events in apps running on a device or in emulators [35]. If the app crashes or receives an unhandled exception, *Monkey* stops and reports the error. To build the Chimp dataset, we first let *Monkey* send 1,000 events to each of our apps. We then repeated this, but this time using 50,000 events. Finally, for the Bites app, we repeated the 50,000 events 49 more times (such that this app in fact had 50 such executions). These three different types of executions will be used across different research questions.

The CrowdSource dataset is obtained through crowdsourcing with students of one computer science lab in Lille and two in Montréal. Since engaging users to participate in crowdsourced experiments is a challenge [159], we designed the experiment as a contest with a prize as incentive for users. The goal of the contest was to try crashing the 5 candidate apps as many times as possible in as many different ways as possible, during a maximum time of 60 minutes. The participants were unaware of the number

and types of crashes in the apps. Eventually, 10 participants engaged in the contest, each of whom was an experienced mobile user.

To run the contest, we provided 5 Android devices with different characteristics to simulate a diverse crowd (cf. Table 8.2). We pre-installed MoTiF and the set of apps under test and borrowed the devices to the participants for the duration of the contest.

Table 8.2 Crowd of devices used in the experiment.

Device model	**Android SDK**
LG-E617G	4.0.3
Samsung Galaxy Nexus	4.0.4
Samsung GT-I9100	4.1.1
Samsung GT-I9100	4.1.2
Samsung Nexus S	4.1.2

8.1.2.1 Exploratory Data Analysis

Before jumping to the research question results, we first want to compare the two obtained datasets in more detail, since the Chimp dataset is obtained automatically, compared to the manually gathered CrowdSource dataset.

In the Chimp dataset (for the 50,000 events sent to each app), the apps *Bites* and *PocketTool* crashed after executing 9,480 events and 33 events, respectively. However, no crash could be found for the other three apps.

In the CrowdSource dataset, on the other hand, the participants were able to generate 52 crashes (each yielding a trace for analysis) across the five apps, distributed across the different devices and Android versions. Table 8.1 (right part) shows, for each subject app, the distribution of crash traces (*i.e.*, crashes) per app and the number of unique crashes amongst them. The observed crashes belong to 4 of the 7 categories of crashes identified in the literature (cf. Table 4.2).

Out of the 52 crashes, we could identify 6 unique crashes, as shown in the table. For the Google I/O app, the crowd discovered two crashes that we were unaware of beforehand. The first crash is a context-related crash occurring only on the devices with Android 4.0.3 and Android 4.0.4. The second crash happens when searching for a word that contains the quote character: ". Since the Google I/O app was developed on purpose by

Google for their annual Android developer conference as an example of best practices (its source code was made publicly available), these two discovered bugs underline the quality of the CrowdSource dataset. Indeed, it demonstrates that even apps that are well-designed and tested, crash in certain contexts. Furthermore, crowd-based crash monitoring seems a valid basis for gathering a wide variety of crashes.

In the remainder of this section, we discuss our findings for the 3 research questions, using the two datasets. For each question, we provide a motivation, approach and findings.

8.1.3 Empirical Study Results

We report the experiments performed to answer each research questions and the findings we obtained.

8.1.3.1 RQ1. Runtime overhead

Motivation: The first phase of MoTiF is the monitoring of an app's execution on a client device. The more overhead monitoring causes, the more users will be aware that MoTiF will be running, and the more it can influence their user experience. Furthermore, users without crashes should not be punished with slow performance due to monitoring. Hence, this question aims to quantify the overhead of MoTiF.

Approach: We study the runtime overhead introduced by MoTiF by monitoring the app executions of the Chimp dataset on the Samsung Galaxy Nexus with Android 4.1.2 and 2 processors. In particular, we used the executions of 1,000 events of the Chimp dataset, then measured the average execution time across the recorded events as well as the average time required to send traces to the server.

Findings: **The mean overhead to log a user event is** 39 ms**.** Due to MoTiF's adaptive logging strategy, MoTiF initially only listens for uncaught exception events. Therefore, the corresponding runtime overhead to store exception events is 0, given that MoTiF logs the exception events only after the app has crashed. Only when an app is suspected to be crash-prone, MoTiF augments the monitoring strategy to log user interactions. As such, the obtained average overhead of 39 ms of the current proof-of-concept implementation is imperceptible to users when interacting with the apps. A response delay < 500 ms is acceptable for users according to the Intel industry

experience values [157]. Nevertheless, additional engineering efforts should be invested to minimize overhead. Different approaches, *e.g.,* code instrumentation, could be explored. Finally, it is important to note that MoTiF distributes its experiments among the different devices available in its crowd, and redistributes them periodically, hence any accidental overhead will even out across different devices.

The median execution time to send crash traces to the server is 666 *ms*. The crash traces are temporarily stored in JSON files in the device memory, until the data is automatically flushed to the remote server for processing. For example, the 50 random traces in the Chimp dataset generated for the Bites app contain 36,603 events and consume 31 MB. Since MoTiF sends the traces to the server only when the device is charging, and the majority of users charge their devices on a daily basis, MoTiF liberates the temporary storage in a short time. Hence, in a typical use scenario, only a limited number of traces will be stored in a device. Since modern devices have several GBs of memory available and incorporate external storage cards with additional memory, the temporal storage of traces in devices is feasible. Furthermore, to minimize the impact on a user's device, only one app is monitored at a given time on each device. Figure 8.2 shows the statistical distribution of the overhead measures.

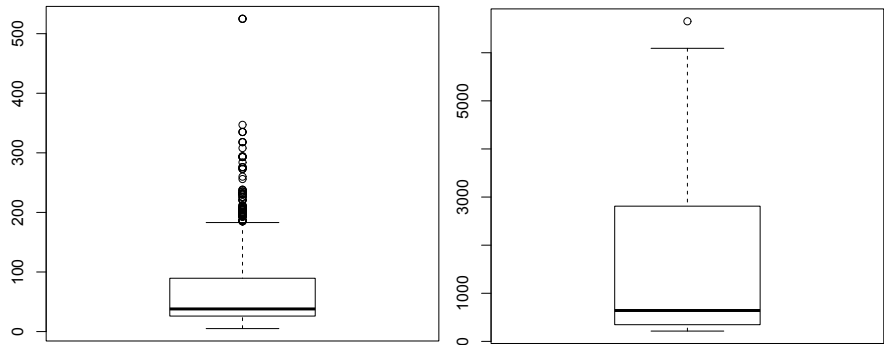


Fig. 8.2 Runtime overhead (ms) of monitoring user interactions (left) and send traces to server (right).

8.1.3.2 RQ2. Identification of crash patterns

Motivation: The second phase in MoTiF is the identification of crash patterns, which is crucial to filter noise from user interaction events and context. The more irrelevant steps MoTiF can eliminate, the more succinct the resulting crash pattern, and hence the less effort is required from developers to interpret the crash patterns. Here, we

Table 8.3 Number of events and compression factor of the crash traces for the CrowdSource (first five rows) and Chimp (last row) datasets.

Android App	**Avg. # Events**	**#Consol. Events**	**Compr. Factor**
Google I/O 2014	22	1	22
Wikipedia	29.5	2	14.75
OpenSudoku	60	8	7.5
Bites (CrowdSource)	56	6	9.33
PocketTool	9.4	1	9.4
Bites (Chimp)	778.79	2	389.40

study the filtering performance of MoTiF for the crash traces, as well as its resilience to noise introduced in the data.

Approach: For each app, we extract the crowd-consolidated traces using MoTiF and measure their compression factor as $\frac{Avg.\ \#events\ in\ crash\ traces}{\#Eventsincrowdconsolidatedtrace}$. The higher this factor, the better MoTiF was able to filter the trace.

Furthermore, in order to assess the impact of noise, we used the 50 executions of the Bites app in the Chimp dataset (each of which crashed the app). In particular, we added these 50 traces to the crash graph of the Bites app obtained from CrowdSource. Since the amount of random crash data from Chimp outweighs the amount of manually generated crash data, this experiment allows to measure how effective MoTiF can deal with noise in the crash data.

Findings: **MoTiF obtains compression factors of 7.5 up to 22.** As shown in Table 8.3, the number of events per trace can significantly differ among apps, since it will depend on the design of the app and the location of the bug causing the crash. However, for all the apps, the total number of events in the crowd-consolidated trace (generated by MoTiF) is smaller than the average size of the original traces. For example, in the *Wikipedia* app, the average size of traces is 29.5, while MoTiF synthesizes a 2-event trace from the crowd data, together with a relevant context: *network disconnection*. Table 8.3 shows the resulting compression factors, which are the highest for the Google I/O app. However, even for the apps with the longest traces (60 for OpenSudoku and 56 for Bites), MoTiF is able to reduce the size of the event trace substantially to 8 and 6, respectively.

In the presence of noise, MoTiF achieved a compression factor of 389.40**.** Indeed, the graph from the Chimp dataset for Bites contains 629 different event nodes and 3,596 relationships among them (extracting the consolidated trace took 267ms).

Whereas the average number of events in the randomly generated crash traces is 778.79, the consolidated crash trace contains only 2 events:
$keyUp(keyCode = 22) -> onMenuItemClick(id = 6)$.

Although this trace reproduces the original crash, we observe that it slightly differs from the consolidated trace synthesized from CrowdSource. This is because the input data in this case contains more randomly generated traces (50) than manually generated ones (16), hence the random Chimp events dominate. However, both 2-event traces are correct and their main difference is that the Chimp traces contain event subsequences that never could be crowdsourced because of the physical limitations of a mobile device.

8.1.3.3 RQ3. Generation of reproducible scenarios

Motivation: The third and final phase of the approach is the generation of a test suite to reproduce crashes. The main challenge here is to generate the same exception types as the original crashes. This is what is evaluated in this question.

Approach: To check if the promoted traces from CrowdSource can reproduce the crashes, we generate the corresponding Robotium tests with MoTiF. We then execute the test cases on the devices and check whether the app crashes again and, if so, whether the same exception types occur.

Findings: **The test cases correctly reproduce the bugs in** 4 **(out of** 5**) apps.** In other words, the execution of the test cases generates the same exception type in the same stack trace location as the original crashes of CrowdSource. Only in the OpenSudoku app, the first consolidated trace failed when trying to reproduce. Closer analysis of the source code revealed that this failure is due to the display of the same dialog box from two different locations in the app. Hence, in the graph, the event was merged into a single node. We plan to further explore such situations to improve the effectiveness of MoTiF. In any case, when we extracted the next most weighted consolidated-trace from the graph, the crash could be reproduced. One benefit of the Crowd Crash Graph is indeed that it will always contain a path that reproduces the crash.

Table 8.4 summarizes the steps extracted from the crowd-consolidated traces to reproduce the context-sensitive crashes found by the participants of the experiment. Each of them is not only compact, but also easy to interpret by developers.

Table 8.4 Candidate traces to reproduce crashes.

App	Events
Google I/O	1) Click on ImageView Search 2) Type “
	1) Click on a Session [Android 4.0.3/4.0.4]
Wikipedia	1) Click on ImageView Menu 2) Disconnect WiFi and mobile data 3) Click on MenuItem “Save Page”
OpenSudoku	1) Click on ListItem position 1 2) LongClick in ListItem position 1 3) Click on MenuItem “Edit note” 4) Click on Button “Save” 5) LongClick in ListItem position 1 6) Click on MenuItem “Delete puzzle” 7) Click on Button “Ok” 8) Change orientation
Bites	1) Click on Tab “Method” 2) Click on context menu 3) Click on MenuItem “insert” 4) Touch text field “Method” 5) Click Button “ok”
PocketTool	1) Click on Button “Level Editor”

8.1.4 Threats to Validity

One important threat to construct validity is the choice of CrowdSource participants which could be biased to young, experienced mobile app users who knew that they had to find crashes. This might not be representative of the typical users of some apps, and hence impact (either positively or negatively) the ability of crowdsourcing to generate certain crash traces. Nonetheless, our experiment is performed under realistic conditions with 5 real apps with different types of crashes.

Although performing a crowdsourcing experiment is challenging, especially taking into account the added difficulty of factoring in different contexts, our approach does not require any critical number of users to work. As soon as MoTiF collects one single trace, it can synthesize a test case to reproduce this trace. However, the larger the number of users (with higher diversity), the more accurate the results that MoTiF produces and the more crash types can cover.

Our approach also has limitations induced by the implementation of the client library. First, for convenience, the apps must have their debug flag enabled to be monitored

by MoTiF. Second, the implementation of MoTiF requires root access and only runs in devices with Android SDKs under 4.2 due to a limitation introduced by the underlying implementation. We plan to make MoTiF compatible with the latest Android versions. To alleviate any potential security risk inherent to the current proof-of-concept implementation, alternative techniques (*e.g.,* bytecode instrumentation, embedding a library in the apps source code) could be used to collect the user traces. Further threats are associated to the way in which we measured the overhead of MoTiF.

Furthermore, the choice of apps and devices are a threat to external validity. Further analyses are necessary to evaluate the efficiency of this approach on different types of apps and crashes. For example, different app categories and sizes of apps should be considered. Finally, other ecosystems than the Android one should be explored.

8.2 Evaluation of CrowdSeer

This section describes the application of Crowdseer in practice to evaluate its applicability and feasibility. Crowdseer has been applied in different real bugs reported by the crowd in Android apps and evaluated with users and developers.

8.2.1 Empirical Study Design

In particular, to evaluate Crowdseer we perform a set of experiments which focus on answering the following research questions:

- **RQ1:** How *effective* is Crowdseer to prevent crashes of Android apps?
- **RQ2:** What do *users* think about the feature deactivation mechanism of Crowdseer?
- **RQ3:** What is the applicability of the approach from the *developer's perspective* in practice?

8.2.2 Empirical Study Results

In the following, we describe the studies we performed to answer these research questions. For each study, we report on the motivation, the experiment design, and the results we obtained.

8.2.2.1 RQ1. Crash Prevention Experiment

The first study focuses on answering `RQ1` to evaluate the effectiveness of CROWDSEER to improve the resilience of apps to crashes. For this study, we conducted an experiment on 10 different apps and real crashes and we checked if CROWDSEER was actually able to prevent those crashes.

8.2.2.2 Experiment Design

We collected a dataset of real app crashes reported by the crowd. We avoided to manually inject crashes in the apps, since we intend to demonstrate real crashes and real apps that are more representative of crashes experienced by users. We limited our search space to open-source apps as we need to instrument the source code in order to integrate the CROWDSEER library.

To select the apps, we inspected issues reported in the issue trackers of open-source Android apps. In particular, we explored the following repositories: *Google Code Archive* [15], *F-droid* [fdr], *GitHub*, and *AOpenSource* [aop]. We also checked the user reviews published in the Google Play Store. In addition, we included apps, which have been studied in previous works [162, 57].

Once we have identified a list of candidate apps, we searched for the source code of the specific version for which the crashes have been reported. Some bug reports lacked the version of the app affected by the crash, so we had to consider several versions of the app to reproduce the crash. To reproduce the crowd-reported crashes, we used the *Monkey* testing tool provided by Android. This tool generates pseudo-random user events, such as clicks, touches, gestures, as well as system-level events in apps running on devices or emulators [35]. If *Monkey* reproduces the crash, we add the app to our app dataset. Otherwise, we discard the app. The final dataset contains 10 apps for which *Monkey* was able to reproduce the reported crash as shown in Table 8.5 (left). All the apps we considered contain real bugs which have been reported by users.

Table 8.5 Experiment results using CROWDSEER to avoid crashes in the benchmark apps[†]

#	**App name**	**Category in store**	**Size (Kb)**	**#Monkey Events before crash without** CROWDSEER	**#Monkey Events with** CROWDSEER	**Crash prevented?**
1	OpenMensa (0.8)	Tools	2,290	3,670	10,000	yes
2	Easy xkcd (3.3.9)	Comics	16,080	1,687	$4,427 \rightarrow 10,000$	yes
3	Ringdroid (2.7.3)	Media	5,450	3,188	10,000	yes
4	ADSdroid (1.5)	Reference	344	3,670	10,000	yes
5	Androimatic K. (1.0)	Communic.	328	1,655	$8,388 \rightarrow 10,000$	yes
6	Dalvik Explorer (3.4)	Tools	148	470	$470 \rightarrow 10,000$	yes
7	Amazed (2.0.2)	Games	44	355	352	no
8	aagtl (1.0.31)	Tools	872	2,146	$3,554 \rightarrow 10,000$	yes
9	NPR News (2.4)	Multimedia	2,260	545	10,000	yes
10	AnkiDroid (2.2.3)	Education	8,500	3,091	10,000	yes

[†]✓ Max. number of events = 10,000

To run the experiments, we used the *Genymotion* emulator [gen]. We install and run the benchmark apps in an image of a device *Samsung Galaxy S4* with Android SDK *4.3*. We chose this device since it is a popular device at the time of writing. For each app in our dataset, we run *Monkey* twice.

The sequence of events generated by *Monkey* is based on a random seed value, which can be set by an optional input parameter. For the same seed value, *Monkey* generates the same sequence of events. We used a fixed seed value in order to reproduce the same scenario multiple times (in particular 12345 selected randomly)[1]. We set *Monkey* to send 10,000 events with $50ms$ time delay between consecutive events (*i.e.*, *throttle* value). In addition, we set *Monkey* to inject 1% of rotation events (since many Android crashes are triggered after rotation due to inadequate management of the Android Activities lifecycle) and to avoid injecting system events (since system events are not involved in the reported crashes)—*i.e.*, start/end call, up/down volume (*pct-syskeys* parameter). The final *Monkey* setting we used in the study is the following:

```
adb shell monkey -p [package_name] --pct-rotation 1 --pct-syskeys 0 -s
                    12345 --throttle 50 -v 10000
```

First, we run *Monkey* with the original app. If the app crashes or receives an unhandled exception, *Monkey* stops and reports the error. Then, we extract the last feature executed just immediately before the crash. Second, we add CROWDSEER into the apps and set it to deactivate the conflictive features extracted in the previous run. Then, we run *Monkey* again with the same configuration to replicate the same sequence

[1]Note that using the same input parameters, occasionally the results can slightly differ because is hardly possible to ensure exactly the same system state across experiments.

of events executed in the first run. Finally, we compare the number of events that *Monkey* completes in both runs.

After deactivating the conflictive feature, the crash should be avoided. This implies that *Monkey* should complete the execution without errors or find a different crash. In the latter case, the number of injected events should be larger than the number of events inserted in the first run. Thus, meaning that the first observed crash is avoided.

8.2.2.3 Experiment Results

Out of 10 **studied apps, our approach effectively avoids the crashes in** 9 **apps**. Table 8.5 summarizes the results of the experiment. For each app, the table shows the app *name* with its *version* (in brackets), and the store *category* to which the app belongs, number of events (out of 10,000 specified) which *Monkey* injected in the first and second runs—*i.e.*, with the original app and finally if our approach successfully prevents the crash. For example, *Monkey* crashed the original `OpenMensa` app at event 3,670 while, after feature deactivation, *Monkey* completes the execution without errors.

In the `Easy xkcd`, `Androimatic`, and `aagtl` apps, CROWDSEER avoids the crash observed in the first run, however it finds a new crash in a different location. We observe that in the three apps, the number of injected events in the second run is higher than the number of injected events in first run ($4,427 > 1,687$, $8,388 > 1,655$, and $3,554 > 2,146$, respectively). This means that CROWDSEER succeeds to prevent the crash and *Monkey* continues the exploration. We manually analyzed the code of these apps to confirm that such apps contain several bugs, which lead to crashes in different locations. The *continuous patch validation* phase of CROWDSEER continuously monitors the patched apps, thus it will detect this situation and it will deploy an additional patch to prevent the second observed crash.

CROWDSEER initially failed to avoid the crash in the `Dalvik Explorer` app. The `Dalvik Explorer` app crashes when clicking on a link in the app. Thus, the candidate feature to deactivate is a link. However, a *link* is not a graphical element of the UI (it is an instance of the `Linkify` class), thus the link lacks an identifier to characterize such element. As CROWDSEER uses *id*'s to identify and deactivate graphical elements, it cannot disable a link. In this situation, CROWDSEER will try with the previous feature contained in the consolidated trace. In the particular case of the `Dalvik Explorer`

app is an item in the menu. Disabling such menu item, it prevents that the user reaches such link.

Finally, CROWDSEER failed to prevent the crash in the `Amazed` app. The `Amazed` app is a game where the user moves the device to direct a ball in the screen. This app uses the accelerometer to work. The app crashes after some movements. For the time being, CROWDSEER does not monitor accelerometer data, thus it cannot deal with this type of crash. In fact, CROWDSEER cannot tackle gesture-induced crashes.

8.2.2.4 RQ2. User Opinion and Preferences Survey

The study aims to evaluate the automatic feature deactivation mechanism from the *users perspective* answering `RQ2`. User satisfaction plays an important role in our approach, given that the underlying goal of CROWDSEER is to enhance user experience with apps. In fact, the higher the user satisfaction with an app, the less risk for developers to loose users in favor of the competitor's apps.

8.2.2.5 Survey Design

We conducted a survey with closed questions in April 2016. The goal is to show users how CROWDSEER works and ask what they think and which option the prefer. To ensure a strong degree of realism, we selected a real popular Android app with a real bug. In particular, we use a version of the Android *Wikipedia* app ($v2.0\alpha$) that contains a crash-inducing bug—*i.e.*, the app crashes when the user tries to save a page and the network connectivity is unavailable[2].

We follow a *within-subjects design* [138] where all participants are exposed to every treatment/method. In our case, every subject is exposed to 1. the original app crash, and 2. the deactivation of the buggy feature to prevent the crash.

We ran the study under two settings: a traditional lab setting with *in situ subjects*, and using a crowdsourcing setting with *crowdsourced subjects*.

Lab Setting: In the lab setting, we asked 33 mobile users to participate in the experiment. The subjects were researchers and master students in Computer Science.

[2]This bug has been fixed in posterior versions of the app.

We performed a case study based evaluation by immersing users in a real application of the approach [161]. First, the subjects had to carry out a real task with the *Wikipedia* app. Afterwards, we asked them to complete a survey.

We provided the subjects with a tablet (*Samsung GALAXY Tab 4* running Android 5.0.2), which had the app under test installed. The experiment started with a *scenario description* which outlined the steps that the subjects must carry out with the subject app. This includes 5 steps: 1. Launch the *Wikipedia* app, 2. Search a page, 3. Deactivate the *WiFi* connection, 4. Open *Menu*, 5. Click on *'Save page'*.

Every subject had to repeat the scenario twice. The first time, the app crashes after clicking on *'Save page'*. The second time, the app disables the *'Save page'* option to prevent the crash and shows a pop-up message to inform about the unavailability of such feature.

Afterwards, participants proceed with our survey to evaluate the feature deactivation mechanism. The form used in the experiment is available in Appendix A. The form had four questions using 5-Likert scale [138], one multiple-choice question to know the preferences of users, and finally one open question where users can provide any additional feedback or comments. Overall, the questionnaire needed 8 minutes to answer.

Crowdsourcing Setting: The use of students and researchers as experimental subjects and the size of the sample (33 subjects) might not be considered as representative of the population of mobile users and it constitutes a potential threat to validity. To alleviate this threat, we used a crowdsourcing platform to engage a broader and more heterogeneous participation in our study.

Experiment subjects: A crowdsourcing platform, such as *Amazon Mechanical Turk (MTurk)* [Amazon Mechanical Turk]), is an online service from which anyone can publish micro-tasks and recruit contributors all over the world to complete the tasks for a small monetary reward. Tasks typically require little time and effort to realize. At the time of writing, *MTurk* is the most popular crowdsourcing platform, but it disallows to request tasks from outside of the US. Thus, we use an alternative service, the *CrowdFlower* platform [cro], to recruit 600 crowdsourced subjects to participate in our study.

We design the experiment as a two-steps job in *CrowdFlower*. First, we provide a video that illustrates the two usage scenarios that *in situ* subjects had to manually

perform with a real device[3]. We add annotations in the video to ensure that participants understand the scenarios. After watching the video, participants had to answer the same questionnaire as provided to *in situ* subjects. However, this questionnaire additionally included a list of demographic questions to characterize the crowd who participated to our study. The demographic questions included: age, country, IT background and smartphone usage. More information about the job published in *CrowdFlower* is available in Appendix A. *CrowdFlower* ensures that each participant only performs the job once.

We designed the job to be completed in less than 10 minutes, and we set a prize of $0.15. To incentive participants to carefully complete the survey, we announced a bonus of $0.15 in addition to the prize of the job.

When designing a crowdsourced survey, we have to consider additional threats as we risk to recruit fraudulent participants, who randomly answer the questions. To alleviate this threat, we incorporated mechanisms to ensure the quality of answers:

- *Validation questions.* We included along the questionnaire two *test questions* to detect fraudulent participants. In particular, these questions were: *Which option is disabled in the app shown in the video?* and *Which smartphone do you use currently?* If a participant fails the first question, or enters random text that does no correspond with any smartphone, we classify such an answer as invalid and excluded it from the evaluation.
- *Time threshold.* We set 5 minutes as the minimum time to complete the job. *CrowdFlower* rejects the jobs that are completed faster than this threshold.
- *Click check.* We add a validator to ensure that the video link was clicked. However, this check cannot ensure that the participants watched the video, we can only be sure that they accessed to it.

We deployed the job to the crowd in April 2016.

We first launched the job in *CrowdFlower* to 100 participants as exploratory data analysis phase. *CrowdFlower* characterizes its crowd workers in 3 levels regarding their experience and the overall quality of their previously completed jobs. A *level-3* crowd implies the most experienced and successful participants. We decided to publish the task only to *level-2* crowd and we obtained the requested 100 answers in 40 minutes. Then, we analyzed the answers and rejected the ones who failed our validation test.

[3]The scenario video can be watched at https://www.youtube.com/watch?v=za2dhlRqhwI

As a result, we kept 71 answers, excluding 29% of the answers. Afterwards, we made adjustments in the settings of the job with the aim of increasing the quality of the answers, and we re-launched the job.

Second round. For the second round, we select only *level-3* crowd, while targeting 400 participants. Note, that in each new round *CrowdFlower* bans participants who have already completed the job. We collected the 400 responses in less than 2 hours. After sanitizing the results, we kept 330 answers—thus excluding 17.5% of fake answers. We observed how increasing the level of the target crowd (from level 2 to 3) reduces the number of fake answers.

Third round. Finally, we made another adjustment to increase the quality of the answers. We continued targeting only crowds of level 3, but in this case we reduced the price of the job, from \$0.15 to \$0.10 and increased the amount of the bonus (from \$0.15 to \$0.20). We re-published the job to 100 more participants and we collected the last 100 answers in 90 minutes. We kept 94 answers—*i.e.*, we reject only 6% of answers. Fraudulent participants tend to apply for the tasks with higher reward.

Finally, we obtained a sample of 495 answers from crowdsourced subjects.

8.2.2.6 Survey Results

Surveyed users prefer deactivating buggy features over suffering from crashes. Figure 8.3 provides details on the experiment results we obtained when evaluating the approach with end-users.

Overall, the evaluation reveals an high satisfaction with our proposed approach. First, 72.2% of users report app crashes as *annoying* or *very annoying*. The 61.5% **of users agree that a disabled feature is less annoying than a crash** (45.6% agree and 18.0% strongly agree). On the contrary, 12.7% of users disagree. There are also 23.7% of users who do not agree neither disagree. We observe that the amount of neutral users (*i.e.*, users that do not agree neither disagree) is larger among the crowdsourced subjects, 25.1% against the 3.0% in the *in situ* subjects. The rational of this could be that some crowdsourced subjects may have misunderstood the formulation of the question, while *in situ* subjects can ask for clarifications. In fact, this happened during the study, several *in situ* subjects asked for confirmation regarding their understanding of some questions. This is one of the risks when using crowdsourced participants for user evaluation [101], some questions can be misunderstood and the participants lack

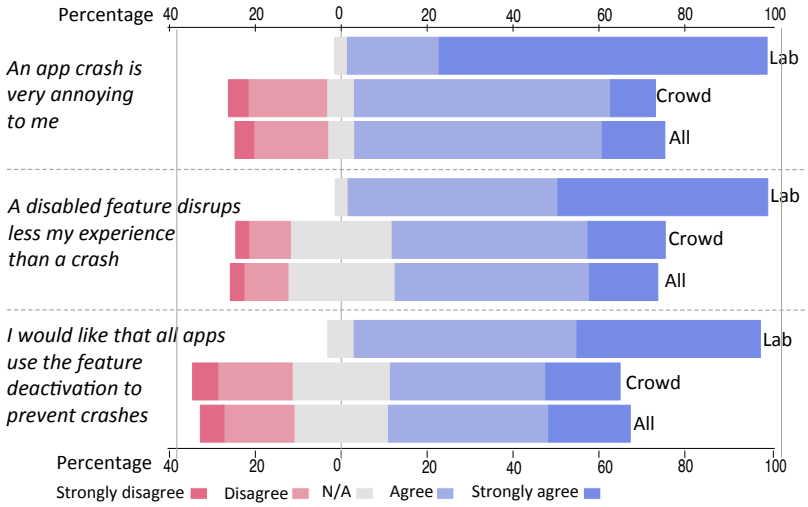


Fig. 8.3 Results of the User Survey (N_{lab}=33, N_{crowd}=495, N_{total}=528).

the opportunity to clarify. To alleviate this threat, we include textual captions along the video to describe what is happening in the app.

Finally, the **56.3% of users would like that all the apps in their devices incorporate the deactivation mechanism to prevent crashes**. While the 21.8% disagrees on that. We observe that any *in situ* subject disagree on this statement, and there are only 2 users that neither agree or disagree. Since *in situ* subjects had to perform the scenario themselves with a real device, their degree of realism is higher, then they perceived the crash more annoying than if the crash was watched in a video.

The survey includes an additional question to determine user preferences regarding the type of prevention action (*e.g.*, feature deactivation) instead of a crash. **Users repeatedly requested for an explanation about the feature deactivation.**

Figure 8.4 summarizes the reported user preferences. *'Displaying a message'* (which informs about the unavailability of a feature) and *'Disabling the feature'* are the most voted options. Followed by the combination of both—*i.e.*, *message* and *disabling*. In addition, the 10.6% of users (56 users) prefer to totally hide the feature in the app if it is unavailable. Furthermore, three-quarters of *in situ* subjects commented that they would like to know the reason why the feature is disabled. In the particular case of the *Wikipedia* app (used as case study), a message that explains that the *'Save page'* is disabled because the network connection is unavailable. Then, they can do a corrective action to make the feature available again. As a future work, we plan to enhance the

messages to include information about the reasons of the deactivations. We also plan to study mechanisms to incorporate recommendations about the actions that users can take to avoid the issues when possible.

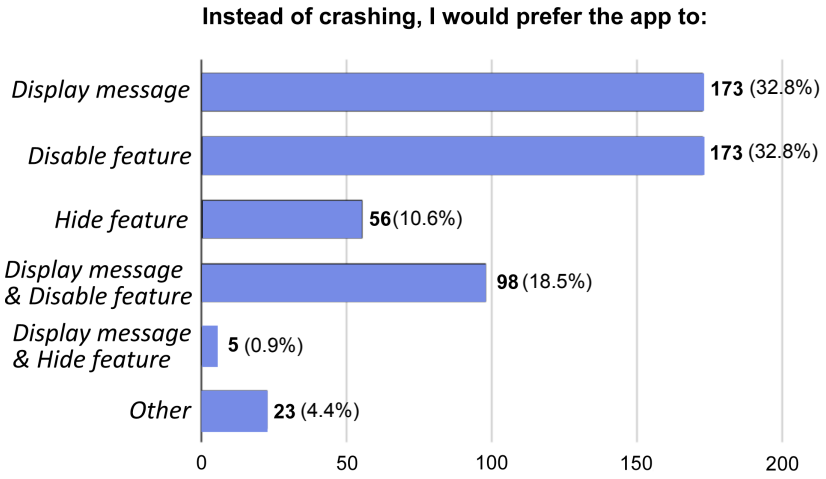


Fig. 8.4 Preferred options instead of crash (N=528).

In addition, 9 subjects prefer other solutions. We asked an open-ended question to *in situ* subjects to have the opportunity to further discuss with them. We summarize alternative preventive mechanisms users suggested:

- *Display an information icon next to the feature.* When the user clicks on the icon, inform that such feature has been observed as buggy and will potentially crash the app. Then, let the user to decide if temporarily disabling the feature or taking the risk.

- *Only show a message after the first time the user taps the feature.* Thus, if such feature is rarely used, the user will not be disturbed with irrelevant notifications.

- *Disable the feature, but if the feature is not fixed in a short time, then hide the feature.*

- *Recommend actions to make the feature available.* For example, after clicking in a feature inform the user that the WiFi must be activated first.

8.2.2.7 RQ3. Developer Feedback Interview

The goal of the last study is to *qualitatively* evaluate the applicability of our approach from the developers' perspective. For this purpose, we conducted several semi-structured exploratory interviews with experienced mobile app developers. This experiment aims to answer RQ3.

Table 8.6 Overview of the interviewed Android developers

ID	EXP. (YEARS)	COUNTRY	ROLE	TYPE OF APPS
D1	7	Germany	Research assistant	Games, Location-based, Context-based (sensors)
D2	6	Germany	Sw developer, Sw architect	1 Game, Social messaging, Security, Language learning
D3	3	Germany	Developer, Tester, UI design	Health care (diabetics)
D4	4	France	Sw developer, Sw architect, Self-employed	Business apps
D5	11	Canada	Android developer	Games, Radio stations, Business apps
D6	10	Canada	Android developer	French retirement system app, Business apps

8.2.2.8 Interview Design

Developer recruitment: We interviewed 6 experienced mobile app developers (from 3 to 11 years of experience) who have been working in companies developing Android apps. We selected developers from Europe and America, with different experiences, from companies located in different countries, and with different roles in the companies, to minimize potential threats of external validity. Table 8.6 provides an overview of the participants.

The goal of the interviews was to explore the applicability of CROWDSEER from the developer's perspective. The interview focused on exploring *how*, *which*, and *why* questions. We structured the interview in four sections: 1. *Background and project context*, 2. *Current crash detection and fixing practice*, 3. *Feedback to* CROWDSEER, and 4. *Overall satisfaction with our approach.* We prepared a list of questions as a guidance, but the interviews were open to discussions with developers. The prepared interview is available in Appendix B.

The interview started with a presentation which provides general information about the approach and the supporting infrastructure. After a short demo of the approach, we proceed with the exploratory questions. Each interview had a duration between 45 and 60 minutes. The interviews were performed during March and April 2016. 3 interviews were performed physically, while the others by video conference.

8.2.2.9 Interview Results

After completing the interviews, for each question, we analyze and compare the answers of the different developers. We divide the interview in topics and discuss the

observations in each of them.

General Crash Detection and Fixing practice: In the first part of the interview, we explore the current practice of crash detection and fixing to gain insights and understand the needs of developers.

To detect when users experience crashes, D2, D3 and D4 developers use third-party libraries—*crash reporting tools*—embedded in their apps (in particular *Crashlytics*, *Sentry* and *New Relic*). The remaining 3 developers rely on the feedback provided by the Android platform via the *Google Developer Console* in the *Google Play Store*. In particular, D1 declared that: *"the* Google Developer Console *provides enough information and I do not need additional tools"*. While, developers D5 and D6 refuse to add any third-party library to their apps because they loose the control of the apps and makes the apps dependent of a third-party.

Developers D5 and D6 (the most experienced developers) commented that their apps are highly stable and hardly ever crash. D5 stated that: *'I like to say that I don't get any crash'* and D6 said: *'Apps never should get to a point of crash'*. They argued that, after many years developing apps, they anticipate potential points of crash and prevent crashes beforehand. In addition, they heavily test their apps previous to release, including beta-testing to a reduced group of users before making the final release publicly available.

All developers agreed that *stack traces* are the most valuable artifact when trying to fix a bug. The stack trace is the starting point to figure out the root cause of a problem. This aligns with the findings of previous research in desktop programs [139]. Furthermore, **developers use user reviews posted in the store to add new features, but not for bug-fixing**. D3 sketched: *"user reviews lack of useful information to reproduce and fix problems"*.

In addition, the 6 developers highlighted that **the reproduction step is the most challenging when facing a crash due to device fragmentation and the lack information about the context of bugs**. D3 sketched: *"We only have 5 different devices to test. It's hard to reproduce because there are many device-specific problems"*.

Feedback on CROWDSEER*:* In the second part of the interview, we explored developers' opinion about CROWDSEER, their acceptance of the approach, and their preferences.

All developers agreed that Crowdseer is a powerful approach to prevent crashes. Developers D2 and D4 highlighted as strength of the approach the possibility to quickly update apps. D3 sketched: *'I love the possibility to quickly update apps in case of critical issues without waiting for the long process of publishing a new app release in the store, which can take from hours to days in some cases'.*

All developers requested means for hooking the decision process of Crowdseer. All the developers manifested that they would like that CROWDSEER notifies them before the feature deactivation is performed. They want to always keep the control of their apps. They would like to receive a notification informing about the crash and the suggestion of the feature to deactivate, then they decide to apply it or not. However, developers D2 and D3 stated that if the crash is very critical—*i.e.*, affecting a large community of users in a short time—then the feature deactivation can be applied and sending a notification to them informing about preventive action adopted. In such a case, they would like to be able to revert the change if they disagree.

Regarding the notification channel that CROWDSEER should use to communicate with developers, they prefer notifications via a *Slack* channel or e-mail. They also mention that the notifications should include a link to directly apply the feature deactivation. Additionally, developers would like that CROWDSEER creates a new issue automatically in their Issue Tracker (*e.g.*, in *Github*).

All developers remarked the fast and easy ability to integrate Crowdseer in their apps. All developers liked the current implementation of CROWDSEER—*i.e.*, a library to embed in apps, and a web interface to update apps on-the-fly. However, developers D5 and D6 refuse to add any third-party library into their apps. D5 claimed: *"we used to use a library for reporting crashes and each time that the Android SDK updated, that library produced a high amount of issues in our app"*. D6 said: *"we were using some third-party libraries which make our apps consume up to 40% more battery"*. They said that they use third-party libraries during *beta-testing* to get insights about the performance of their apps; but when they release the final version to the public, they remove all the third-party libraries.

Regarding the web interface of CROWDSEER, developers D3, D4, and D6 mentioned that there should be a mechanism to automatically test the apps after the feature deactivation to ensure that such a deactivation does not introduce any additional crashes or undesired behavior. Furthermore, D4 suggested to keep an history of patches that can be applied or rejected at any moment. Also, there should be some mechanism to order patches by priority, regarding how many users are affected by the crashes.

Five developers found the deactivation mechanism really useful to reduce crashes in the apps. While, the reminder developer cannot agree neither disagree if he does not test it. D6 mentions that some crashes appear for operations running in background and CROWDSEER cannot prevent such crashes. We are aware that CROWDSEER cannot prevent all types of crashes in mobile apps.

Four (out of six) developers would like to incorporate Crowdseer in their apps now. Remark that one of these developers contacted us some days after the interview asking to use our library. On the contrary, developers D5 and D6 need more information before accepting to incorporate the approach. In particular, these 2 developers are the same developers who declared that they rarely experience crashes and dislike to integrate third-party libraries in their apps. They said that they need to test CROWDSEER before to include it in their developed apps. D6 said that: *"for me security is a very critical concern, I need to be sure that any malicious app can attack your library and then get the control of my app"*. As future work, we plan study mechanisms to enforce the security of CROWDSEER.

We extracted additional insights from the interviews. From the end-users perspective, the approach should include a feedback to them and explain why some features are unavailable, to help users understand and avoid frustration. This aligns with the feedback extracted from users' survey. Developer D2 suggested that defective apps could incorporate a small information icon in the top informing about some features have crashed for other users, and then the user can accept or dismiss that the app proceeds with the feature deactivation.

In addition, the approach could recommend different solutions to prevent the crash (*e.g.*, disable a menu item or disable the whole menu) and developers decide which is the most adequate action to be applied depending on the type of app and the location of the crash. D4 requested that it would be nice to incorporate mechanisms to recommend fixes to the crashes. CROWDSEER could be extended to incorporate automatic software repair techniques [115] in order to automatically apply patches fixing the crashes in some specific situations.

To sum up, developers have in general a positive opinion about the approach and have provided valuable feedback regarding the adoption of CROWDSEER in practice. In contrast, they request mechanisms to control the decision process of CROWDSEER. We take all their insights into consideration to improve the approach.

8.2.3 Threats to Validity

We describe the threats that could potentially limit the validity of our results and the mechanisms we apply to mitigate them.

Construct validity. The benchmark of apps used in the evaluation constitutes a potential threat to construct validity. First, this sample only includes open-source Android apps. We use a benchmark of real apps, which contain real bugs, in order to ensure that the crashes are representative of real crashes faced by users in practice. The apps belong to different categories and sizes, and some of them have also been used to evaluate prior approaches related with Android testing. Nevertheless, more experiments with more apps and types of crashes are necessary to gain confidence on the results.

Internal validity. The selection of users and developers is a potential threat of internal validity. To mitigate this threat, we consider an heterogeneous sample with people from different countries, experiences and background, to gain confidence that the participants are representative of the population in general. First, we perform the study with users *in situ*. Since these users are researchers and students, there is a threat that they do not represent the population of mobile users in general. To alleviate this threat, we extended the experiment with crowdsourced participants from different countries, ages, and background. Thus, we increased the size of our sample population, from 33 participants to 528 participants.

By using crowdsourced participants, we alleviate the threat of sample size and heterogeneity, but we also introduce some additional threats. First, we risk to consider fraudulent users that complete the task randomly. We adopted several measures—*i.e.*, test questions, time threshold, click check—to alleviate this threat. However, *in situ* subjects can also randomly answer the surveys. Second, we risk that participants misunderstand questions, and they do not have the opportunity to clarify as *in situ* subjects have.

We design the user study with high degree of realism, by making users to see the approach running in practices, using a real device and a real bug in a popular app. However, the results can be biased because of the app selected for the study—*i.e.*, the Wikipedia app.

External validity. To generalize our findings, more experiments with a larger number of users and developers are necessary. To alleviate this threat, we used a crowdsourcing

platform to recruit a bigger and more heterogeneous sample of subjects. Similarly, we select developers with different backgrounds, who work in different types of companies, have different experiences, and live in different countries.

8.3 Conclusion

This chapter has presented different empirical studies to evaluate the applicability and feasibility of the App Store 2.0 in practice. Our experiments demonstrate the effectiveness of the feature deactivation mechanism to prevent crashes and that users are ready to accept the automated feature deactivation mechanism in practice. In fact, they prefer our solution instead of crashing. However, users demand explanations about the preventive actions taken. On the contrary, developers have concerns about loosing the control of their apps and request to intervene in the prevention mechanism.

Part IV

Final Remarks

Chapter 9

Conclusions and Perspectives

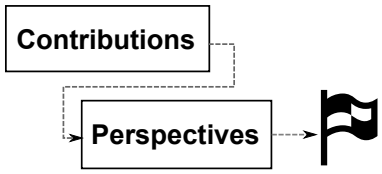


Fig. 9.1 Chapter 9 at a glance

With the proliferation of mobile devices, the development of mobile applications has gained a huge popularity over the last years. The reputation of a mobile app vendor is crucial to survive amongst the ever increasing app market competition. Hence, detecting and preventing quality issues, in particular crashes and unresponsive apps, has become a major goal for app vendors.

This thesis addresses the development of a new generation of app stores—App Store 2.0. For this purpose, we have introduced a crowdsourced framework for extending current app stores with advanced services to assist app developers to automatically improve the quality of mobile apps. This research combines: *app store analysis*, *debugging*, *crowd monitoring*, and *app testing*.

This chapter summarizes the contributions of this thesis and discusses perspectives for future research. The last chapter is structured as follows. Section 9.1 summarizes the contributions of this research. Section 9.2 states future research lines to extend this thesis. Finally, section 9.3 concludes this manuscript.

9.1 Contributions

The main contribution of this research is a **crowdsourced framework to engineer a new generation of app stores**. Since app stores already have access to different types of crowds—*crowd of apps*, *crowd of users*, *crowd of devices*—, our key insight is to make them exploit the wisdom of those crowds to generate actionable feedbacks. The App Store 2.0 includes the capability to assist developers to deal with potential errors and threats that can affect apps prior to publication or even when the apps are in the hands of end-users. In particular, this thesis focuses on alleviating *crash-inducing bugs* and *UI janks* in mobile apps. We have demonstrated that app stores can leverage the *wisdom of the crowd* to improve the quality of mobile apps. This contribution has resulted in three publications: [75], [74], [IEEESw].

The proposed crowdsourced framework comprises 4 software engineering approaches, as well as supporting tools, to produce 4 actionable insights (risk reports, performance reports, reproducible scenarios, and app patches) from crowdsourced feedback. In Chapter 1, we formulated four research questions. In this section, we review these questions and discuss the specific contributions which complement the main contribution:

An approach to predict app crashes before executing apps. This contribution relates to *research question 1: Is it possible to predict occurrences of mobile app bugs from the feedback returned by the crowd?* To answer this question, we introduced an approach to predict potentially buggy apps by using correlations between permission patterns (requested by apps) which are frequent among apps reported as buggy through user reviews. The proposed approach leverages the *wisdom of apps*. In particular, we combine unsupervised and supervised machine learning on top of user reviews and app permissions. We conclude that certain types of bugs can be anticipated with certain measure, before executing apps, by exploiting crowd feedback. As a result, a publication has been derived from this contribution: [78].

A crowdsourced approach to reproduce context-related crashes. The second contribution connects to *research question 2: Is it possible to faithfully reproduce in vitro mobile app bugs experienced by users in the wild?* To answer this question we propose MoTiF. MoTiF exploits the *wisdom of users* and *wisdom of devices* to identify UI sequences and contexts which induce crashes. In particular, MoTiF monitors app usages and contexts, analyzes crowdsourced crash logs, groups failures, and constructs minimized black-box reproducible scenarios from such crowdsourced

data. Developers can then use the generated scenarios to automatically reproduce crashes *in vitro*. This contribution has resulted in a publication: [77].

An automated crash prevention approach for mobile apps. The third contribution relates to *research question 3: Is it possible to prevent recurrences of mobile app bugs in vivo?* To answer this question we devise CROWDSEER. CROWDSEER exploits the *wisdom of users* and *wisdom of devices* to identify buggy functionalities and contexts under which crashes arise. CROWDSEER contributes two patching strategies for mobile apps. The former strategy learns from previous crashes reported by the crowd and patches apps to mute unhanded exceptions. The latter strategy diagnoses the UI feature and context triggering the crash, and immediately deactivates crash-triggering feature on-the-fly, without requiring users to re-install apps. As a result, two publications have been produced from this contribution: [75], [ASE-journal].

A context-aware approach to detect UI lags in mobile apps. The fourth contribution relates to *research question 4: Is it possible to exploit the diversity of the crowd to improve user experience with mobile apps?* To answer this question, we propose DUNE. DUNE leverages the *wisdom of devices* to automatically identify UI performance deviations among different versions of an app and heterogeneous contexts. DUNE monitors app usages, UI performance-related metrics, and contexts. Using this information DUNE spots specific UI events and context properties (*e.g.*, a specific SDK version) that potentially trigger janks. This contribution has resulted in a publication: [76].

Table 9.1 summarizes the contributions of this thesis and the publications that support them.

Table 9.1 Relationship between contributions and publications

Contribution	**Publication**
Crowdsourced framework	[75]
	[74]
	IEEESw (under review)
Crash prediction (Checkers)	[78]
Crash reproduction (MOTIF)	[77]
Hot-patching (CROWDSEER)	[75]
	ASE-journal (to submit Nov. 16)
UI jank detection (DUNE)	[76]

To conclude, the aforementioned contributions demonstrate that by leveraging the wisdom of the crowd, app stores can orchestrate the maintenance process of mobile apps in an automated way. This contributes to reduce human effort and time which is a crucial factor for the success of apps in the app market.

9.2 Perspectives

In this dissertation, we have presented our vision of App Store 2.0 and our solution towards making it a reality. However, this research opens new research directions. In this section, we present short-term and long-term perspectives for the continuation of this research.

9.2.1 Short-term Perspectives

In this section, we describe the immediate continuations of this work. In particular, for each module of the App Store 2.0, we envision several extensions.

9.2.1.1 Dealing with run-time permissions

Traditionally, Android users need to grant access to all the requested permissions by apps at installation time. Starting with Android 6.0 (API level 23) [37], Android introduced a new run-time permission model which gives the app's permission control to users. Under the new permission model, users grant and revoke app permissions at run-time. For example, the user can give access to the camera in an app, but disprove access to the device location. When some permissions are revoked, apps must be able to continue working with limited features.

The new permission model poses new challenges for app developers who need to test apps under additional conditions. In particular, various combinations of granted and revoked permissions need to be tested, in order to ensure that apps can handle all permission configurations without leading to crashes.

We plan to extend the *permission checkers* presented in this thesis with additional *run-time permission checkers*. Prior to release, the *run-time permission checkers* can stress different permission configurations to anticipate crashes related with specific permission combinations. After releasing apps, the *run-time permission checkers* can

learn from conflictive permission configurations observed from the crowd of users. In other words, the *run-time permission checkers* can learn permission configurations which have lead to crashes. The permission knowledge extracted from the crowd can be translated into several actionable insights. First, a *permission configuration report* for app developers in order to fix their apps. Second, *recommendations to users*, which inform users that if they revoke a specific permission A the app will potentially crash, unless a permission B is activated. Third, *patches* will be generated to prevent that other users face a previously observed permission combination at run-time.

9.2.1.2 Improving Dune's performance

Regarding DUNE, we plan to study new models to improve the performance of DUNE. In particular, we focus on the fine-grained detection mechanisms (UI level). For this purpose, we will explore different models and algorithms available into the Kevoree Modelling Framework [98]. These models have already been successfully used in other domains, such as the IoT (Internet of Things) [118].

9.2.1.3 Reaching Monitoring Trade-offs

The current monitoring strategy of the APP STORE 2.0 is designed to minimize user disturbance and runtime overhead. First, only one app is monitored in each device. Second, the monitoring of user interactions only starts after a crash is observed and ends as soon as the store generates an actionable insight. Finally, the monitoring is redistributed periodically among the devices in the crowd. We intend to extend and complement these measures to minimize the invasiveness of the monitoring and the runtime overhead even more.

We plan to investigate further mechanisms to reach trade-off between the amount of collected data and the reproducibility rate of MOTIF. We will analyze different strategies to reduce the amount of data collected while still being able to extract relevant knowledge to isolate bugs. For example, we could consider only logging relevant context data depending on the types of the apps (instead of collecting all context data always). As an illustration, in location-based apps, the GPS context will be relevant. We will also analyze if we can spread different types of data collected in different devices and the resulting aggregation can still report insightful knowledge.

9.2.1.4 Enforcing privacy-preserving mechanisms

Although the crowd can help developers to deal with current challenges of the mobile app development process, it also exposes additional challenges, such as user privacy. The proposed monitoring solution includes privacy-preserving mechanisms as part of its design. First, the monitoring block applies anonymization [30] to ensure user anonymity. In addition, our implementation assigns a pseudo identifier, which identifies each app execution in a specific device. Such an identifier cannot reveal the original device identifier (which could expose user identity).

To enforce privacy, we plan to explore and incorporate a *decentralized dissemination strategy* [142]. Furthermore, App Store 2.0 needs to establish privacy policies and reach trade-offs between developers, stores and users to fulfill the needs of all the parties. We will set privacy policies which enable end-users using App Store 2.0 to configure their privacy preferences. For example, granting access to specific types of data during the data collection, and the phases of the process where they volunteer to participate.

9.2.1.5 Dealing with input-induced crashes

Input-induce crashes are crashes which manifest when the user inputs text in an app (mainly through *TextFields* or *TextAreas*) in an incorrect format, and the app inefficiently processes inputs. For example, inserting text when numbers are expected, leaving a field empty, inserting special characters, etc. In such situations, the current implementation of Crowdseer generates a patch which consists of disabling the *TextField*. Nevertheless, a more suitable prevention action would be to disable the *TextField* only when the 'conflictive' input is inserted.

We propose to further explore this type of bug and to enhance the prevention strategies applied by Crowdseer. To provide smarter prevention mechanisms, we will again exploit the *wisdom of the crowd* to learn which is the minimum input sequence that induces the crash. For this purpose, *input minimization* techniques will be explored [164]. At the same time, privacy mechanisms will be considered to deal with inputs. Finally, different text processing techniques will be studied.

9.2.1.6 Recommending corrective actions for end-users

To prevent the emergence of crashes, the current strategy of CROWDSEER deactivates crash-triggering features. From the user study performed as part of the evaluation, we discovered that end-users request explanations regarding the deactivation.

Hence, we plan to extend CROWDSEER to providing explanations as part of the patches. We will enhance the messages (which CROWDSEER shows to users) to include information about the reasons of the deactivations. For example, *'The menu SAVE has been disabled because the network is unavailable'*. We also plan to study mechanisms to incorporate recommendations about corrective actions that users can take to avoid the issues when possible. For example, if a crash is induced by a network unavailability, a message could notify that the network should be turn on to access a specific functionality.

9.2.2 Long-term Perspectives

In this section, we present additional research directions we would like to tackle on the long term.

9.2.2.1 Embedding security mechanisms

As part of the evaluations of CROWDSEER, we performed a series of interviews with experienced app developers. From this study, we found that security is an important concern for developers to use our approach. For example, our system needs to ensure that any malware app can attack our system and get control of apps in the wild. As future work, we will study security mechanisms and embed security measures to strengthen the approach.

9.2.2.2 Incorporating further automatic repair strategies

Current app stores only have access to the bytecode of apps. Nevertheless, the APP STORE 2.0 could enable developers to upload additional artifacts such as apps' source code and test suites. Then, allowing for more powerful repairing techniques. We will explore and integrate existing automatic software repair approaches, such as

Nopol [158]. Nopol is an automatic software repair tool for Java which repairs condition bugs (if condition, missing precondition) in Java code.

9.2.2.3 Dealing with other types of bugs

In Chapter 4, we discussed different types of bugs in mobile apps. The conceptual foundations of the presented approach could be extended in order to tackle other types of bugs, such as energy leaks and memory bloats.

9.2.2.4 Addressing other mobile platforms

The solution proposed in this thesis focuses on Android because currently such OS accounts for 81.61% of all global smartphone sales [146]. However, we would like to extend our solution for other mobile platforms. More specifically, to the iOS platform, which is the second most used mobile platform worldwide.

9.2.2.5 Providing incentive mechanisms

Crowdsourced systems need mechanisms to foment the end-users willingness to contribute to such system [159]. We will investigate incentive mechanisms to be included in the App Store 2.0 to encourage users to participate in the debugging process.

9.2.2.6 Adoption in practice

To demonstrate the feasibility of the proposal, we have implemented a set of prototype tools to support the different approaches which compose the App Store 2.0. We plan to invest engineering efforts to convert the prototypes into mature tools. The final goal is to transfer this research to industry. Finally, we will evaluate the approach with larger crowds of devices and apps.

9.3 Final Conclusion

To close this manuscript, a quote stated by Helen Keller synthesizes the main idea of this thesis:

> *"Alone we can do so little; together we can do so much."*
>
> —Helen Keller

This quote remarks the value of collaborations to successfully achieve any goal. Exploiting the *wisdom of the crowd* from app stores brings new opportunities to improve mobile apps. This thesis evidences the value of the diversity of the crowd to face inherent challenges to the mobile ecosystem.

References

[adb] Android adb. http://developer.android.com/tools/help/adb.html.

[2] Android Developers. Activities. http://developer.android.com/guide/components/activities.html.

[60f] Android Developers. Android Performance Patterns: Why 60 FPS? https://www.youtube.com/watch?v=CaMTIgxCSqU.

[4] Android developers guide. Input events. http://developer.android.com/guide/topics/ui/ui-events.html.

[dum] Android dumpsys. https://source.android.com/devices/tech/debug/dumpsys.html.

[log] Android logcat. http://developer.android.com/tools/help/logcat.html.

[7] Android UI Overview. http://developer.android.com/guide/topics/ui/overview.html.

[aop] AOPensource. http://aopensource.com.

[cra] Crashlytics. https://try.crashlytics.com/.

[cro] CrowdFlower. http://crowdflower.com.

[11] ExoPlayer. http://developer.android.com/guide/topics/media/exoplayer.html.

[12] ExoPlayer Adaptive video streaming on Android (YouTube) . https://www.youtube.com/watch?v=6VjF638VObA.

[fdr] F-droid. https://f-droid.org.

[gen] Genymotion. https://www.genymotion.com.

[15] Google Code Archive. https://code.google.com/archive.

[16] Google Play installs reached 65 billion last year. https://techcrunch.com/2016/05/18/google-play-installs-reached-65-billion-last-year/.

[k9-] K-9 issue report. https://github.com/k9mail/k-9/issues/643.

[k9] K9-Mail Android app. https://play.google.com/store/apps/details?id=com.fsck.k9&hl=en.

[GPU] Profiling GPU Rendering Walkthrough. http://developer.android.com/tools/performance/profile-gpu-rendering/index.html.

[rob] Robotium. https://code.google.com/p/robotium.

[spa] Space Blaster Android app. https://play.google.com/store/apps/details?id=com.iraqdev.spece&hl=en.

[tes] Testdroid. http://testdroid.com. [Online; accessed Jan-2016].

[Tes] Testing Display Performance. http://developer.android.com/training/testing/performance.html. [Online; accessed Mar-2016].

[Sys] Testing Display Performance. http://developer.android.com/tools/help/systrace.html.

[lis] The Big List of App Stores for Developers. http://appindex.com/blog/list-app-stores/http://appindex.com/blog/list-app-stores/. [Online; accessed June-2016].

[ACRA] ACRA. http://www.acra.ch/.

[27] Adams, B., Bellomo, S., Bird, C., Marshall-Keim, T., Khomh, F., and Moir, K. (2015). The practice and future of release engineering: A roundtable with three release engineers. *IEEE Software*, 32(2):42–49.

[28] Adams, B. and McIntosh, S. (2016). Modern release engineering in a nutshell–why researchers should care. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 78–90. IEEE.

[29] Agarwal, S., Mahajan, R., Zheng, A., and Bahl, V. (2010). Diagnosing Mobile Applications in the Wild. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, HotNets, pages 22:1–22:6. ACM.

[30] Aggarwal, C. C. and Philip, S. Y. (2008). *A general survey of privacy-preserving data mining models and algorithms*. Springer.

[31] Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., and Memon, A. M. (2015). Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59.

[Amazon Mechanical Turk] Amazon Mechanical Turk. *https://mturk.com*.

[33] Anand, S., Naik, M., Harrold, M. J., and Yang, H. (2012). Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM.

[34] Android. Android Espresso. https://developer.android.com/training/testing/ui-testing/espresso-testing.html.

[35] Android. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[36] Android. UIAutomator. https://google.github.io/android-testing-support-library/docs/uiautomator/.

[37] Android (2016). Requesting permissions at run time). https://developer.android.com/training/permissions/requesting.html/.

[Android Observatory] Android Observatory. http://androidobservatory.org.

[Android-System Permissions] Android-System Permissions. http://developer.android.com/guide/topics/security/permissions.html.

[apisense] apisense. APISENSE. http://apisense.io.

[41] Applause (2015). Analysis: In search of the best apps for ios and android of 2015. https://arc.applause.com/2015/12/16/applause-analytics-state-of-the-app-store-and-google-play-2015/.

[42] Artzi, S., Kim, S., and Ernst, M. (2008). ReCrash: Making Software Failures Reproducible by Preserving Object States. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP, pages 542–565. Springer.

[aspectj] aspectj. AspectJ. https://eclipse.org/aspectj.

[44] Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. (2012). Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 217–228.

[45] Azim, M. T., Neamtiu, I., and Marvel, L. M. (2014). Towards Self-healing Smartphone Software via Automated Patching. In *29th ACM/IEEE International Conference on Automated Software Engineering*, ASE'14, pages 623–628.

[46] Azim, T. and Neamtiu, I. (2013). Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 641–660. ACM.

[47] Barrera, D., Kayacik, H. G., van Oorschot, P. C., and Somayaji, A. (2010). A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 73–84, New York, NY, USA. ACM.

[48] Bartel, A., Klein, J., Le Traon, Y., and Monperrus, M. (2012a). Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE'12, page 274–277, New York, NY, USA. ACM.

[49] Bartel, A., Klein, J., Monperrus, M., and Le Traon, Y. (2012b). Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In *ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis*.

[50] Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022.

[Calabash] Calabash. Calabash. Automated acceptance testing for mobile apps. http://calaba.sh/.

[52] Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., and Pezze, M. (2013). Automatic recovery from runtime failures. In *International Conference on Software Engineering*, pages 782–791. IEEE Press.

[53] Chen, N. and Kim, S. (2014). STAR: Stack Trace based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering*, 41:1–1.

[54] Chia, P. H., Yamamoto, Y., and Asokan, N. (2012). Is this app safe?: A large scale study on application permissions and risk signals. In *Proceedings of the 21st International Conference on World Wide Web*, WWW'12, pages 311–320, New York, NY, USA. ACM.

[55] Choi, K. and Chang, B.-M. (2015). A lightweight approach to component-level exception mechanism for robust android apps. *Computer Languages, Systems & Structures*, 44:283–298.

[56] Choi, W., Necula, G., and Sen, K. (2013). Guided gui testing of android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, volume 48, pages 623–640. ACM.

[57] Choudhary, S. R., Gorla, A., and Orso, A. (2015). Automated Test Input Generation for Android: Are We There Yet? In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440. IEEE.

[58] Clark, P. and Boswell, R. (1991). Rule induction with cn2: Some recent improvements. In *Machine learning—EWSL-91*, pages 151–163. Springer.

[59] Crittercism (2014). Mobile experience benchmark.

[Cypher] Cypher. http://docs.neo4j.org/refcard/2.0.

[61] Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176.

[62] Dang, Y., Wu, R., Zhang, H., Zhang, D., and Nobel, P. (2012). ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 1084–1093, Piscataway, NJ, USA. IEEE Press.

[63] Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

[64] Doug Sillars (2015). *High Performance Android Apps: Improve ratings with speed, optimizations, and testing.* O'Reilly Media, first edition.

[65] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011). Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA. ACM.

[66] Flurry (2015). Mobile addicts multiply across the globe. http://flurrymobile.tumblr.com/post/124152019870/mobile-addicts-multiply-across-the-globe.

[67] Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y., and Flora, P. (2010). Mining performance regression testing repositories for automated performance analysis. In *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, pages 32–41.

[68] Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y., and Flora, P. (2015). An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 159–168.

[69] Frank, M., Dong, B., Felt, A., and Song, D. (2012). Mining permission request patterns from android and facebook applications. In *Proceedings of the 12th IEEE International Conference on Data Mining*, ICDM'12, pages 870–875.

[70] Fu, B., Lin, J., Li, L., Faloutsos, C., Hong, J., and Sadeh, N. (2013). Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'13, New York, NY, USA. ACM.

[71] Gaudin, B., Vassev, E. I., Nixon, P., and Hinchey, M. (2011). A control theory based approach for self-healing of un-handled runtime exceptions. In *8th ACM international conference on Autonomic computing*, pages 217–220. ACM.

[getevent] getevent. Getevent tool. https://source.android.com/devices/input/getevent.html.

[73] Gomez, L., Neamtiu, I., Azim, T., and Millstein, T. (2013). Reran: Timing-and touch-sensitive record and replay for android. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 72–81. IEEE.

[74] Gómez, M. (2015). Debugging of Mobile Apps in the Wild Guided by the Wisdom of the Crowd. In Dig, D. and Dubinsky, Y., editors, *2nd ACM International Conference on Mobile Software Engineering and Systems. ACM Student Research Competition.*, Firenze, Italy. IEEE.

[75] Gómez, M., Martinez, M., Monperrus, M., and Rouvoy, R. (2015a). When App Stores Listen to the Crowd to Fight Bugs in the Wild. In *37th International Conference on Software Engineering (ICSE), track on New Ideas and Emerging Results*, Firenze, Italy. IEEE.

[76] Gómez, M., Rouvoy, R., Adams, B., and Seinturier, L. (2016a). Mining Test Repositories for Automatic Detection of UI Performance Regressions in Android Apps. In Robbes, R. and Bird, C., editors, *13th International Conference on Mining Software Repositories (MSR'16)*, Austin, Texas, United States. IEEE.

[77] Gómez, M., Rouvoy, R., Adams, B., and Seinturier, L. (2016b). Reproducing Context-sensitive Crashes of Mobile Apps using Crowdsourced Monitoring. In Flynn, L. and Inverardi, P., editors, *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'16)*, Austin, Texas, United States. IEEE.

[78] Gómez, M., Rouvoy, R., Monperrus, M., and Seinturier, L. (2015b). A Recommender System of Buggy App Checkers for App Store Moderators. In Dig, D. and Dubinsky, Y., editors, *Proceedings of the 2nd ACM International Conference on Mobile Software Engineering and Systems*, Firenze, Italy. IEEE.

[Google] Google. Google Cloud Test Lab. https://firebase.google.com/docs/test-lab/.

[googleanalytics] googleanalytics. Google Analytics. http://www.google.com/analytics.

[81] Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. (2014). Checking app behavior against app descriptions. In *ICSE'14: Proceedings of the 36th International Conference on Software Engineering*, Hyderabad (India), 31 May - 7 June.

[82] Ha, E. and Wagner, D. (2013). Do android users write about electric sheep? examining consumer reviews in google play. In *Proceedings of the IEEE Consumer Communications and Networking Conference*, CCNC'13.

[83] Haderer, N., Rouvoy, R., and Seinturier, L. (2013). Dynamic deployment of sensing experiments in the wild using smartphones. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, DAIS'13, pages 43–56.

[84] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18.

[85] Halpern, M., Zhu, Y., Peri, R., and Reddi, V. J. (2015). Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 215–224. IEEE.

[86] Hao, S., Liu, B., Nath, S., Halfond, W. G., and Govindan, R. (2014). Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 204–217. ACM.

[87] Herbold, S., Grabowski, J., Waack, S., and Bünting, U. (2011). Improved bug reporting and reproduction through non-intrusive gui usage monitoring and automated replaying. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 232–241. IEEE.

[88] Hewlett Packard (2015). Failing to Meet mobile App User Expectations: A Mobile User Survey. Technical report.

[89] Hu, C. and Neamtiu, I. (2011). Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST, pages 77–83. ACM.

[90] Hu, G., Yuan, X., Tang, Y., and Yang, J. (2014). Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*, page 18. ACM.

[91] Hu, Y., Azim, T., and Neamtiu, I. (2015). Versatile yet lightweight record-and-replay for android. In *ACM SIGPLAN Notices*, volume 50, pages 349–366. ACM.

[92] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition.

[93] Iacob, C. and Harrison, R. (2013). Retrieving and analyzing mobile apps feature requests from online reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR'13, pages 41–44, Piscataway, NJ, USA. IEEE Press.

[Java] Java. Javatm platform debugger architecture. http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/.

[95] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. (2012). Dr. android and mr. hide: Fine-grained permissions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 3–14, New York, NY, USA. ACM.

[96] Jin, W. and Orso, A. (2012). BugRedux: Reproducing Field Failures for In-house Debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484, Piscataway, NJ, USA. IEEE Press.

[97] Kechagia, M., Mitropoulos, D., and Spinellis, D. (2014). Charting the API minefield using software telemetry data. *Empirical Software Engineering*, pages 1–46.

[98] Kevoree (2016). Kevoree modeling framework). https://http://kevoree.org/.

[99] Khalid, H., Shihab, E., Nagappan, M., and Hassan, A. (2015). What do mobile app users complain about? *Software, IEEE*, 32(3):70–77.

[100] Kim, S., Zimmermann, T., and Nagappan, N. (2011). Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Proceedings of the 41st International Conference on Dependable Systems & Networks (DSN)*, pages 486–493. IEEE.

[101] Kittur, A., Chi, E. H., and Suh, B. (2008). Crowdsourcing user studies with mechanical turk. In *SIGCHI conference on human factors in computing systems*, pages 453–456. ACM.

[102] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012). GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72.

[103] Liang, C.-J. M., Lane, N. D., Brouwers, N., Zhang, L., Karlsson, B. F., Liu, H., Liu, Y., Tang, J., Shan, X., and Chandra (2014). Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing. In *Proceedings of the 20th International Conference on Mobile Computing and Networking*, MobiCom. ACM.

[104] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2013). API change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 477–487, New York, NY, USA. ACM.

[105] Linares-Vásquez, M., White, M., Bernal-Cárdenas, C., and Moran, K Poshyvanyk, D. (2015). Mining android app usages for generating actionable gui-based execution scenarios. In *12th IEEE Working Conference on Mining Software Repositories (MSR'15)*.

[106] Liu, Y., Xu, C., and Cheung, S.-C. (2014a). Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA. ACM.

[107] Liu, Y., Xu, C., and Cheung, S.-C. (2014b). Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA. ACM.

[108] Maalej, W., Kurtanovic, Z., Nabil, H., and Stanik, C. (2016). On the automatic classification of app reviews. *Requir. Eng.*, 21(3):311–331.

[109] Maalej, W. and Nabil, H. (2015). Bug report, feature request, or simply praise? on automatically classifying app reviews. In *23rd IEEE International Requirements Engineering Conference, RE 2015, Ottawa, ON, Canada, August 24-28, 2015*, pages 116–125.

[110] Maalej, W., Tiarks, R., Roehm, T., and Koschke, R. (2014). On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):31.

[111] Mabroukeh, N. R. and Ezeife, C. I. (2010). A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1):3.

[112] Machiry, A., Tahiliani, R., and Naik, M. (2013). Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, pages 224–234, New York, NY, USA. ACM.

[113] Mahmood, R., Mirzaei, N., and Malek, S. (2014). Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609. ACM.

[114] Mao, K., Harman, M., and Jia, Y. (2016). Sapienz: multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105. ACM.

[115] Martin Monperrus (2015). Automatic Software Repair: a Bibliography. Technical report.

[116] McCallum, A. K. (2002). MALLET: a machine learning for language toolkit. http://mallet.cs.umass.edu.

[117] McIlroy, S., Ali, N., and Hassan, A. E. (2016). Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empirical Software Engineering*, 21(3):1346–1370.

[118] Moawad, A., Hartmann, T., Fouquet, F., Nain, G., Klein, J., and Le Traon, Y. (2015). Beyond discrete modeling: a continuous and efficient model for iot. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 90–99. IEEE.

[119] Mojica, I. J., Nagappan, M., Adams, B., Berger, T., Dienst, S., and Hassan, A. E. (2015). An examination of the current rating system used in mobile app stores. *IEEE Software*.

[Neo4J] Neo4J. http://www.neo4j.org.

[121] Nguyen, T. A., Csallner, C., and Tillmann, N. (2013). Gropg: A graphical on-phone debugger. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1189–1192. IEEE.

[122] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang (2014). AR-Miner: mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14.

[123] Norris, J. R. (1998). *Markov chains*. Cambridge university press.

[124] Oliner, A. J., Iyer, A. P., Stoica, I., Lagerspetz, E., and Tarkoma, S. (2013). Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 10:1–10:14, New York, NY, USA. ACM.

[125] Ongkosit, T. and Takada, S. (2014). Responsiveness analysis tool for android application. In *Proceedings of the 2Nd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2014, pages 1–4.

[126] OpenSignal (2015). Android fragmentation visualized. http://opensignal.com/reports/2015/08/android-fragmentation/.

[127] Pagano, D., Juan, M. A., Bagnato, A., Roehm, T., Bruegge, B., and Maalej, W. (2012). FastFix: Monitoring control for remote software maintenance. In *34th International Conference on Software Engineering*, pages 1437–1438. IEEE Press.

[128] Pei, Y., Furia, C. A., Nordio, M., Wei, Y., Meyer, B., and Zeller, A. (2014). Automated fixing of programs with contracts. *Ieee transactions on software engineering*, 40(5):427–449.

[129] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., et al. (2009). Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM.

[130] Quinlan, J. R. (1993). *C4. 5: programs for machine learning*, volume 1. Morgan kaufmann.

[131] Radatz, J. (1990). Ieee standard glossary of software engineering terminology. *IEEE-SA Standards Board*.

[132] Ravindranath, L., Padhye, J., Agarwal, S., Mahajan, R., Obermiller, I., and Shayandeh, S. (2012). AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, pages 107–120.

[133] Röβler, J., Fraser, G., Zeller, A., and Orso, A. (2012). Isolating failure causes through test case generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, pages 309–319. ACM.

[134] Robinson, I., Webber, J., and Eifrem, E. (2013). *Graph Databases*. O'Reilly.

[Robotium API] Robotium API. http://robotium.googlecode.com/svn/doc.

[136] Roehm, T., Gurbanova, N., Bruegge, B., Joubert, C., and Maalej, W. (2013). Monitoring user interactions for supporting failure reproduction. In *21st International IEEE Conference on Program Comprehension (ICPC)*, pages 73–82. IEEE.

[137] Roehm, T., Nosovic, S., and Bruegge, B. (2015). Automated extraction of failure reproduction steps from user interaction traces. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 121–130. IEEE.

[138] Rosnow, R. L. (2008). *Beginning behavioral research: a conceptual primer*. {Pearson/Prentice} Hall, Upper Saddle River, {N.J}, 6th ed edition.

[139] Schroter, A., Bettenburg, N., and Premraj, R. (2010). Do stack traces help developers fix bugs? In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 118–121. IEEE.

[selendroid] selendroid. Selendroid. http://selendroid.io/. [Online; accessed Jan-2016].

[141] Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., and Keromytis, A. D. (2009). Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News*, 37(1):37–48.

[142] Sommerard, R. and Rouvoy, R. (2016). Towards privacy-preserving data dissemination in crowd-sensing middleware platform. In *11èmes journées francophones Mobilité et Ubiquité (UbiMob'16)*, page 6.

[splunk] splunk. SPLUNK. https://mint.splunk.com.

[spmf] spmf. SPMF: An open-source data mining library. http://www.philippe-fournier-viger.com/spmf/index.php. [Online; accessed Jan-2016].

[145] Statista (2015). Worldwide mobile app revenues in 2015, 2015 and 2020 (in billion u.s. dollars). https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/.

[146] Statista (2016). Global market share held by smartphone operating systems from 2009 to 2015). https://www.statista.com/statistics/263453/global-market-share-held-by-smartphone-operating-systems/.

[147] Statista (2016). Number of apps available in Google Play Store as of February 2016. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store. [Online; accessed June-2016].

[148] Strauss, A. and Corbin, J. (1994). Grounded theory methodology: An overview. In Denzin, N. K. and Lincoln, Y. S., editors, *Handbook of Qualitative Research*, pages 273–285+. Sage Publications, Thousand Oaks, CA.

[149] Vidas, T., Christin, N., and Cranor, L. (2011). Curbing android permission creep. In *Proceedings of the Web 2.0 Security and Privacy 2011 workshop*, W2SP'11, Oakland, CA.

[150] Wang, J. and Han, J. (2004). BIDE: Efficient mining of frequent closed sequences. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 79–90. IEEE.

[151] Wei, X., Gomez, L., Neamtiu, I., and Faloutsos, M. (2012). Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 31–40, New York, NY, USA. ACM.

[152] White, M., Linares-Vásquez, M., Johnson, P., Bernal-Cárdenas, C., and Poshyvanyk, D. (2015). Generating Reproducible and Replayable Bug Reports from Android Application Crashes. In *23rd IEEE International Conference on Program Comprehension (ICPC)*.

[153] Whittaker, J. A. (2000). What is software testing? and why is it so hard? *IEEE software*, 17(1):70–79.

[154] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman (2016). A Survey of App Store Analysis for Software Engineering.

[xamarin] xamarin. Xamarin Test Cloud. http://xamarin.com/test-cloud. [Online; accessed Jan-2016].

[156] Xamarin (2016). Understanding android api levels). https://developer.xamarin.com/guides/android/application_fundamentals/understanding_android_api_levels/.

[157] Xiao-Feng Li (2012). Quantify and Optimize the User Interactions with Android* Devices. https://software.intel.com/en-us/android/articles/quantify-and-optimize-the-user-interactions-with-android-devices.

[158] Xuan, J., Martinez, M., Demarco, F., Clément, M., Lamelas, S., Durieux, T., Le Berre, D., and Monperrus, M. (2016). Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*.

[159] Yang, D., Xue, G., Fang, X., and Tang, J. (2016). Incentive mechanisms for crowdsensing: Crowdsourcing with smartphones. *IEEE/ACM Transactions on Networking*, 24(3):1732–1744.

[160] Yang, S., Yan, D., and Rountev, A. (2013). Testing for poor responsiveness in android applications. In *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*, pages 1–6. IEEE.

[161] Yin, R. K. (2013). *Case study research: Design and methods*. Sage publications.

[162] Zaeem, R. N., Prasad, M. R., and Khurshid, S. (2014). Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST'14, pages 183–192, Washington, DC, USA. IEEE.

[163] Zeller, A. (2005). *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[164] Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200.

[165] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li (2016). MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications. Austin, Texas.

[166] Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schroter, A., and Weiss, C. (2010). What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618–643.

Appendix A

Users' Opinion and Preferences Survey

This appendix presents the survey design to evaluate the CROWDSEER approach from the users' perspective.

To evaluate CROWDSEER from the users perspective, we conducted a survey with closed questions. We ran the survey under two settings:

- lab setting with *in situ* participants,
- crowdsourcing setting with crowdsourced participants.

Lab Setting

In the lab setting, **33 *in situ* participants** performed a case study based evaluation. Figure A.1 shows the instructions of the case study provided to users. Figure A.2 provides the questionnaire that participants had to complete.

Crowdsourcing Setting

In the crowdsourcing setting, we recruited **600 participants** using the crowdsourcing platform *CrowdFlower*. We published the survey (which in-situ participants performed) as a job in *CrowdFlower*. Fig A.3 shows a screenshot of the *CrowdFlower* job.

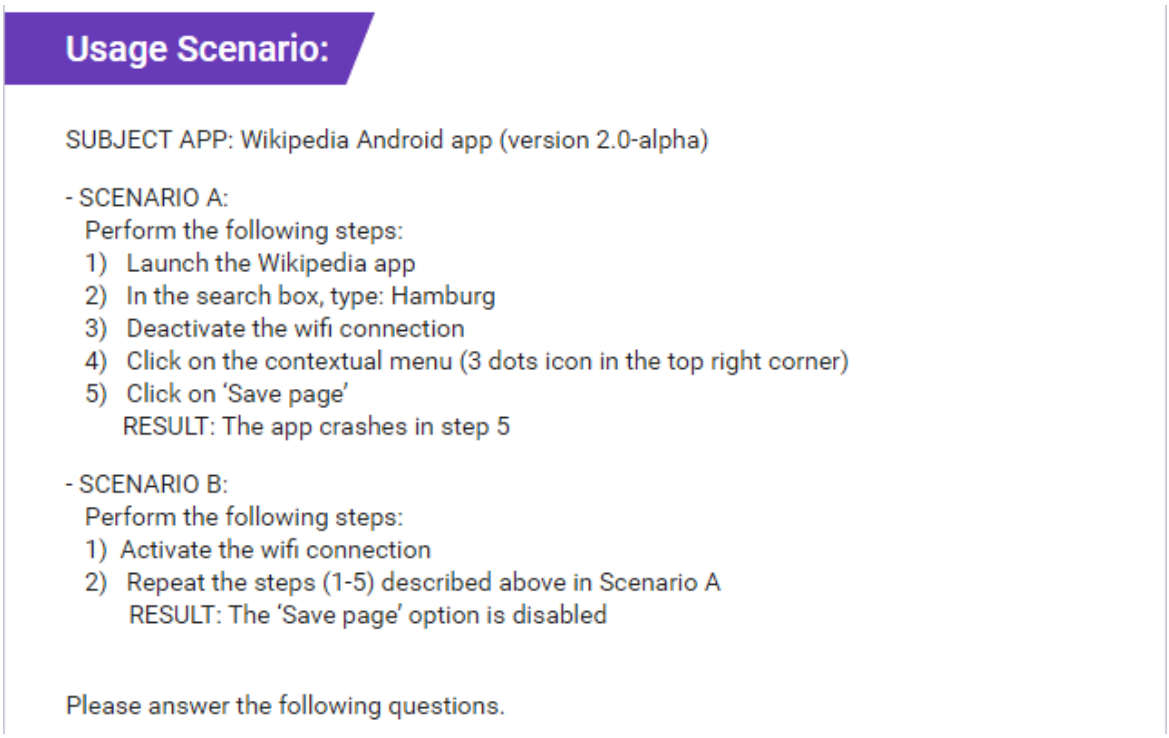


Fig. A.1 Description of the usage scenario

The CrowdFlower job contains two steps:

- *Step 1*: Participants have to watch the following video: https://www.youtube.com/watch?v=za2dhlRqhwI
- *Step 2*: After watching the video, participants have to complete the same questionnaire that in situ participants completed (cf. Fig. A.2).

Out of 600 participants we kept **495 answers** which passed our quality filters (i.e., validation questions, time threshold, click check). The crowdsourced participants belong to 75 different countries.

Figure A.4 provides demographic information of the crowdsourced participants: country, age, IT background and smartphone usage.

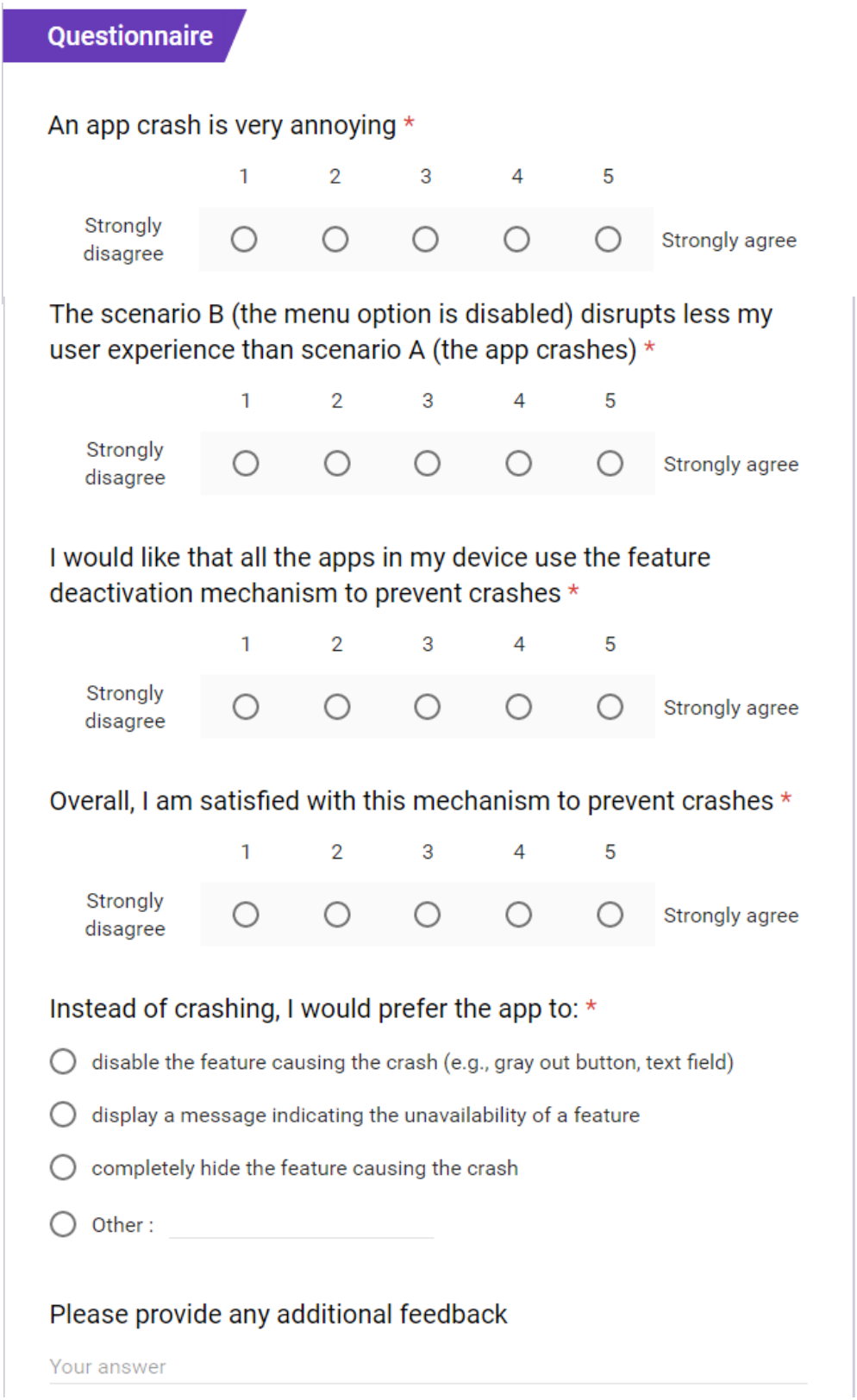


Fig. A.2 Set of Questions to Evaluate

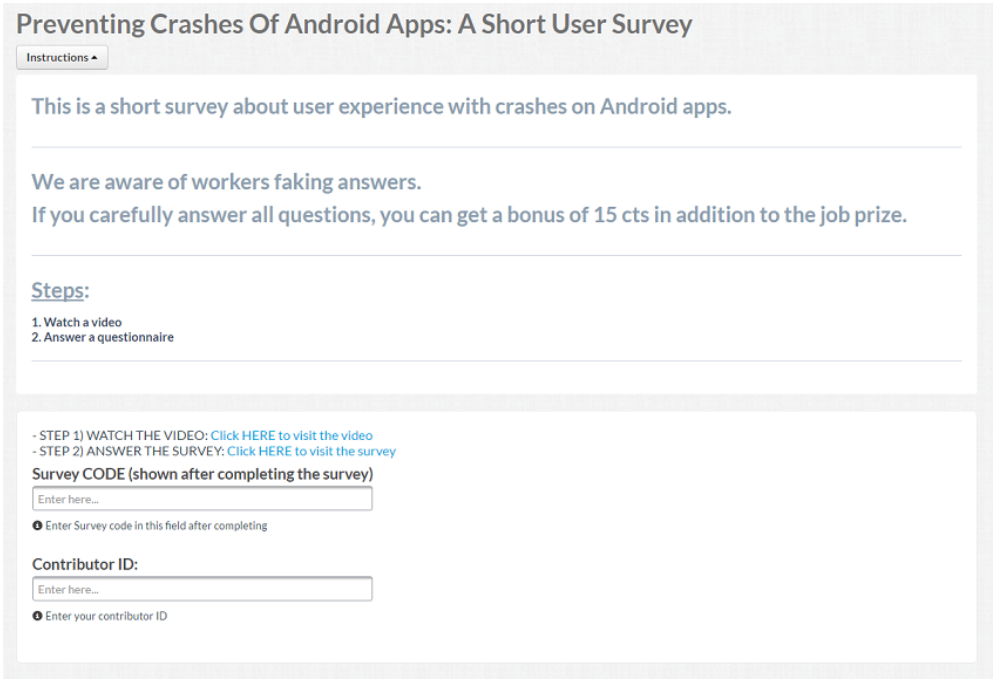
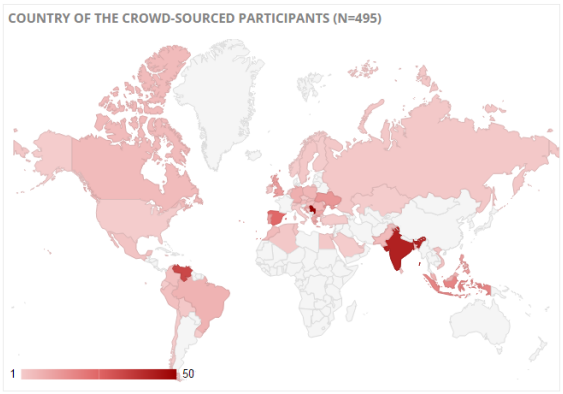
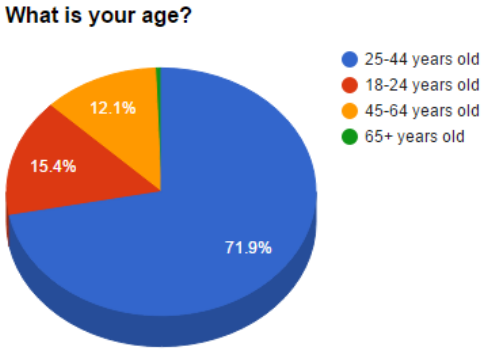


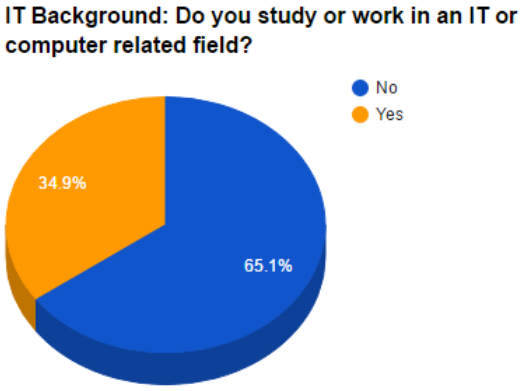
Fig. A.3 Crowdflower job



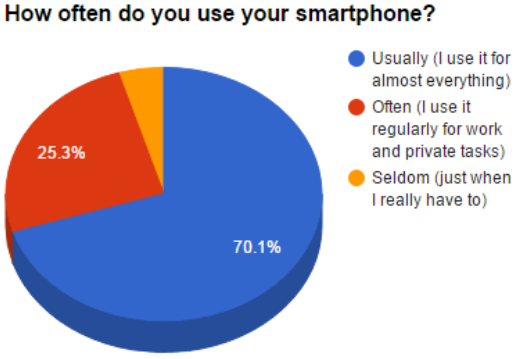
(a) Country



(b) Smartphone usage



(c) Age



(d) IT background

Fig. A.4 Demographic information of the crowdsourced participants (N=495)

Appendix B

App Developer Interview

This appendix presents the interview we designed to evaluate the CrowdSeer approach from the developers' perspective.

Interview Design

This interview aims to qualitatively evaluate CrowdSeer from the developers' perspective. The interview is divided in 4 parts. In the following, we illustrate the questions which composed the interview.

PART A. Brackground and Project Context

- For how long have you been developing mobile apps?
- Which mobile platform(s) are you targeting?
- Which is your role in the company?
- Which types of apps have you developed? How many users do your apps have?

PART B. Current Crash Detection and Fixing Practice

- Process. How do you currently detect when users experience crashes with your app?
- How do you proceed when users report crashes with your app?
- Reporting Tools. Do you use any crash reporting system? If yes, which one(s)? How do you use it? If not, why not?

- Crash types. Which kind(s) of crashes do you usually face with your apps? Do you detect them before or after release?
- Do you follow different strategies depending on the types of crashes? Notify users, new release, downgrade the version...
- Do you publish temporary patches to fix crashes before you provide the final fix of the app? If yes, why?
- Issues. Which step(s) (e.g., detection, reproduction, fixing) are more difficult to complete when facing a crash?

PART C. Feedback to Crowdseer

- General impression. How do you like the proposed solution (i.e., a library embedded in your app)? Why?
- How would you compare our approach with current crash reporting systems? Would you prefer a different solution? How?
- Interact with CrowdSeer. How CrowdSeed should proceed when a crash is detected and the "preventive patch" (feature to deactivate) is synthesized?
- How would you like to be notified when CrowdSeer synthesizes a patch? Adding a new issue in your Issue Tracker...
- Workflow integration. Integration with your development tools, as a separated tool... For example, Eclipse, Issue Tracker...

PART D. Overal satisfaction

- I find the automatic deactivation functionality really useful to reduce my app's crashes (1-5)
- I would like to include the automatic deactivation functionality to prevent crashes in all my developed apps (1-5)
- I find the CrowdSeer mobile library (1-5)
- I find the CrowdSeer web interface (1-5)
- Overall, I am very satisfy with this system (1-5)
- Add any additinal feedback (1-5)